# Efficient Indexing Mechanism for Unstructured Data Sharing Systems in Edge Computing

Junjie Xie*†, Chen Qian†, Deke Guo*‡, Minmei Wang†, Shouqian Shi†, 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
‡Tianjin Key Laboratory of Advanced Networking, School of Computer Science and Technology
Tianjin University, Tianjin 300350, P. R. China.
Deke Guo is the corresponding author: dekeguo@nudt.edu.cn

***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, these edge servers cache data to provide services for edge users. The data sharing among 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 the switches and the data indexes are associated with the 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 the 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 the data index compared to using Chord, a well-known DHT solution.**

## 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. In edge computing, edge servers (also called as 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]. 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 Cloud for the edge user. However, retrieving data from the Cloud would incur a large amount of backhaul traffic and a 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 [3]. 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 location of a data 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 [4] 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 [5] 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 [6], 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 two edge nodes.

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 effective because any query request from an edge server can be responded 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 index copies are essential in the edge network. In this case, the key challenge is how to quickly retrieve the data index from the nearest indexing edge server. To enable these advantages, our COIN mechanism embeds the path length between switches into the distance between points in the virtual space. After that, the data requester can instantly retrieve the data index from the nearest indexing edge servers by comparing their distances in the virtual space.

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 the data index compared to using the well-known DHT indexing mechanism [6].

The rest of this paper is organized as follows. In section II, we introduce the motivation and the design overview of this paper. We detail the COIN mechanism in Section III. In Section IV, 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 V and Section VI, 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][7]. In edge computing, each edge node caches some data to provide services for those edge users located in a given area. When a user 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, as shown in Fig. 1. 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 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 [7]. 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. Another one is that the edge servers can deliver those data generated 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

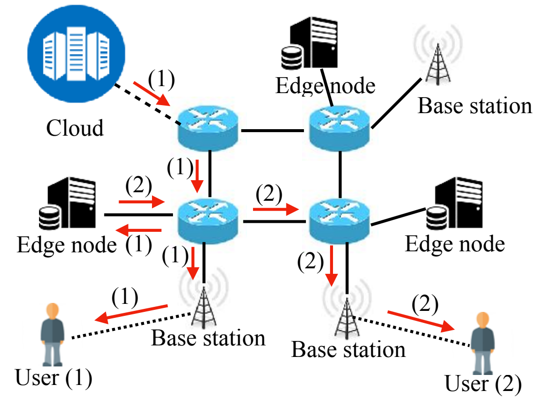



Fig. 1. Retrieving data in the edge computing environment.

speeds are growing exponentially, following Moores Law [7]. 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 [3]. 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 Cloud. Meanwhile, for the robustness of the indexing system, multiple index copies could be maintained for each shared data. It is also essential to optimize the indexing system for multiple index copies.

### B. Design Overview

To achieve the data sharing among edge nodes, the key challenges are where to place the data index, how to search the data index 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**. 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.

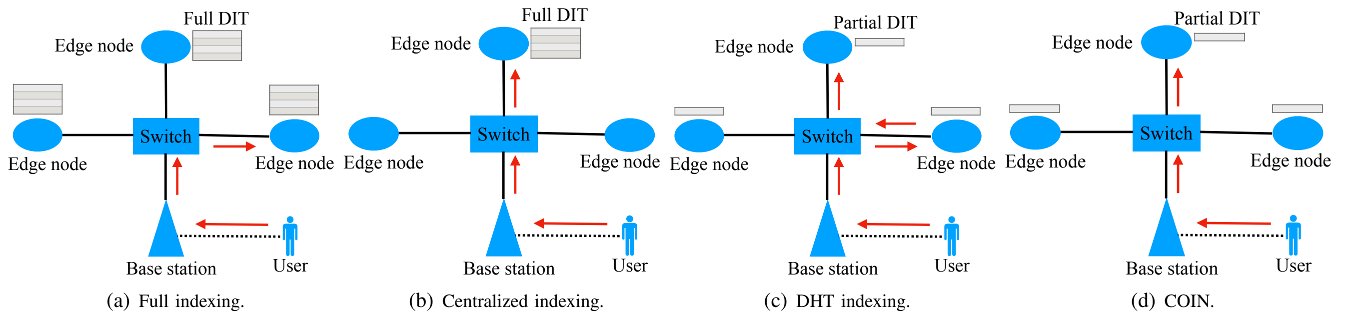



Fig. 2. Packet forwarding under different data indexing mechanisms.

Note that each edge node determines one of the edge servers as the indexing edge server.

4) An *indirect edge server* refers to an intermediate edge server that forwards any query request of data index, not including the ingress edge server and the indexing edge server.

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 [6][8][9]. The DHT indexing is a distributed indexing mechanism, and each indexing edge server just stores partial DIT. However, the DHT indexing mechanism 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 where $n$ is the number of edge nodes in the edge network [6]. 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). 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

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 |

(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) [10][11] where the control plane can collect the network topology and state including switch, port, link, and host information [12]. 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, which generate the forwading table entries for switches. The switches in the switch plane only forward packets according to the installed entries derived from the controller. Note that the performance and scalability of the control plane is the key to the COIN mechanism. At current, there have been much research on improving the performance and scalability of the control plane [13][14].

To achieve the COIN mechanism, the control plane maintains a virtual 2D space where each switch is associated with a coordinate. The coordinates of switches are calculated in Section III-A1. Then, the control plane constructs a Delaunay Triangulation (DT) graph [15][16] in Section III-B1 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

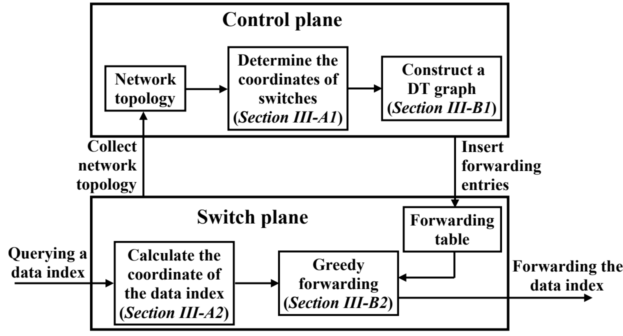



Fig. 3. The framework of the COIN mechanism over a software-defined edge network.

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-A2. 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-B2. Finally, the switch forwards the data index to the only indexing edge server among all directly connected edge servers.

## III. Coordinate-based indexing

In this section, we introduce the details of COIN include the main functions shown in Fig. 3.

### 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 [12][11]. 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 [17]. 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.

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 $\Lambda$ is diagonal.

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

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

where $P$ is an $n{\times}m$ orthogonal matrix, (i.e., $P'P{=}I$), $\Phi$ is an $m{\times}m$ diagonal matrix, and $Q$ is an $m{\times}m$ orthogonal matrix, (i.e., $Q'Q{=}I$).

**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 $\lambda_1, \lambda_2, ..., \lambda_m$ and corresponding eigenvectors $e_1, e_2, ..., e_m$ 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.

Assume that we know the decompositions 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 $\phi_i^2$ 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 $B{=}(Q\Lambda^{1/2})(Q\Lambda^{1/2})'{=}UU'$, where $\Lambda^{1/2}$ is a diagonal matrix with diagonal elements $\lambda_i^{1/2}$. Thus, $U{=}Q\Lambda^{1/2}$ gives coordinates that reconstruct $B$. The coordinates in $U$ may differ from those of $X$ in Equation (1). This simply means that they are expressed relative to two different coordinate systems, which, however, can be rotated into each other [17].

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, which can preserve the network distances between switches 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 [12]. 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 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.

*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* [18], 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.

### *B. Publishing a data index*

Under the COIN mechanism, each switch greedily forwards a data index to its neighbor, whose coordinate is closest to the coordinate of the data index. Recall that greedy routing on a DT graph provides the property of guaranteed delivery [15][19], which is based on a rigorous theoretical foundation. Therefore, to achieve the guaranteed delivery, the control plane first constructs a DT graph, which connects all switches' coordinates in the virtual space.

*1) DT construction:* Given a set of switches and their coordinates in a set of points $P$, we adopt the randomized incremental algorithm [20] to construct the DT $DT(P)$ in the 2D virtual space. After constructing a DT, note that a DT neighbor of a switch may not be its physical neighbor. 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. Note that a DT neighbor could also be a physical neighbor. But, here, those DT neighbors are the neighbors except for the physical neighbors. 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 the 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 $t$.

*2) Forwarding the data index:* 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 [21][22]. Multiple match-action tables are declared in the P4 switch where the standard table includes two properties: key and action [22]. 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-A1. 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 the data index in the virtual space.

When a data item is cached by an storing edge server. The edge server will publish the 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 to compute estimated distances. 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 every 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 $R_{v^*}{<}Dis(u, d)$, $u$ 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. 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.

**Forwarding to a DT neighbor.** 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. 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 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-B1. 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.

### *C. Querying a data index*

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 the 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-B2 to determine the indexing edge server, which will respond to the querying request. Then, the indexing edge

server returns the 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.

### *D. The optimization design for 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 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 $\alpha$ index copies, the indexing edge server that stores the $\alpha_{th}$ index copy is uniquely determined by the hash value $H(d{+}\alpha{-}1)$.

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-A1. After that, the switch can forward the querying request of a data index to the nearest index copy based on the coordinates of the switch and the index copies. Therefore, under the COIN mechanism, the ingress edge server can quickly select the index copy that achieves the shortest path length to retrieve the data index.

## IV. Performance Evaluation

### *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. 4. We implement the centralized indexing (*C-index*), the DHT indexing (*D-index*) [6] 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 [21], and the function in the control plane is written in Java. The P4 compiler generates Thrift APIs for the controller to insert 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 [22] 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.

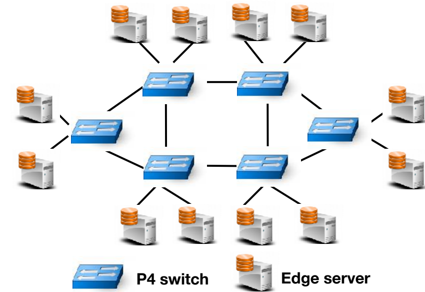
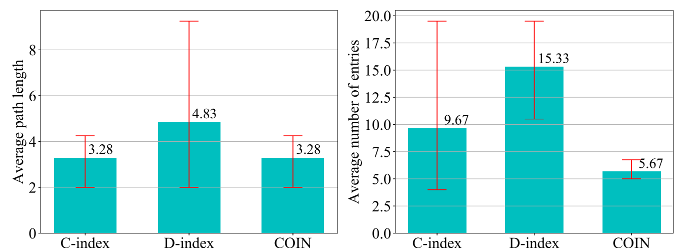



Fig. 4. The network topology consists of 6 P4 switches and 12 edge servers.

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. 5(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. Note that 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 the data indexing under different indexing mechanisms where the C-index and D-index mechanisms forward the packets by matching the source and destination addresses. Fig.




(a) The path lengths of retrieving indexes. (b) The number of forwarding entries.

Fig. 5. The path lengths and the numbers of forwarding entries under different indexing mechanisms in a small-scale testbed.

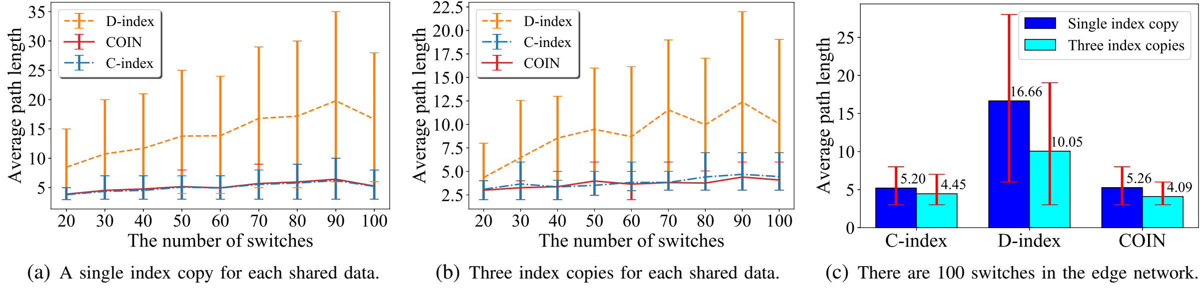



Fig. 6. The average path lengths for retrieving data indexes under different indexing mechanisms.

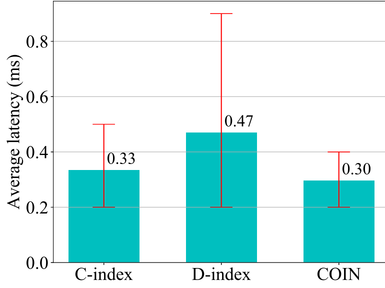



Fig. 7. The latency of retrieving indexes under different indexing mechanisms in a small-scale testbed.

5(b) shows the average number of forwarding table entries per switch under different indexing mechanisms. As shown in Fig. 5(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 the 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.

We evaluate the impact of multiple index copies on the performance of the COIN mechanism. We have stored $10,000$ data items, 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 under different indexing mechanisms. Fig. 7 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. 7. It is mainly because that the scale of our testbed is small. Furthermore, we will conduct large-scale simulations in the next section.

### *B. Setting of large-scale simulations*

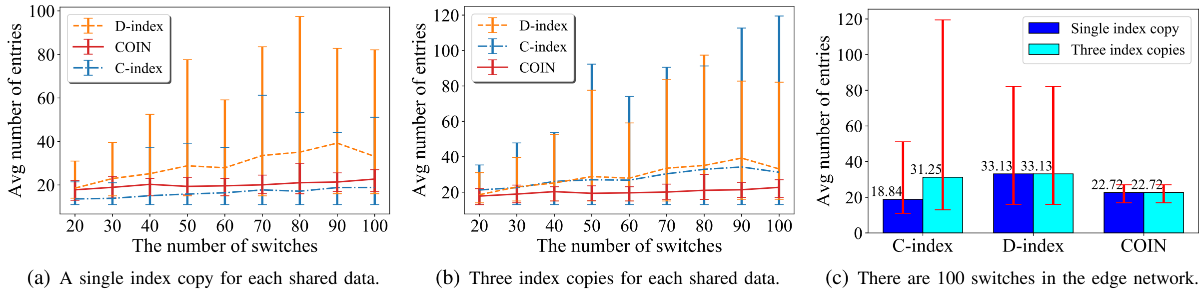
In simulations, we use BRITE [23] with the Waxman model to generate synthetic topologies at the switch level where each switch connects to 10 edge servers. We vary the number of switches from 20 to 100. Note that our COIN mechanism can be scaled to larger networks. At current, there have been much research on improving the performance and scalability of the control plane [12][11][13]. Meanwhile, it is worth noting that the advantage of the COIN mechanism will be more obvious when the network size increases. We compare the centralized indexing (*C-index*), the DHT indexing (*D-index*) [6] with our COIN mechanism. We adopt two performance metrics to evaluate different indexing mechanisms including the path lengths and the number of forwarding table entries for retrieving data indexes. In the following figures, each error bar is constructed using a $95\%$ confidence interval of the mean. Meanwhile, we evaluate the impact of multiple index copies on the path lengths of retrieving data indexes.

### *C. The path lengths for retrieving data indexes*

In this section, we evaluate the path lengths for retrieving 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. 6(a) shows that the average path length of retrieving data indexes are almost the same for COIN and C-index mechanisms. Note that 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. 6(a). The average path length under the D-index mechanism has an obvious increase as the increase in the number of switches in Fig. 6(a). However, the increase is slow for COIN and C-index mechanisms when the number of switches changes.

Note that the results are achieved in Fig. 6(a) where 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. 6(b), which shows almost the same trend as Fig. 6(a). That is, the average path length for retrieving data indexes under COIN mechanism is close to the average path length achieved by C-index mechanism and is obviously shorter than the average path length under D-index mechanism. It is worth noting that C-index mechanism is a centralized indexing mechanism and suffers from the performance drawbacks in the fault tolerance and the scalability.




(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. 8. The number of forwarding table entries under different indexing mechanisms.

Fig. 6(c) shows that more index copies result in shorter path lengths for retrieving data indexes under the three indexing mechanisms. Meanwhile, we can see that the impact of index copies on the path length under D-index mechanism is more obvious than the other two mechanisms. However, D-index mechanism still needs a longer path to retrieve a data index than COIN and C-index mechanisms. As shown in Fig. 6(c), our COIN mechanism employs the shortest paths to retrieve data indexes 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 D-index mechanism when there are only one index copy and three index copies, respectively.

### *D. Forwarding entries for retrieving data indexes*

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

Fig. 8(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. 8(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. 8(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. 8(a). It is because that the C-index mechanism employs the shortest path routing where 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 of Fig. 8(a) is achieved when there is only one index copy for each shared data. Furthermore, Fig. 8(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. Note that the average number of forwarding entries decreases when

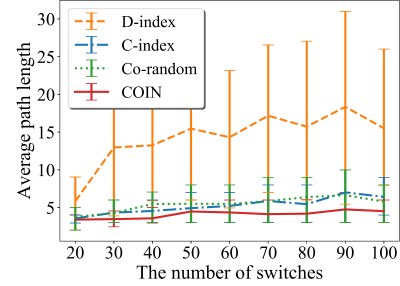



Fig. 9. The impact of multiple index copies on the average path lengths.

the number of switches varies from 90 to 100. The reason is that the network topologies are generated independently under different network sizes. 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. 8(c). However, more index copies have no impact on the number of forwarding table entries for D-index and COIN 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 index copies*

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. 9, the path lengths of retrieving data indexes under Co-random and C-index mechanisms are very close, and they are obviously shorter than the path lengths under D-index mechanism. Meanwhile, we can see that our COIN mechanism employs the shortest paths to retrieve data indexes than the other three indexing mechanisms under any network size from Fig. 9. 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.

## V. Related Work

**Data sharing in other computing environments.** Although there are some researches about the data sharing in P2P

networks and Web Proxies, we present why those researches 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” [4], where each proxy keeps a summary of the cache directory, and checks these summaries for potential hits before sending any queries. Iyer et al. presented a decentralized, peer-to-peer web cache called Squirrel [24]. Bo et al. proposed reference architecture for P2P systems [8] that focuses on the data indexing technology required to support resource locating. A P2P index can be local [25], centralized [5] or distributed [6]. 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.

**Peer data sharing among edge devices.** At current, there are some researches 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 [26]. Furthermore, Huang et al. considered caching fairness for peer data sharing among edge devices [27]. However, there is still a lack of researches about the data sharing among edge servers, which can provide more opportunities for peer data sharing among edge devices.

## VI. 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.

## Acknowledgment

This work is partially supported by the National Natural Science Foundation for Outstanding Excellent young scholars of China under Grant No.61422214, National Natural Science Foundation of China under Grant Nos.61772544 and U1536202, the Hunan Provincial Natural Science Fund for Distinguished Young Scholars under Grant No.2016JJ1002, and the Guangxi Cooperative Innovation Center of cloud computing and Big Data under Grant Nos.YD16507 and YD17X11.

## References

[1] 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.

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

[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] 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, 2000.

[5] B. Yang and H. Garcia-Molina, “Comparing hybrid peer-to-peer systems,” in *Proc. 27th Intl. Conf. on Very Large Data Bases*, 2001.

[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. ACM SIGCOMM*, 2001, pp. 149–160.

[7] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, “A survey on mobile edge computing: The communication perspective,” *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[8] J. Bo and J. Zhao, “Index-based search scheme in peer-to-peer networks,” in *Computer Science for Environmental Engineering and EcoInformatics*. Springer, 2011, pp. 102–106.

[9] C. Dannewitz, M. DAmbrosio, and V. Vercellone, “Hierarchical dht-based name resolution for information-centric networks,” *Computer Communications*, vol. 36, no. 7, pp. 736 – 749, 2013.

[10] 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 Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

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

[12] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, and G. Parulkar, “Onos: Towards an open, distributed sdn os,” in *Proc. 3th ACM SIGCOMM HotSDN*, August 2014.

[13] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, “Scl: Simplifying distributed sdn control planes.” in *NSDI*, 2017, pp. 329–345.

[14] J. Xie, D. Guo, X. Zhu, B. Ren, and H. Chen, “Minimal fault-tolerant coverage of controllers in iaas datacenters,” *IEEE Transactions on Services Computing*, 2017.

[15] 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.

[16] P. Bose and P. Morin, “Online routing in triangulations,” *SIAM journal on computing*, vol. 33, no. 4, pp. 937–951, 2004.

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

[18] A. Biryukov, M. Lamberger, F. Mendel, and I. Nikolić, “Second-order differential collisions for reduced sha-256,” in *Advances in Cryptology – ASIACRYPT 2011*, pp. 270–287.

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

[20] L. J. Guibas, D. E. Knuth, and M. Sharir, “Randomized incremental construction of delaunay and voronoi diagrams,” *Algorithmica*, vol. 7, no. 1, pp. 381–413, 1992.

[21] 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.

[22] “$P4_{16}$ language specification,” [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf, accessed June 2018.

[23] A. Medina, A. Lakhina, I. Matta, and J. Byers, “Brite: An approach to universal topology generation,” in *Proc. 9th International Symposium in MASCOTS*, Washington, DC, USA, 2001.

[24] S. Iyer, A. Rowstron, and P. Druschel, “Squirrel: A decentralized peer-to-peer web cache,” in *Proc. 21th annual symposium on Principles of distributed computing*. ACM, 2002, pp. 213–222.

[25] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, “Search and replication in unstructured peer-to-peer networks,” in *Proc. 16th international conference on Supercomputing*. ACM, 2002, pp. 84–95.

[26] 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*, June 2017, pp. 287–297.

[27] 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*, June 2017, pp. 605–614.