COIN: An Efficient Indexing Mechanism for Unstructured Data Sharing Systems

Junjie Xie, Chen Qian, *Senior Member, IEEE*, *Member, ACM*, Deke Guo, *Senior Member, IEEE*, *Member, ACM*, Minmei Wang, *Graduate Student Member, IEEE*, Ge Wang, and Honghui Chen

***Abstract*— Edge computing promises a dramatic reduction in the network latency and the traffic volume, where many edge servers are placed at the edge of the Internet. Furthermore, those edge servers cache data to provide services for edge users. The data sharing among those edge servers can effectively shorten the latency to retrieve the data and further reduce the network bandwidth consumption. The key challenge is to construct an efficient data indexing mechanism no matter how the data is cached in the edge network. Although this is essential, it is still an open problem. Moreover, existing methods such as the centralized indexing and the DHT indexing in other fields fail to meet the performance demand of edge computing. This paper presents a COordinate-based INdexing (COIN) mechanism for the data sharing in edge computing. COIN maintains a virtual space where switches and data indexes are associated with their coordinates. Then, COIN distributes data indexes to indexing edge servers based on those coordinates. The COIN is effective because any query request from an edge server can be responded when the data has been stored in the edge network. More importantly, COIN is efficient in both routing path lengths and forwarding table sizes for publishing/querying data indexes. We implement COIN in a P4 prototype. Experimental results show that COIN uses 59% shorter path length and 30% less forwarding table entries to retrieve data indexes compared to using Chord, a well-known DHT solution.**

***Index Terms*— Data indexing, data sharing, edge computing, software-defined networking.**

I. Introduction

EDGE Computing has been proposed to shift computing and storage capacities from the remote Cloud to the network edge in close proximity to mobile devices, sensors, and end users [1], [2]. Meanwhile, it promises a dramatic reduction in network latency and traffic volume, tackling the key challenges for materializing 5G vision. Terms such as ‘cloudlets’, ‘micro data centers’, and ‘fog computing’ have been used in the literature to refer to similar edge-located services [3]–[6]. In edge computing, edge servers (also called as edge nodes) can perform computation offloading, data storage, data caching and data processing for edge users.

However, unlike Cloud data center servers, edge servers are usually widely geographically distributed and have heterogeneous computation and storage capacities [2], [3], [5]. In edge computing, when an edge user sends a data request, the request is first directed to the nearest edge server. If the edge server has cached the data, it will return the data to the edge user, otherwise, it will retrieve the data from the remote Cloud for the edge user. However, retrieving data from the Cloud would incur a large amount of backhaul traffic and the long latency. Furthermore, retrieving the data from those neighboring edge servers that have cached the required data can efficiently reduce the bandwidth consumption and the latency of request response, as shown in the literature [7], [8]. Therefore, there is an urgent need to study the data sharing among edge servers.

To enable the data sharing, the key challenge is to achieve the data index, which indicates the data location in the edge computing environment. However, it remains an open problem, and an efficient data indexing mechanism is very essential. Some earlier work about the data indexing in other computing environments is divided into three categories. The first one is the full indexing [9] where each edge node maintains a full index for all data in the edge network. The main drawback is that the bandwidth cost is too high to maintain the full index since each data location needs to be transferred to all edge nodes by the broadcast or the multicast. The second one is the centralized indexing [10] where a dedicated indexing server is needed to maintain all the data indexes. However, the centralized indexing suffers from the performance bottleneck and drawbacks in the fault-tolerance and the scalability. The last one is the Distributed Hash Table (DHT) indexing [11], which has been extensively studied in Peer-to-Peer (P2P) networks and could be a candidate solution for the data sharing in edge computing. However, our observation shows that the DHT indexing goes through a significantly longer path to retrieve a data index compared to the shortest path between the corresponding edge nodes.

Manuscript received March 21, 2019; revised January 15, 2021; accepted August 23, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Guan. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFE0207600, in part by the National Natural Science Foundation of China under Grant U19B2024 and Grant 62002284, and in part by the Tianjin Science and Technology Foundation under Grant 18ZXJMTG00290. *(Corresponding author: Deke Guo.)*

Junjie Xie is with the Institute of Systems Engineering, AMS, PLA, Beijing 100141, China (e-mail: xiejunjie06@gmail.com).

Chen Qian and Minmei Wang are with the Department of Computer Science and Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064 USA (e-mail: cqian12@ucsc.edu).

Deke Guo and Honghui Chen are with the Science and Technology Laboratory on Information Systems Engineering, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: guodeke@gmail.com).

Ge Wang is with the Department of Computer Science and Engineering, Xi’an Jiaotong University, Xi’an 710049, China (e-mail: wangge@stu.xjtu.edu.cn).

Digital Object Identifier 10.1109/TNET.2021.3110782

In this paper, we propose an efficient data indexing mechanism, called COordinate-based INdexing (COIN), for the data sharing in the edge computing environment. Those data from the Cloud and edge devices are locally cached in edge servers, and no global caching rules are required in the whole edge network. Therefore, for the data sharing of those unstructured data, this is an unstructured data sharing system. To achieve the COIN mechanism, the control plane of the network maintains a virtual 2-dimensional (2D) space where each switch is associated with a coordinate. Furthermore, each data index is also mapped into a coordinate in the virtual space. Then, the data index is stored in the indexing edge server that is directly connected to the switch whose coordinate is closest to the coordinate of the data index in the virtual space. The COIN is an extension of DHT. Those data indexes are distributed into edge servers based on the hash values of data indexes However, there are some differences between the traditional DHT and the COIN. The edge servers need to maintain the finger tables [11] for the lookup of data indexes under the traditional DHT. Meanwhile, for each requested data index, the lookup request needs to go through $log(n)$ overlay hops between edge servers. Under the COIN mechanism, only one overlay hop is needed to locate each data index with the support of programmable switches.

The COIN is effective because any query request from an edge server can be responded to when the data has been cached in the edge network. More importantly, the lookup speed shows the efficiency of the COIN mechanism, which achieves the shortest path lengths and the fewest forwarding table entries in switches to retrieve the data indexes. Furthermore, to enhance the robustness of the indexing systems, multiple data copies and multiple index copies are essential in the edge network. In this case, the key challenge is how to quickly retrieve the data index and the data item from the nearest edge server. To enable these advantages, our COIN mechanism embeds the path length between switches into the distance between coordinates in the virtual space. After that, the data requester can instantly retrieve the data index and the data from the nearest edge servers by comparing their distances in the virtual space without sampling all copies.

We conducted extensive experiments, using both P4 implementation and simulations, to evaluate the performance of the COIN mechanism. Experimental results show that the COIN mechanism uses 30% less forwarding table entries and 59% shorter path length to retrieve data indexes compared to using the well-known DHT indexing mechanism [11].

In summary, we make the following major contributions:

1) We propose a coordinate-based indexing mechanism (COIN) for the unstructured data sharing in the edge computing environment. The COIN is effective and efficient due to not only the shortest path length but also the fewest forwarding table entries in switches for searching data indexes.
2) We design an efficient distance embedding algorithm, Algorithm 1, which makes any edge server can instantly select the optimal index copy and the optimal data copy when multiple index copies and multiple data copies are available in the edge network.

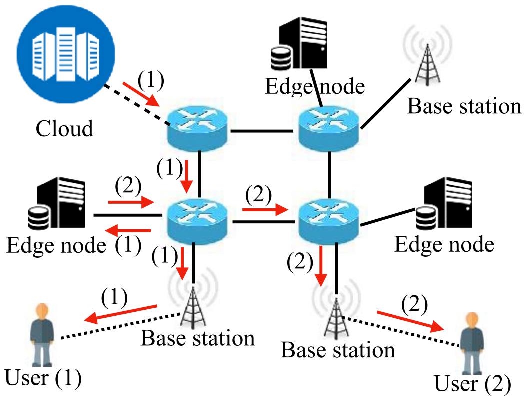


Fig. 1. Retrieving data in the edge computing environment.

3) We implement the COIN mechanism in P4, and further evaluate its performance through large-scale simulations. The experiment results show the efficiency and effectiveness of the COIN mechanism for not only searching the data index but also retrieving the data.

The rest of this paper is organized as follows. In section II, we introduce the motivation and the design overview of this paper. In Section III, we detail the COIN mechanism. Section IV presents the optimization designs of the COIN mechanism for multiple data copies and multiple index copies. In Section V, we evaluate the performance of the COIN mechanism based on a small-size testbed and large-scale simulations. We introduce the related work and conclude this paper in Section VI and Section VII, respectively.

II. Motivation and Overview

A. Motivation

Edge computing is to offload computing and storage to the network edges so as to enable computation-intensive and latency-critical applications. The promised gains of edge computing have motivated extensive efforts in both academia and industry on developing the technology [2], [12], [13]. In edge computing, each edge node caches some data to provide services for those edge users located in a given area. As shown in Fig. 1, when *user(1)* sends a data request, the request is first directed to the edge node that is nearest to the Access Point (e.g. a base station). If the edge node has cached the data, it will immediately return the data to the edge user, otherwise, it will retrieve the data from the remote Cloud for the user. Meanwhile, the data will be also cached in the corresponding edge node. It is no doubt that retrieving the data from the remote Cloud consumes too much bandwidth and incurs significantly long latency.

In edge computing, the need for data sharing mainly comes from two folds. One is that many popular contents in the Cloud are asynchronously and repeatedly requested by different edge users. It has been predicted by Cisco that mobile video streaming will occupy up to 72% of the entire mobile data traffic by 2019 [12]. One unique property of such services is that the content requests are highly concentrated. Motivated by this fact, wireless content caching was proposed to avoid the frequent retrieval of the same contents [14]–[16]. Another one is that the edge servers can deliver those data generated

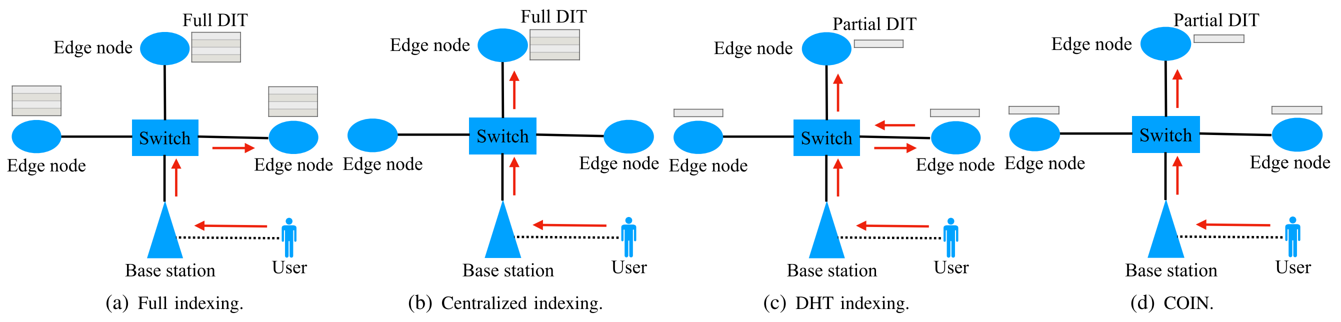


Fig. 2. Packet forwarding under different data indexing mechanisms.

by some edge devices to other edge devices that are located in different geographical areas. It is estimated that tens of billions of edge devices will be deployed in the near future, and their processor speeds are growing exponentially, following Moore's Law [12]. A large amount of edge devices will produce the data that is first cached in those edge servers [9]. The development of the edge devices further promotes the data sharing and the repeated using of the data among edge servers. Therefore, there is an urgent need to study the data sharing among edge servers.

Meanwhile, the data sharing among multiple edge nodes can efficiently reduce the latency of data retrieval and the bandwidth consumption in the backhaul network [7]. Given a data sharing framework, when an edge node receives a data request, it will first lookup if the data has been cached in itself or other edge nodes in the edge network. If the data has been cached in the edge network, retrieving the data from the closer edge node is more efficient than from the remote Cloud. As shown in Fig. 1, when *user(2)* requests the same data as *user(1)*, *user(2)* can retrieve the data from the corresponding edge node instead of the remote Cloud. Furthermore, when retrieving a data from other edge nodes fails to meet the low-latency demand, the edge node closest to the edge user will cache the data again. After that, there are multiple data copies in the edge network. In this case, the key challenge is how to quickly obtain the data copy that is closest to the data requester. Therefore, the data sharing framework needs to efficiently support multiple data copies. Meanwhile, for the robustness of the indexing system, multiple index copies could be maintained for each shared data. It is essential to optimize the indexing system for supporting multiple index copies.

B. Design Overview

To achieve the data sharing among edge nodes, the key challenges are where to place data indexes, how to search data indexes, and how to retrieve the data after getting a data index. Our COIN mechanism mainly solves the first two problems. The routing problem from the data requester to the data location that is indicated by the data index is orthogonal to our work. When a data requester gets a data index that indicates a data location, the data can be retrieved from the location by using the shortest path routing or other more efficient routing schemes. In the edge network, there are multiple edge nodes where **each edge node consists of multiple edge servers**. Furthermore, those edge nodes cache data items from the Cloud or edge devices to provide services for edge users. We use the following terms to describe the data sharing framework throughout the paper:

1) An *ingress edge server* refers to the closest edge server to a Base Station (BS). All data requests from the BS are firstly forwarded to this edge server.
2) A *storing edge server* refers to an edge server that stores some shared data items.
3) An *indexing edge server* refers to the edge server that stores the indexes of cached data at storing edge servers. Note that each edge node determines one of its edge servers as the indexing edge server.
4) An *Indirect edge server* refers to an intermediate edge server that forwards any query request of a data index, not including the ingress edge server and the indexing edge server. Note that the indirect edge server is only employed under the traditional DHT indexing.

We explain the design choices for the data indexing and compare those choices with representative alternative designs to illustrate why we make those choices. To efficiently support the data indexing in edge computing, a direct design is to maintain a full Data Indexing Table (DIT) of all shared data in each edge node in the edge network. As shown in Fig. 2(a), on top of the full indexing mechanism, each edge node can quickly know if a data item exists in the edge network. However, the disadvantage of the full indexing is that the bandwidth cost of maintaining the full indexing is too huge. When an edge node caches a new data item, it needs to publish the data location to all edge nodes in the edge network.

The second choice is to choose a dedicated edge server to provide the centralized indexing service for all shared data in the edge network as shown in Fig. 2(b). In this scenario, the dedicated indexing server stores all data indexes, and each edge node forwards the data request to the unique indexing server. That is, only the dedicated indexing edge server needs to store the full DIT. However, an obvious flaw of this design is that the centralized indexing server will become the performance bottleneck. Furthermore, it also suffers from worse fault tolerance and load balance.

The third design is the DHT indexing mechanism, which has been extensively studied in peer-to-peer (P2P) networks [10], [11], [17], [18]. The DHT indexing is a distributed indexing mechanism, and each indexing edge server just stores partial DIT. Meanwhile, under the traditional DHT indexing mechanism, each edge server maintains a finger table [11] for the lookup of data indexes. The DHT indexing mechanism

TABLE I
COMPARISON OF DIFFERENT INDEXING MECHANISMS

Indexing mechanism	Lookup speed	Memory Scalability	Request load balancing	Bandwidth cost
COIN	Median	Good	Good	Low
DHT indexing	Slow	Good	Good	Median
Centralized indexing	Median	Bad	Bad	Low
Full indexing	Fast	Bad	Good	High

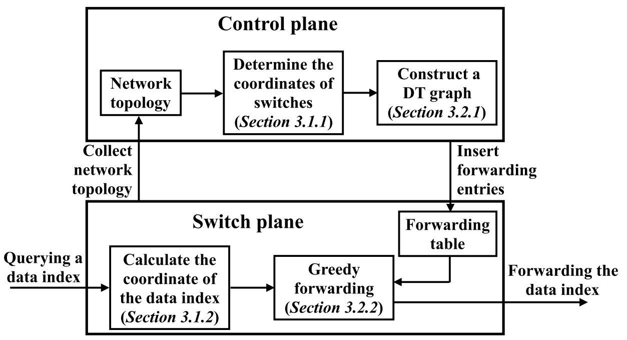

Control plane
Network topology
Determine the coordinates of switches (Section 3.1.1)
Construct a DT graph (Section 3.2.1)
Collect network topology
Insert forwarding entries
Switch plane
Forwarding table
Querying a data index
Calculate the coordinate of the data index (Section 3.1.2)
Greedy forwarding (Section 3.2.2)
Forwarding the data index

Fig. 3. The framework of the COIN mechanism over a software-defined edge network.

employs multiple overlay hops to retrieve a data index where each overlay hop means the shortest path between two edge servers. More precisely, for any query, the searching process usually involves $log(n)$ forwardings [18] where n is the number of edge nodes in the edge network. That is, the ingress edge server could forward each incoming packet to a series of intermediate indirect edge servers before reaching the final indexing edge server, as shown in Fig. 2(c), where each indirect edge server stores the finger table to lookup the data index. It is no doubt that the longer path increases the query processing latency, server load and consumes more internal link capacity in the edge network.

In this paper, our solution is a COordinate-based INdexing (COIN) mechanism, which just takes one overlay hop to search the data index as shown in Fig. 2(d). Furthermore, it achieves the benefits of the distributed data indexing, and needs less forwarding entries at each switch to support the data indexing than the DHT indexing mechanism. The features of different indexing mechanisms are concluded in Table I. Note that our COIN mechanism fully utilizes the advantages of software-defined networking (SDN) [19], [20], where the control plane can collect the network topology and state including switch, port, link, and host information [21]. When we apply the principle of SDN to the edge computing, the network is called a Software-Defined Edge Network (SDEN). Fig 3 shows the framework of the COIN mechanism, including the main functions in the control plane and the switch plane. In SDN, the network management is logically centralized in the control plane consisting of one or multiple controllers [22] which generate forwarding table entries for switches. The switches in the switch plane only forward packets according to the installed entries derived from the controller.

To achieve the COIN mechanism, the control plane maintains a virtual 2D space where each switch is associated with a coordinate. As shown in Fig 3, the control plane first collects the network topology from the switch plane, and then the coordinates of switches are calculated in Section III-A.1. The control plane constructs a Delaunay Triangulation (DT) graph [23], [24] in Section III-B.1 to connect those points, which indicate switches' coordinates in the virtual space. Further, the control plane inserts forwarding entries into the forwarding tables of switches where each forwarding entry indicates the coordinate of a neighboring switch. More precisely, the index of each shared data is also assigned to a coordinate in the virtual space based on its identifier in Section III-A.2. Then, the data index is greedily forwarded to the switch whose coordinate is the nearest to that of the data index in the virtual space in Section III-B.2. Finally, the switch forwards the data index to the only indexing edge server among all directly connected edge servers. Note that the performance and scalability of the control plane is the key to the COIN mechanism. Much research has been conducted to improve the performance and scalability of the control plane [20], [25], [26].

III. COORDINATE-BASED INDEXING

A. Determining Coordinates

1) Determining the Coordinates of Switches: The control plane can obtain the network topology and state by collecting switch, port, link, and host information [19], [21]. Then, the shortest path matrix between switches can be firstly calculated by the control plane. However, the key challenge is how to calculate the coordinate matrix of n points where the shortest path lengths between n switches can be indirectly reflected by the distances between points in the virtual space. In other words, we need to solve the problem of finding a point configuration that represents a given scalar-product matrix [27]. In matrix notation, this amounts to solving the equation:

$$B = XX', \tag{1}$$

where X is the $n \times m$ coordinate matrix of n points in m-dimensional space, and X' is the transpose of matrix X.

Every $n \times n$ matrix B of real numbers can be decomposed into a product of several matrices. The eigendecomposition can be constructed for most matrices, but always for symmetric ones. Formally,

$$B = Q \Lambda Q', \tag{2}$$

where Q is orthonormal (i.e., $QQ' = Q'Q = I$) and Λ is diagonal.

Every $n \times m$ matrix X can be decomposed into

$$X = P \Phi Q', \tag{3}$$

Equation (3) indicate a singular value decomposition (SVD). Where P is an $n \times n$ orthonormal matrix, (i.e., $P'P = I$), Φ is an $n \times m$ singular value matrix, and Q is an $m \times m$ orthonormal matrix, (i.e., $Q'Q = I$).

Assume that we know the decomposition of X as given in Formula (3). Then,

$$XX' = P \Phi Q' Q \Phi P' = P \Phi \Phi P' = P \Phi^2 P', \tag{4}$$

which is just the eigendecomposition of XX' based on Equation (2). This proves that the eigenvalues of XX' are all nonnegative because they consist of ϕ_i^2 and squared numbers are always nonnegative. Furthermore, suppose that we do an eigendecomposition of $B = Q\Lambda Q'$. We know that scalar product matrices are symmetric and have nonnegative eigenvalues based on Equations (2) and (4). Therefore, we may write $B = (Q\Lambda^{1/2})(Q\Lambda^{1/2})' = XX'$, where $\Lambda^{1/2}$ is a diagonal matrix with diagonal elements $\lambda_i^{1/2}$. Thus, $X = Q\Lambda^{1/2}$ gives coordinates that reconstruct B.

Algorithm 1 Calculate the Coordinates of Switches in the Virtual Space While Achieving the Distance Embedding

```
Require: The shortest path matrix $L$.
Ensure: The coordinates of the switches $U$.
1: Compute the squared ditance matrix $L^{(2)} = [l_{ij}^2]$.
2: Construct the scalar product matrix $B$ by multiplying the squared ditance matrix $L^{(2)}$ with the matrix $J = I - \frac{1}{n}A$. That is $B = -\frac{1}{2}JL^{(2)}J$, where $n$ is the number of switches, and $A$ is the squared matrix with all elements are 1. This procedure is called double centering.
3: Determine the $m$ largest eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$ and corresponding eigenvectors $e_1, e_2, \ldots, e_m$ of the matrix $B$ (where $m$ is the number of dimensions).
4: The coordinates of the switches $U = Q_m\Lambda_m^{1/2}$, where $Q_m$ is the matrix of $m$ eigenvectors and $\Lambda_m$ is the diagonal matrix of $m$ eigenvalues of the matrix $B$, respectively.
```

Based on the above analysis, we design the embedding algorithm of path lengths to calculate the coordinates of switches in the virtual space as shown in Algorithm 1. Algorithm 1 follows the principle of the classic multidimensional scaling [28], which can preserve the Euclidean distances between coordinates as well as possible. First, Algorithm 1 takes an input matrix giving network distances between pairs of switches, which is known to the control plane of the network [21]. The shortest path matrix $L = [l_{ij}]$, where l_{ij} is the length of the shortest path between the i^{th} and j^{th} switches. Then, Algorithm 1 utilizes the fact that the coordinate matrix can be derived by eigenvalue decomposition from $B = UU'$ where the scalar product matrix B can be computed from the distance matrix L by using the double centering [28] in Step 2 of Algorithm 1. Last, the coordinates of the switches U in the virtual space are obtained by multiplying eigenvalues and eigenvectors in Step 4 of Algorithm 1. Based on the Algorithm 1, the coordinates of switches in the virtual space can be determined. After that, Algorithm 1 can preserve the network distances between switches as well as possible. In detail, the coordinates of switches $\{u_1, u_2\}$ is in the output U of Algorithm 1. The real coordinates of switches $\{x_1, x_2\}$ is in the matrix X in Equation (1). Thus, $||u_1 - u_2||^2 \approx ||x_1 - x_2||^2$.

2) Determining Coordinates for Data Indexes: The coordinate of a data index is achieved by the hash value $H(d)$ of the identifier of the data index d. In this paper, we adopt the hash function, *SHA-256* [29], which outputs a 32-byte binary value. Note that other hash functions can also be used. Meanwhile, in the case of a hash collision, it just means that two or more data indexes are assigned to the same coordinate and stored in the same indexing edge server. Furthermore, the hash value $H(d)$ is reduced to the scope of the 2D virtual space. We only use the last 8 bytes of $H(d)$ and convert them to two 4-byte binary numbers, x and y. We limit that the coordinate value ranges from 0 to 1 in each dimension. Then, the coordinate of a data index in 2D is a two-tuple $(\frac{x}{2^{32}-1}, \frac{y}{2^{32}-1})$. The coordinate can be stored in decimal format, using 4 bytes per dimension. Hereafter, for any data identifier, d, we use $H(d)$ to represent its coordinate.

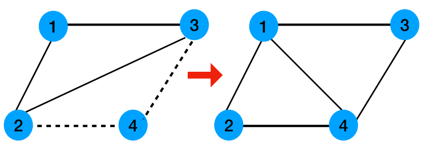


Fig. 4. The formulation of the DT graph.

B. Publishing Data Indexes

To publish data indexes, it is necessary to introduce the DT graph. A unique feature of the DT graph is that no other points exist in the circumcircle of any triangle. As shown in Fig. 4, the left graph is not a DT graph because point 4 is in the circumcircle of triangle $<1, 2, 3>$. However, the right graph is a DT graph. Recall that greedy routing on a DT graph provides the property of guaranteed delivery, which is based on a rigorous theoretical foundation [23], [30]. To achieve the guaranteed delivery of the COIN mechanism, the control plane first constructs a DT graph, which connects all switches' coordinates in the virtual space. After that, each switch greedily forwards a data index to its neighbor, whose coordinate is closest to the coordinate of the data index.

1) DT Construction: Given a set of switches and their coordinates in a set of points P, we adopt the randomized incremental algorithm [31] to construct the DT $DT(P)$ in the 2D virtual space. First, an appropriate triangle boundingbox is constructed to contain P. Then, those points in P are inserted in random order, and a DT corresponding to the current point set is maintained and updated throughout the whole process. Last, the boundingbox and relative triangles that contains any vertex of the boudingbox triangle are removed. Meanwhile, it is necessary to ensure that the union of all simplices in the triangulation is the convex hull of those points.

Consider that $DT(v_1, v_2, \ldots, v_{i-1})$ formed by inserting all previous points $v_1, v_2, \ldots, v_{i-1}$ is already a DT. The change caused by inserting v_i is adjusted and $DT(v_1, v_2, \ldots, v_{i-1}) \cup v_i$ is made a new $DT(v_1, v_2, \ldots, v_i)$. The adjustment process is as follows. First, we determine which triangle (or edge) v_i falls on, and then connect v_i with the three vertices of the triangle to form three triangles (or connect the vertices of two triangles of the common edge to form four triangles). Since the newly generated edges and the original edges may not be Delaunay edges, we would make a flipping [32] to make them all Delaunay edges to get $DT(v_1, v_2, \ldots, v_i)$. For example, in Fig. 4, there is a $DT(1, 2, 3, 4)$. We change the common edge $<2, 3>$ to the common edge $<1, 4>$ to produce two triangles that do meet the Delaunay condition when two original triangles do not meet the Delaunay condition. This operation is called a flipping. Based on the above analysis, the time complexity of the DT construction is $O(n)$ where n is the number of switches in the network.

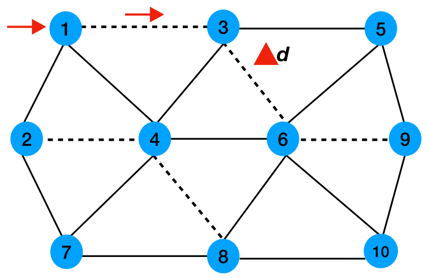


Fig. 5. The principle of forwarding data indexes.

Algorithm 2 The Greedy Forwarding at Switch u

```
1: For each physical neighbor $v$, $R_v = Dis(v, d)$, Euclidean distance from $v$ to $d$ in the virtual space;
2: For each DT neighbor $\tilde{v}$, $R_{\tilde{v}} = Dis(\tilde{v}, d)$, Euclidean distance from $\tilde{v}$ to $d$.
3: $R_{v^*} = \min\{R_v, R_{\tilde{v}}\}$;
4: **if** $R_{v^*} < Dis(u, d)$ **then**
5:     Forward the packets to $v^*$ directly if $v^*$ is a physical neighbor or by the virtual link to $v^*$ if $v^*$ is a DT neighbor;
6: **else**
7:     Forwards the data index $d$ to its indexing edge server;
8: **end if**
```

It is worth noting that a switch could not be directly connected with its DT neighbor. As shown in Fig. 5, switch 3 is a DT neighbor of switch 1 in the virtual space. However, there is no a directly physical link between switch 1 and switch 3 in the data plane. Therefore, to achieve the guaranteed delivery, each switch maintains two kinds of forwarding entries. The first one makes it forward packets to its physical neighbors, while the other one makes it forward requests to its DT neighbors.[1] The switches that are not directly connected to an indexing edge server would not participate in the construction of the DT. Those switches are just used as the intermediate switches to transfer data indexes to those DT neighbors.

For a switch u, the forwarding table F_u is used to forward packets to DT neighbors. **Each entry in F_u is a 4-tuple** $<src, pred, succ, des>$, which is a sequence of switches with src and des being the source and destination switches of a path, and $pred$ and $succ$ being the predecessor and successor switches of switch u in the path. A tuple in F_u is used by u for message forwarding from src to des. For a specific tuple t, we use $t.src$, $t.pred$, $t.succ$, and $t.des$ to denote the corresponding switches in the tuple t. In the next section, we will introduce how to forward a data index to an indexing edge server based on the DT graph.

2) Forwarding Data Indexes: The switches are associated with their coordinates in a virtual space. A switch knows the coordinates of itself, its physical neighbors and its DT neighbors. The switch can obtain the coordinates of such switches based on its forwarding table entries where we utilize the P4 switch [33], [34]. Multiple match-action tables are declared in the P4 switch where the standard table includes two properties: key and action [34]. Meanwhile, the action could include some parameters, which are provided by the control plane. Recall that the coordinates of switches are calculated by the control plane in Section III-A.1. Then, the control plane converts the coordinates of the switches into the forwarding entries and inserts those forwarding entries into the corresponding switches. More precisely, the control plane inserts the parameters of an action that include x and y values of a coordinate into a match-action table through a *table_add* command. After that, in each match-action stage, the switch can calculate the distance between a pair of coordinates of a neighboring switch and a data index in the virtual space.

When a data item is cached by a storing edge server. The edge server will publish the corresponding data index to an indexing edge server. The data index is firstly sent to a switch. The switch, say u, uses the virtual coordinates of its physical and DT neighbors and the coordinate $p = H(d)$ of the data index d to compute estimated distances. Note that a DT neighbor could also be its physical neighbor. But, here, the DT neighbors are those neighbors except for the physical neighbors. Each switch performs the greedy forwarding based on the DT graph constructed by the control plane in Section III-B.1. The greedy forwarding at switch u is shown in Algorithm 2. For each physical neighbor v, switch u computes the estimated distance $R_v = Dis(v, d)$, which is the Euclidean distance from v to d in the virtual space. For each DT neighbor $\tilde{v}$, switch u computes the estimated distance from $\tilde{v}$ to d by $R_{\tilde{v}} = Dis(\tilde{v}, d)$. Switch u selects the neighbor switch v^* that makes $R_{v^*} = \min\{R_v, R_{\tilde{v}}\}$. If $R_{v^*} < Dis(u, d)$, switch u sends the packet to v^* directly if v^* is a physical neighbor or by the virtual link to v^* if v^* is a DT neighbor. As shown in Fig. 5, the DT neighbor switch 3 of switch 1 is closer to the point d in the virtual space. Therefore, switch 1 forwards the data index d to switch 3. Note that there is a virtual link between switch 1 and switch 3. The real path traversed by the data index is the path $<1, 4, 3>$ for the virtual link $<1, 3>$ in Fig. 5. If $R_{v^*} < Dis(u, d)$ is not satisfied, switch u is closest to the coordinate of the data index. Then, switch u directly forwards the data index to its indexing edge server. In the following paragraph, we further introduce the data transmission in a virtual link.

Forwarding to a DT neighbor. A switch connects to one of its DT neighbors by a virtual link based on the DT graph in Section III-B.1. A virtual link is a physical path, which is utilized to transfer packets from a switch to one of its DT neighbors. When switch u receives a packet that is being forwarded in a virtual link, the packet is processed as follows. Assume that a switch u has received a data index d to forward. Switch u stores it with the format: $d = <d.des, d.src, d.relay, d.index>$ in a local data structure, where $d.des$ is the DT neighboring switch of the source switch $d.src$, $d.relay$ is the relay switch, and $d.index$ is the payload of the data index. When $d.relay \neq null$, the data index d is traversing a virtual link.

The forwarding at switch u is specified by two conditions and the corresponding actions in Table II. When the first condition $u = = d.des$ is found to be true, switch u is the DT neighboring switch, which is the endpoint of the virtual

[1]Hereafter the DT neighbors of a switch do not include its physical neighbors.

TABLE II
THE FORWARDING IN A VIRTUAL LINK AT SWITCH u

	Condition	Action
1	u==$d.des$	Continue to forward the data to the neighbor switch which is closest to the coordinate of the data index.
2	u==$d.relay$	Find tuple t in F_u with $t.des$==$d.des$; revise $d.relay$=$t.succ$ according to F_u; and transmit the data to $d.relay$.

link. Then, switch u will continue to forward the data index d to its neighbor, which is closest to the coordinate of the data index in the virtual space. In particular, the second condition is for handling messages traversing a virtual link. When $u = =d.relay$, switch u first finds tuple t from the forwarding table F_u with $t.des = =d.des$ where F_u is defined in Section III-B.1. Then, switch u revises $d.relay = t.succ$ based on the matched tuple t. The last step in switch u is to transmit the data index to $d.relay$. Based on this setting, messages can be forwarded to the DT neighbor of a switch. Last, the data index will be forwarded to the switch whose coordinate is closest to the coordinate of the data index in the virtual space, and then, the switch forwards the data index to its indexing edge server. In addition, a global minimum may not be unique. The tie will be broken by ranking the x coordinate, then y coordinate. Furthermore, Theorem 1 states that the COIN mechanism always succeeds to forward the data index to a unique switch based on the DT graph in the virtual space, and this theorem is proved in Appendix 1.

Theorem 1: Based on the DT graph in the virtual space, the COIN mechanism always succeeds to forward a data index d to a unique switch, which is closest to the data index in the virtual space.

The complexity analysis. Algorithm 2 shows how a switch forwards a data index in the switch plane. Our COIN mechanism fully utilizes the advantages of SDN [19]. In SDN, the network management is logically centralized in the control plane consisting of one or multiple controllers, which generate forwarding table entries for switches. The switches in the switch plane only forward packets according to the installed entries derived from the controller. Under our COIN mechanism, the control plane constructs a DT graph in Section III-B.1 to connect those points, which indicate the switches' coordinates in the virtual space. Further, the control plane inserts the forwarding entries into the forwarding tables of switches where each forwarding entry indicates the coordinate of a neighboring switch. Then, the data index is greedily forwarded to the switch whose coordinate is the nearest to that of the data index in the virtual space based on Algorithm 2. The key to establish Algorithm 2 is the DT graph, which is constructed in the control plane in Section III-B.1. The time complexity to construct the DT graph is $O(n)$, where n is the number of switches in the network. Meanwhile, the communication overhead is mainly caused between the control plane and the switch plane.

The control plane will send the forwarding table entries to the switches. This process will consume the network bandwidth of the links between the control plane and the switch plane. Therefore, the communication overhead is related to the number of forwarding table entries. It is worth noting that our COIN mechanism incurs less communication overhead than other schemes. Because the COIN mechanism needs less forwarding table entries than other indexing mechanisms, which has been proved by the experiment results. Under the COIN mechanism, Algorithm 2 is only dependent on the DT graph. Given the network topology, the DT graph can be constructed. After that, the forwarding table entries can be determined for each switch. Therefore, the process to install the forwarding table entries will not affect the performance of the data index lookup. In addition, the time complexity of Step 1 and Step 2 in Algorithm 2 is $O(\gamma)$, where γ is the number of neighbors of switch u. The time complexity of Step 3 is $O(1)$. The time complexity of Step 5 is $O(\mu)$, where μ is the number of hops from switch u to its DT neighbor. The time complexity of Step 7 is $O(1)$. Thus, the time complexity of Algorithm 2 is $O(max\{\gamma, \mu\})$, where $max\{\gamma, \mu\}$ is a constant number.

C. Storing Data Indexes

Those data indexes are stored in indexing edge servers with the following pattern, <*key*, *value*>, where the *key* is the data identifier, the *value* indicates the address of the corresponding storing edge server. An indexing edge server can store a large amount of data indexes. To reduce the latency of searching indexes, the HashMap [35] is utilized to store those data indexes in each indexing edge server. The HashMap<key, value> implementation provides constant-time performance for the basic operations (get and put). The function *put(key, value)* is employed to store a pair of key and value into the hashmap. Furthermore, the function *get(key)* is used to get the data location to which the specified key is mapped. If this hashmap contains no mapping for the key, it returns *null*.

D. Querying Data Indexes

So far, we have introduced the procedure of publishing a data index. Under the COIN mechanism, querying a data index is similar to the publishing procedure. The querying procedure is also to use the identifier of data index, and each switch greedily forwards the querying request to the switch whose coordinate is closest to the coordinate of the data index in the virtual space. That is, the switch uses the same method shown in Section III-B.2 to determine the indexing edge server, which will respond to the querying request. Then, the indexing edge server returns the corresponding data index that indicates the data location in the edge network. Last, the data requester can retrieve the data using the shortest path routing or other routing schemes, which is orthogonal to this work.

IV. OPTIMIZATION DESIGNS FOR MULTIPLE COPIES

A. The Scheme of Multiple Data Copies

An ingress edge server will also cache a data copy when retrieving the data from other edge servers fails to meet the low latency demand in the edge computing environment. Meanwhile, some popular contents could be concentratedly requested. In this case, for the load balance, multiple data copies are cached in the edge network. To enable multiple

Algorithm 3 Select the Optimal Index Copy

```
Require: The number of index copies $\alpha$, the data index $d$, and the switch $u$ that is directly connected with the ingress edge server.
Ensure: The identifier $d+(i-1)$ of the optimal index copy.
1: For each index copy $j$, $R_j = Dis(u, d+(i-1))$, Euclidean distance from $u$ to $d+(i-1)$ in the virtual space;
2: $R_i = \min\{R_j\}$, $1 \leq j \leq \alpha$;
3: The ingress edge server transfers the identifier $d+(i-1)$ to switch $u$.
```

data copies, the data structure <*Key*, *Vector*> is used to store the data index, where there are multiple elements in the *Vector* and each element indicates a location of data copy. When an ingress edge server caches a data copy again, it publishes the data index to its indexing edge server. Then, the indexing edge server finds the corresponding *Key* and adds a new element into the corresponding *Vector*. The key challenge is how to fully exploit multiple data copies to provide better services for edge users. That is, each ingress edge server hopes to retrieve the data copy from the closest storing edge server.

However, the path length to each data copy is unknown. A direct method is to send the probing packets to all data copies, but it incurs longer latency and more bandwidth consumption. Recall that we have embedded the distances between switches into the virtual space in Section III-A.1. Then, the path length between two edge servers can be estimated by calculating the distances between the two corresponding switches that are directly connected to the two edge servers. To enable this advantage, the data location is indicated by the address of the storing edge server and the coordinate of the switch that is directly connected to the storing edge server. After that, when an ingress edge server retrieves a data index with the format <*Key*, *Vector*>, which includes the locations of multiple data copies. The ingress edge server can instantly select a storing edge server to retrieve the data with the shortest path by comparing the distances between the corresponding switches in the virtual space.

B. The Scheme of Multiple Index Copies

At current, we only consider one data index for each shared data. However, for the fault tolerance or the load balance, the edge network could store multiple data indexes for each shared data. That is, the data indexes of a shared data can be stored in multiple different indexing edge servers. To enable this, we further optimize the COIN mechanism under multiple index copies. We have described that the indexing edge server for a data index is determined by the hash value $H(d)$ of the data index in Section III where d is the identifier of the data index. Now, to enable multiple index copies, the indexing edge server for the i_{th} index copy is determined by the hash value $H(d+i-1)$. Note that the data identifier is a string. The serial number i of the index copy is converted to a character, and then, the string of the data identifier and the character are concatenated. Last, the hash value of the new string uniquely determines the indexing edge server that will store the index copy. Furthermore, when there are α index copies, the indexing edge server that stores the α_{th} index copy is uniquely determined by the hash value $H(d+\alpha-1)$.

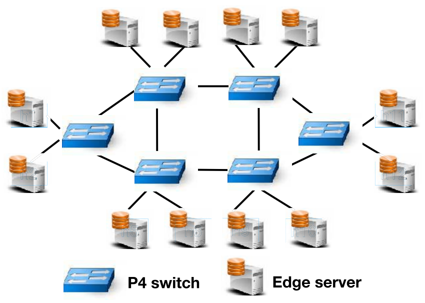


Fig. 6. The network topology consists of 6 P4 switches and 12 edge servers.

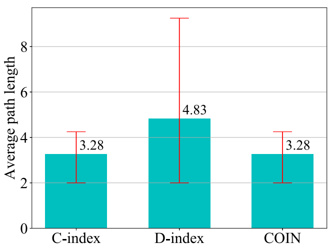
The key challenge is how to quickly obtain the optimal index copy that is closest to the ingress edge server when multiple index copies are available. It means that the path of retrieving the index is shortest. However, achieving this goal is hard because we just know the identifier of the data index, and we do not require the ingress edge server to store other more information. Recall that the coordinate of the data index is calculated based on the hash value of each index copy. Then, the data index is forwarded to the switch whose coordinate is closest to the coordinate of the data index in the virtual space, and the indexing edge server directly connected to the switch will store the data index. In this case, to select the optimal index copy without probing all index copies, the key enabler is to reflect the path length between two switches by the distance between the corresponding points in the virtual space, which has been achieved in Section III-A.1. As shown in Algorithm 3, the ingress edge server can select the optimal index copy. Furthermore, it will transfer the string of the optimal index copy to switch u, which is directly connected with the ingress edge server. After that, the switch can forward the querying request of a data index to the nearest index copy based on the coordinates of the switches and the index copy. Therefore, under the COIN mechanism, the ingress edge server can quickly select the optimal index copy that achieves the shortest path length to retrieve the data index.

V. Performance Evaluation

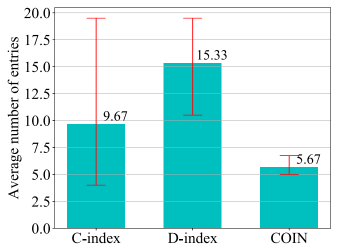
In this section, we first implement and evaluate our COIN mechanism on a small-size testbed. Furthermore, we evaluate the effectiveness and efficiency of the COIN mechanism through large-scale simulations.

A. Implementation and Prototype-Based Experiments

We have built a testbed, which consists of 6 P4 switches and 12 edge servers as shown in Fig. 6. We implement the centralized indexing (*C-index*), the DHT indexing (*D-index*) [11] and our COIN mechanisms on our testbed, and further compare the performances of the three different indexing mechanisms. We implement the COIN mechanism, including all switch plane and control plane features described in Section II-B, where the switch plane is written in P4 [33], and the functions in the control plane are written in Java. The P4 compiler generates Thrift APIs for the controller to insert

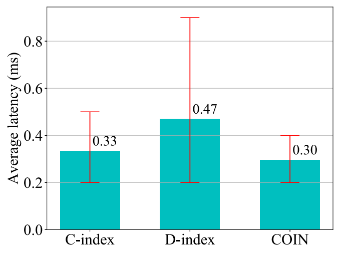


(a) The path lengths of retrieving indexes.

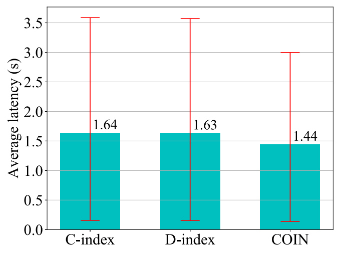


(b) The number of forwarding entries for searching data indexes.

Fig. 7. The path lengths and the numbers of forwarding entries under different indexing mechanisms in a small-scale testbed.



(a) The latency of retrieving indexes.



(b) The latency of retrieving data.

Fig. 8. The latencies of retrieving the indexes and the data under different indexing mechanisms in a small-scale testbed.

the forwarding entries into the switches. The P4 switch supports a programmable parser to allow new headers to be defined where multiple match+action stages [34] are designed in series to achieve the neighboring switch whose coordinate is closest to the coordinate of the data index. The P4 switch calculates the distance from a neighboring switch to the data index in the virtual space in a match+action stage.

We first compare the path lengths and the number of forwarding table entries under different indexing mechanisms. The path lengths from all edge servers to the indexing edge server are calculated, and then, the average path lengths under different indexing mechanisms are obtained. In the following figures, each error bar is constructed using a 95% confidence interval of the mean. As shown in Fig. 7(a), the average path length achieved by our COIN mechanism is close to the average path length achieved by the C-index mechanism and is obviously shorter than the average path length achieved by the D-index mechanism. The C-index mechanism uses the shortest path between an ingress edge server and the indexing edge server to retrieve a data index. The D-index mechanism retrieves a data index while employing multiple overlay hops where one overlay hop is related to the shortest path between two edge servers. However, our COIN mechanism only employs one overlay hop to retrieve the data index.

Furthermore, we compare the number of forwarding table entries for searching data indexes under different indexing mechanisms where C-index and D-index mechanisms forward the packets by matching the source and destination addresses. Fig. 7(b) shows the average number of forwarding table entries per switch under different indexing mechanisms. As shown in Fig. 7(b), our COIN mechanism achieves fewer forwarding table entries in switches than the other two indexing mechanisms. It is because that under our COIN mechanism, the number of forwarding table entries in each switch is just related to the number of its neighboring switches. However, under C-index and D-index mechanisms, the number of forwarding table entries increases as the increase of the number of flows in the edge network.

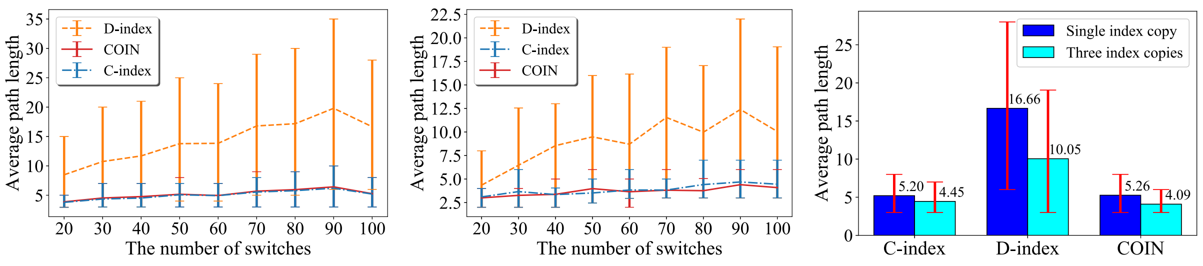
Meanwhile, we evaluate the impact of the multiple copies on the performance of the COIN mechanism. We have stored $10,000$ data items in the edge network where the data size varies from *10KB* to *10MB*. Two data copies and two index copies are maintained for each shared data. Then, we randomly generate some data requests and test the latencies of retrieving the data indexes and the data copies under different indexing mechanisms. Fig. 8(a) shows that the COIN mechanism achieves the least average latency to retrieve the index copy among the three indexing mechanisms since the COIN mechanism can quickly select the optimal index copy based on the coordinates of switches and index copies in the virtual space. We can find that the gap between the C-index mechanism and the COIN mechanism is small in Fig. 8(a). It is mainly because that the scale of our testbed is small. Furthermore, we test the impact of multiple data copies on the latency of retrieving the data. As shown in Fig. 8(b), the COIN mechanism incurs the least average latency of retrieving the data among the three mechanisms without sampling all data copies. Note that the average latencies of retrieving the data for the C-index and the D-index mechanisms are very close in Fig. 8(b) because they all retrieve the data using the shortest path routing after obtaining the data index.

B. Setting of Large-Scale Simulations

In simulations, we use BRITE [36] with the Waxman model to generate synthetic topologies at the switch level where each switch connects to 10 edge servers. We vary the number of switches from 20 to 100. In this case, the number of edge servers varies from 200 to 1000 in edge networks. Note that our COIN mechanism can be scaled to larger networks, which have the same size as software-defined networks [19]. Meanwhile, it is worth noting that the advantage of the COIN mechanism will be more obvious when the network size increases. The compared methods are as follows.

1) The C-index mechanism: it means that a dedicated edge server is selected as the global indexing server in the edge network where each ingress edge server uses the shortest paths to retrieve the data indexes.
2) The D-index mechanism: it uses the storage principle of Chord [11], a well-known DHT solution, to distribute the data indexes in the edge network.
3) The COIN mechanism: it stores the data index based on the coordinates of the switches and the data indexes in the virtual space, as shown in Section III.

We adopt two performance metrics to evaluate different indexing mechanisms including **the path length** and **the number of forwarding table entries** for retrieving data indexes. Note that each entry in the forwarding table indicates the coordinate of a neighboring switch under the COIN mechanism. Under C-index and D-index mechanisms, packets are forwarded by matching the source and destination addresses. In this case, the forwarding entry includes the next hop for forwarding packets, and a new entry is added into the



(a) A single index copy for each shared data. (b) Three index copies for each shared data. (c) There are 100 switches in the edge network.

Fig. 9. The average path lengths for retrieving data indexes under different indexing mechanisms.

forwarding table for a new pair of source and destination. To reduce the number of forwarding table entries, we use the wildcard forwarding entries for C-index and D-index mechanisms. In the following figures, each error bar is constructed using a 95% confidence interval of the mean. That is, the lower bar denotes the 2.5th percentile, and the upper bar presents the 97.5th percentile. Thus, the probability between the lower bar and the upper bar is 95%. Furthermore, we evaluate the impact of multiple copies on the path lengths of retrieving the data items and the data indexes.

C. The Path Lengths for Retrieving Data Indexes

In this section, we evaluate the path lengths for retrieving the data indexes under different indexing mechanisms. The path lengths from all edge servers to the indexing edge server are calculated, and then, the average path length is obtained.

Fig. 9(a) shows that the average path length of retrieving the data indexes are almost the same for COIN and C-index mechanisms. Note that the C-index mechanism uses the shortest path from an ingress edge server to the dedicated indexing server to retrieve the data index. Meanwhile, we can see that COIN and C-index mechanisms achieve significantly shorter path lengths than the D-index mechanism from Fig. 9(a). The average path length under the D-index mechanism has an obvious increase as the increase in the number of switches in Fig. 9(a). However, the increase is slow for COIN and C-index mechanisms when the number of switches changes.

In Fig. 9(a), the results are achieved when only one index copy is maintained for each shared data. Furthermore, we evaluate the change of the average path length when there are three index copies for each shared data. In this case, we test the path length for each index copy, and the path length of the shortest path is recorded for each indexing mechanism under each network setting. The experiment results are shown in Fig. 9(b), which shows almost the same trend as Fig. 9(a). That is, the average path length for retrieving the data index under the COIN mechanism is close to the average path length achieved by the C-index mechanism and is obviously shorter than the average path length under the D-index mechanism. It is worth noting that the C-index mechanism is a centralized indexing mechanism and suffers from the performance drawbacks in the fault tolerance and the scalability.

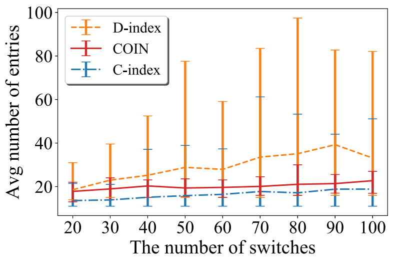
Fig. 9(c) shows that more index copies result in shorter path lengths for retrieving the data indexes under the three indexing mechanisms. Meanwhile, we can see that the impact of the index copies on the path length under the D-index mechanism is more obvious than the other two mechanisms. However, the D-index mechanism still needs a longer path to retrieve the data index than COIN and C-index mechanisms. As shown in Fig. 9(c), our COIN mechanism employs the shortest path to retrieve the data index than D-index and C-index mechanisms when three index copies are available for each shared data. More precisely, our COIN mechanism employs the average 68% and 59% shorter path lengths than the D-index mechanism when there are only one index copy and three index copies, respectively.

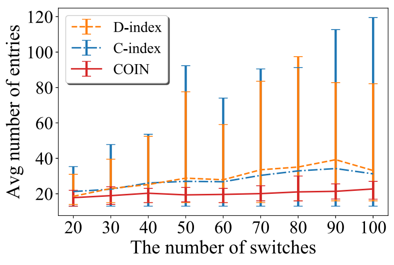
D. Forwarding Entries for Retrieving Indexes

In this section, we evaluate the number of forwarding table entries for searching the data indexes under different indexing mechanisms. For C-index and D-index mechanisms, we utilize the wildcard forwarding entries to significantly reduce the number of forwarding table entries.

Fig. 10(a) shows the change trend of the number of forwarding table entries as the increase of the number of switches under different indexing mechanisms. Each point in Fig. 10(a) indicates the average number of forwarding table entries over all switches under each network setting. We can see that, for C-index and D-index mechanisms, the average number of forwarding table entries increases as the increase in the number of switches from Fig. 10(a). However, the average number forwarding table entries of our COIN mechanism is almost independent of the network size since it is only related to the number of neighboring switches for each switch. Meanwhile, we can see that the upper error bars for the C-index mechanism are significantly higher than our COIN mechanism from Fig. 10(a). It is because that the C-index mechanism employs the shortest path routing to forward data indexes. In this case, some switches are frequently used in most of shortest paths, and then, a large amount of forwarding table entries are inserted into those switches.

The result in Fig. 10(a) is achieved when there is only one index copy for each shared data. Furthermore, Fig. 10(b) shows the average number of forwarding table entries for different indexing mechanisms when three index copies are stored for each shared data. In this scenario, we can see that the average number of forwarding entries for our COIN mechanism is the least among the three indexing mechanisms, as shown in Fig. 10(b). For the D-index mechanism, the average number of forwarding entries decreases when the number of switches varies from 90 to 100. The reason is that the network topologies are generated independently under different network sizes.


(a) A single index copy for each shared data.


(b) Three index copies for each shared data.

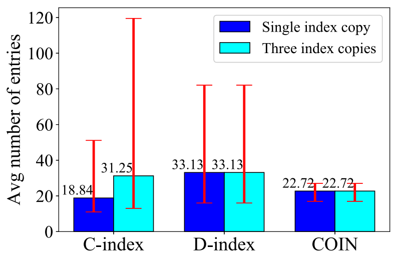
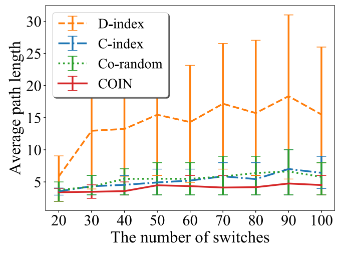

(c) There are 100 switches in the edge network.

Fig. 10. The number of forwarding table entries under different indexing mechanisms.


(a) The path lengths for retrieving indexes.

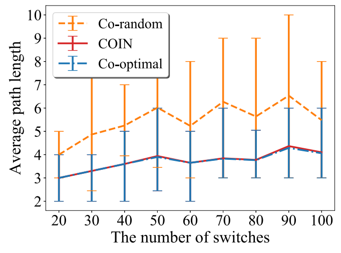

(b) The path lengths for retrieving data.

Fig. 11. The impact of multiple copies on the average path lengths.

Fig. 10(c) shows the impact of the index copies on the average number of forwarding entries under different indexing mechanisms, where 100 switches are deployed in the edge network. We can see that, for the C-index mechanism, the increase of the number of index copies causes the increase in the number of forwarding table entries from Fig. 10(c). It is because that multiple indexing servers are maintained to support multiple index copies under the C-index mechanism. However, more index copies have no impact on the number of forwarding table entries for D-index and COIN mechanisms since they are distributed indexing mechanisms. Furthermore, our COIN mechanism uses 30% less forwarding table entries compared to the well-known distributed D-index mechanism.

E. The Impact of Multiple Copies

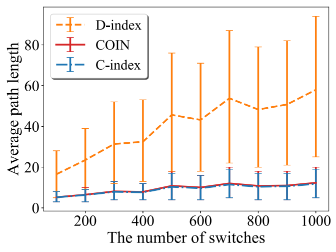
In this section, we further evaluate the impact of multiple copies on the path lengths of retrieving the data indexes and the data items. First, we test the impact of multiple index copies. Here, three index copies are maintained for each shared data. C-index and D-index mechanisms randomly select one index copy to retrieve the data index. The Co-random mechanism also employs the coordinate-based indexing mechanism, but randomly select an index copy to retrieve the data index. In Fig. 11(a), the path lengths of retrieving the data indexes under Co-random and C-index mechanisms are very close, and they are obviously shorter than the path lengths under the D-index mechanism. Furthermore, we can see that our COIN mechanism employs the shortest path to retrieve the data index than the other three indexing mechanisms under any network size from Fig. 11(a). That is, the experiment results show that retrieving the data index from the nearest index copy in the virtual space incurs obviously shorter path length than retrieving the data index from a randomly selected index copy without sampling all index copies.

Furthermore, we evaluate the impact of multiple data copies on the path length of retrieving a data item where three data copies are stored for the same data in the edge network. Here, we compared three methods of retrieving the data. The Co-random mechanism utilizes the coordinate-based indexing mechanism but randomly selects a data copy. The results of the Co-optimal mechanism are achieved by testing the path lengths to all data copies, and only the shortest path length is recorded for each data item. As shown in Fig. 11(b), the average path length of retrieving a data under the COIN mechanism is very close to the average path length under the Co-optimal mechanism and is significantly shorter than the result of the Co-random mechanism. In those experiments, we find that in rare cases, the COIN mechanism fails to achieve the optimal result, where the distance to the storing edge server employed by the COIN mechanism is very close to the distance to the storing edge server used by the Co-optimal mechanism. Therefore, it also means that the path length achieved by the COIN mechanism is just a little longer than the optimal path in those rare cases. It is worth noting that the COIN mechanism achieves the optimal path length in most cases without sampling all data copies. It is because that the path lengths between switches are embedded into the distance between the corresponding points in the virtual space in Section III-A.1.

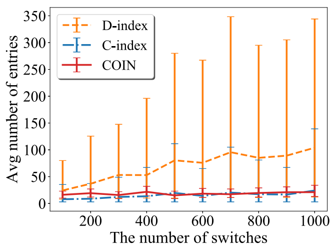
F. The Scalability of the COIN Mechanism

Note that the control plane needs to maintain a DT graph in the virtual space. When the scale of the network is large, the control plane needs a little longer computation time to construct the DT graph. After that, each switch conducts greedy forwardings without the interactions with the control plane, and the greedy forwardings can effectively reduce the load of the control plane. Furthermore, much research has been investigated to improve the performance and scalability of the control plane [20], [25], [37], [38]. To evaluate the scalability of the COIN mechanism, we further carry out experiments on large-scale networks where the number of switches varies from 100 to 1000.

Fig. 12(a) shows the impact of the network size on the average path length under different indexing mechanisms. The changing trend in Fig. 12(a) is similar to Fig. 9(a). In Fig. 12(a), we can see that the average path length achieved by our COIN mechanism is almost the same with that of the C-index mechansim and is obviously shorter than the average path length achieved by the D-index mechanism. Note that each ingress edge server uses the shortest path routing to



(a) The average path lengths for retrieving data indexes.



(b) The average number of forwarding table entries for retrieving data indexes.

Fig. 12. The performance comparison of different indexing mechanisms in large-scale networks.

forward requests to the indexing edge server under the C-index mechanism. However, the C-index mechanism is a centralized indexing mechanism, which suffers from poor scalability and performance bottleneck. Furthermore, Fig. 12(b) presents the impact of the network size on the average number of forwarding table entries under different indexing mechanisms. Fig. 12(b) shows a similar trend to Fig. 10(a). We can see that the average number of forwarding table entries under our COIN mechanism is significantly less than that of the D-index mechanism from Fig. 12(b). Therefore, our COIN mechanism has all the advantages of the distributed indexing mechanism while employing significantly shorter path lengths and less forwarding table entries than the well-known distributed indexing mechanism.

VI. Related Work

In this paper, we do not address data caching algorithms or cache consistency maintenance but focuses on the benefits of data cache sharing in edge computing. The edge computing aims to push the function of Clouds towards the network edges to dramatically reduce the network latency and traffic volume [1]–[3], [5]. At current, the edge computing has attracted much research where the major research problems include the deployment of edge servers, the network architecture, the mobility management, the caching mechanism, and so on.

Content cache in edge computing. In edge computing, edge servers perform computing offloading, data storage, caching and processing, as well as distribute request and delivery service [2]. The stored data in edge servers mainly comes from two folds. One is to cache the data from the Cloud to provide low-latency services for edge users. Another is to temporarily store the data produced by the edge users. All the data can be shared among multiple edge servers to provide services for more edge users. Motivated by this fact, wireless content caching was proposed in [15] to avoid frequent replication for the same contents by caching them at BSs. Predicting users' behavior, and proactively caching the users' content in the edge of the network also shows that further gains can be obtained in terms of backhaul savings and user satisfaction [7]. Tran *et al.* investigated collaborative multi-bitrate video caching and processing [39], where multiple bitrate versions of a video can be delivered so as to adapt to the heterogeneity of user capabilities and the varying of network condition. At current, much research has been conducted to study how to cache the data into edge servers. However, there is still a lack of research about the data sharing among edge servers. Therefore, in this paper, we investigate how to efficiently share the cached data among edge servers.

Data sharing in other computing environments. Although there are some studies about the data sharing in P2P networks and Web Proxies, we present why those studies are not enough to solve the corresponding problems in edge computing. The sharing of caches among Web proxies is an important technique to reduce Web traffic and alleviate network bottlenecks. Fan *et al.* proposed a new protocol called "summary cache" [9], where each proxy keeps a summary of the cache directory, and checks those summaries for potential hits before sending any queries. Iyer *et al.* presented a decentralized, peer-to-peer web cache called Squirrel [40]. Bo *et al.* proposed reference architecture for P2P systems [17] that focuses on the data indexing technology required to support resource locating. A P2P index can be local [41], centralized [10] or distributed [11]. With distributed index-based search scheme, pointers towards the target reside at several nodes, and distributed indexes are used in most P2P designs nowadays. We have compared those indexing schemes in Section II-B. For prior distributed indexing mechanisms, the main drawback is that they employ too long paths to retrieve data indexes, and further, that incur long latencies and more bandwidth consumption.

Peer data sharing among edge devices. At current, there are also some research about the peer data sharing in edge devices (e.g., smartphones). Song *et al.* proposed Peer Data Sharing (PDS) that enables mobile devices to quickly discover what data exist in nearby peers and retrieve desired data from possibly multiple edge devices [42]. Furthermore, Huang *et al.* considered caching fairness for peer data sharing among edge devices [43]. They proposed two caching algorithms to achieve the fair workload among selected caching nodes for data sharing in pervasive edge environments. However, there is still a lack of research about the data sharing among edge servers, which can provide more opportunities for peer data sharing among edge devices.

VII. Conclusion

In edge computing, edge servers need to cache the data to provide services for edge users and many emerging applications. The data sharing among edge servers can effectively shorten the latency of retrieving the data and further reduce the network bandwidth consumption. A key challenge to achieve this goal is to provide an efficient data indexing mechanism no matter how the data is cached in the edge computing environment. The COIN solves this challenging problem, and attractive features of COIN include its routing simplicity, provable correctness, shorter path length, and less forwarding table entries. Our experimental results confirm that the effectiveness and efficiency of the COIN mechanism. We believe that COIN will be a valuable component of edge computing.

Appendix

Appendix 1: **The proof of Theorem 1**

Based on the DT graph in the virtual space, the COIN mechanism always succeeds to forward a data index d to a unique switch, which is closest to the data index in the virtual space.

Proof: First, the coordinate of data index d is achieved by hashing the data identifier, $p = Hash(d)$. We prove this theorem by showing that every vertex u in the DT has a neighbor that is strictly closer to p than u is. Thus, at each routing step, the data index gets closer to p. After at most n steps, the data index reaches switch w^*, which is closest to p.

We use P to denote the set of switches' coordinates, and $DT(P)$ denotes the DT graph consisting of those points. Consider the Voronoi diagram $VD(P)$ is the straight line face dual of the Delaunay triangulation $DT(P)$ [31]. Let Ω be a metric space with distance function ϕ. Assume that there are n switches. The coordinate of a switch $(w_k)_{1\leq k\leq n}$ be a point in the space Ω. If $\phi(r, W) = inf\{\phi(r, w)|w\in W\}$ denotes the distance between the point r and the subset W, then we define a region R_k associated with the site w_k as follows.

$$R_k = \{r\in\Omega|\phi(r, w_k)\leq\phi(r, w_j), j = 1, \ldots, n, j\neq k\} \quad (5)$$

That is, the region R_k is the set of all points in Ω whose distance to w_k is not greater than their distance to the other sites w_j, where j is any index different from k. Accordingly, those regions are called Voronoi cells, and the diagram is a general Voronoi diagram [44].

Consider the Voronoi diagram $VD(P)$ of the vertices of P and let e be the first edge of $VD(P)$ intersected by the directed line segment (u, p). Note that e is on the boundary of two Voronoi cells, one for u and one for some other vertex v, and the supporting line of e partitions the plane into two open half planes $h_u = \{r : \phi(r, u)<\phi(r, v)\}$ and $h_v = \{r : \phi(r, v)<\phi(r, u)\}$. The edge $(u, v) \in DT(P)$ because the Delaunay triangulation $DT(P)$ is straight line face dual of the Voronoi diagram $VD(P)$. Therefore, switch u will forward the data index to switch v when $p \in h_v$. At last, the data index d will be forwarded to switch w^* closest to p.

Note that two special cases could occur. In the first scenario, point p could be on an edge e of $VD(P)$. In this case, we suppose the edge $(u, v) \in DT(P)$ is intersected with e. Therefore, u and v have the same distance to p and are closer to p than all other points in $DT(P)$. Assume that the data index is first forwarded to switch u. Switch u finds that its neighbor v has the same distance to p, and then compares their coordinates. We use (x_1, y_1) and (x_2, y_2) to denote the coordinates of switches u and v, respectively. If $(x_1 < x_2)$, then $w^* = u$. If $(x_1 > x_2)$, then $w^* = v$, and switch u forwards the data index d to switch v. If $(x_1 == x_2)$ and $y_1 < y_2$, then $w^* = u$. If $(x_1 == x_2)$ and $y_1 > y_2$, then $w^* = v$, and switch u forwards the data index d to switch v. In the second scenario, point p is inside a triangular in $DT(P)$ and is also an endpoint of $VD(P)$. In this case, the three points of the triangular have the same distance to p. Meanwhile, the three points are mutual neighbors in $DT(P)$. Therefore, using the same method, we rank the x coordinate, then y coordinate to determine the switch w^*.

Thus, Theorem 1 is proved. ■

REFERENCES

[1] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing a key technology towards 5G," Eur. Telecommun. Standards Inst., Sophia Antipolis CEDEX, France, Tech. Rep. 979-10-92620-08-5, 2015.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[3] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, Oct. 2014.

[4] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet Things J.*, vol. 3, no. 6, pp. 854–864, Dec. 2016.

[5] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Proc. HotWeb*, Nov. 2015, pp. 73–78.

[6] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Riviere, "On using micro-clouds to deliver the fog," *IEEE Internet Comput.*, vol. 21, no. 2, pp. 8–15, Mar. 2017.

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

[8] 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. IEEE INFOCOM*, Apr. 2019, pp. 820–828.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[10] B. Yang and H. Garcia-Molina, "Comparing hybrid peer-to-peer systems," in *Proc. 27th Intl. Conf. Very Large Data Bases*, 2001, pp. 1–4.

[11] 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. ACM SIGCOMM*, 2001, pp. 149–160.

[12] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 4th Quart., 2017.

[13] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart., 2017.

[14] A. S. Gomes *et al.*, "Edge caching with mobility prediction in virtualized LTE mobile networks," *Future Gener. Comput. Syst.*, vol. 70, pp. 148–162, May 2017.

[15] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "FemtoCaching: Wireless content delivery through distributed caching helpers," *IEEE Trans. Inf. Theory*, vol. 59, no. 12, pp. 8402–8413, Dec. 2013.

[16] X. Wang, M. Chen, T. Taleb, A. Ksentini, and V. C. M. Leung, "Cache in the air: Exploiting content caching and delivery techniques for 5G systems," *IEEE Commun. Mag.*, vol. 52, no. 2, pp. 131–139, Feb. 2014.

[17] J. Bo and J. Zhao, "Index-based search scheme in peer-to-peer networks," in *Computer Science for Environmental Engineering and EcoInformatics*. Kunming, China: Springer, 2011, pp. 102–106.

[18] C. Dannewitz, M. D'Ambrosio, and V. Vercellone, "Hierarchical DHT-based name resolution for information-centric networks," *Comput. Commun.*, vol. 36, no. 7, pp. 736–749, Apr. 2013.

[19] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 3, pp. 1617–1634, 3rd Quart., 2014.

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

[21] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3th ACM SIGCOMM HotSDN*, Aug. 2014, pp. 1–6.

[22] J. Xie, D. Guo, C. Qian, L. Liu, B. Ren, and H. Chen, "Validation of distributed SDN control plane under uncertain failures," *IEEE/ACM Trans. Netw.*, vol. 27, no. 3, pp. 1234–1247, Jun. 2019.

[23] S. S. Lam and C. Qian, "Geographic routing in d-dimensional spaces with guaranteed delivery and low stretch," *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 663–677, 2013.

[24] P. Bose and P. Morin, "Online routing in triangulations," *SIAM J. Comput.*, vol. 33, no. 4, pp. 937–951, Jan. 2004.

[25] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 136–141, Feb. 2013.

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

[27] I. Borg and P. J. Groenen, *Modern Multidimensional Scaling: Theory Application*. New York, NY, USA: Springer, 2005.

[28] F. Wickelmaier, "An introduction to MDS," Sound Qual. Res. Unit, Aalborg Univ., Aalborg, Denmark, Tech. Rep. R00-6003, 2003.

[29] A. Biryukov, M. Lamberger, F. Mendel, and I. Nikolić, "Second-order differential collisions for reduced SHA-256," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2011, pp. 270–287.

[30] C. Qian and S. S. Lam, "Greedy routing by network distance embedding," *IEEE/ACM Trans. Netw.*, vol. 24, no. 4, pp. 2100–2113, Aug. 2016.

[31] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, nos. 1–6, pp. 381–413, Jun. 1992.

[32] J. A. De Loera, J. Rambau, and F. Santos, *Triangulations Structures for Algorithms and Applications*. Berlin, Germany: Springer, 2010.

[33] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[34] (Feb. 2019). *$P4_{16}$ Language Specification*. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[35] (Feb. 2019). *Java Class Hashmap<K, V>*. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

[36] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in *Proc. 9th Int. Symp.*, Washington, DC, USA, Aug. 2001, pp. 346–353.

[37] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "SCL: Simplifying distributed SDN control planes," in *Proc. NSDI*, 2017, pp. 329–345.

[38] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," *Hot-ICE*, vol. 12, pp. 1–6, Apr. 2012.

[39] T. X. Tran, P. Pandey, A. Hajisami, and D. Pompili, "Collaborative multi-bitrate video caching and processing in mobile-edge computing networks," in *Proc. 13th Annu. Conf. Wireless Demand Netw. Syst. Services (WONS)*, Feb. 2017, pp. 165–172.

[40] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A decentralized peer-to-peer web cache," in *Proc. 21st Annu. Symp. Princ. Distrib. Comput. (PODC)*, 2002, pp. 213–222.

[41] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proc. 25th Anniversary Int. Conf. Supercomput. Anniversary*, 2014, pp. 84–95.

[42] X. Song, Y. Huang, Q. Zhou, F. Ye, Y. Yang, and X. Li, "Content centric peer data sharing in pervasive edge computing environments," in *Proc. 37th IEEE ICDCS*, Jun. 2017, pp. 287–297.

[43] Y. Huang, X. Song, F. Ye, Y. Yang, and X. Li, "Fair caching algorithms for peer data sharing in pervasive edge computing environments," in *Proc. 37th IEEE ICDCS*, Jun. 2017, pp. 605–614.

[44] Q. Du, V. Faber, and M. Gunzburger, "Centroidal Voronoi tessellations: Applications and algorithms," *SIAM Rev.*, vol. 41, no. 4, pp. 637–676, Apr. 1999.

Junjie Xie received the B.E. degree in computer science and technology from Beijing Institute of Technology, Beijing, China, in 2013, and the M.E. and Ph.D. degrees in management science and engineering from the National University of Defense Technology, Changsha, China, in 2015 and 2020, respectively. He is currently an Engineer with the Institute of Systems Engineering, AMS, PLA, Beijing. His research interests include distributed systems, software-defined networking, and mobile edge computing.

Chen Qian (Senior Member, IEEE) received the B.Sc. degree in computer science from Nanjing University in 2006, the M.Phil. degree in computer science from The Hong Kong University of Science and Technology in 2008, and the Ph.D. degree in computer science from The University of Texas at Austin in 2013. He is currently an Assistant Professor with the Department of Computer Science and Engineering, University of California at Santa Cruz. He has published more than 60 research papers in highly competitive conferences and journals. His research interests include computer networking, network security, and the Internet of Things. He is a member of ACM.

Deke Guo (Senior Member, IEEE) received the B.S. degree in industry engineering from Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2008. He is currently a Professor with the College of Information System and Management, National University of Defense Technology, and a Professor with the School of Computer Science and Technology, Tianjin University. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of ACM.

Minmei Wang (Graduate Student Member, IEEE) received the B.E. degree from Nanjing University of Posts and Telecommunications in 2014 and the M.Sc. degree from Nanjing University in 2017. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of California at Santa Cruz. Her research interests include the Internet of Things and network security.

Ge Wang received the Ph.D. degree from Xi'an Jiaotong University in 2019. She was a Visiting Student with the University of California at Santa Cruz from 2017 to 2019. She is currently an Assistant Professor with Xi'an Jiaotong University. Her research interests include wireless sensor networks, RFID, and mobile computing.

Honghui Chen received the M.S. degree in operational research and the Ph.D. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 1994 and 2007, respectively. He is currently a Professor of information system and management, National University of Defense Technology. His research interests include information systems, cloud computing, and information retrieval.