Fair and Efficient Caching Algorithms and Strategies for Peer Data Sharing in Pervasive Edge Computing Environments

Yaodong Huang, Xintong Song, Fan Ye, Yuanyuan Yang, *Fellow, IEEE*, and Xiaoming Li, *Senior Member, IEEE*

Abstract—Edge devices with sensing, storage, and communication resources (e.g., smartphones, tablets, connected vehicles, and IoT nodes) are increasingly penetrating our daily lives. Many novel applications can be created through sharing data among nearby peer edge devices. In such applications, caching data at some edge devices can greatly improve data availability, retrieval robustness, and delivery latency. In this paper, we study the unique problem of caching fairness in edge computing environments. Due to the heterogeneity of peer edge devices, load balance is a critical issue that affects the fairness in caching. We propose fairness metrics to characterize this issue and formulate the caching fairness problem as an integer linear programming problem, which is shown as the summation of multiple Connected Facility Location (ConFL) problems. We provide an approximation algorithm by leveraging an existing ConFL approximation algorithm, and prove that it preserves a 6.55 approximation ratio. We further develop a distributed algorithm where devices exchange data reachability information and identify popular candidates as caching nodes. Finally, we update the fairness metric and apply it to algorithms for making continuous caching decisions over time. Our extensive evaluation results show that compared with existing caching algorithms for wireless networks, our proposed algorithms significantly improve the data caching fairness while keeping the contention induced latency comparable to the best existing algorithms.

Index Terms—Pervasive edge computing, peer data sharing, cooperative caching

1 Introduction

Our surrounding environments are increasingly penetrated with various types of *edge devices*, including mobile phones, tablets, connected vehicles, road-side cameras, and diverse Internet-of-Things (IoT). These devices possess sensing, computing, storage and communication capabilities. They produce pervasive sensing data about physical phenomena in the surrounding environment. By sharing sensing data among such peer edge devices, numerous novel applications can be created.

1.1 Motivation

Consider a large outdoor public event (e.g., music festival, or university commencement). Smartphones carried by people can capture diverse data, including human activities, their locations, and image/video clips. When shared among peer devices, such data can help people avoid food stands of long lines, discover interesting souvenirs and artifacts, or enjoy images, video clips of special, memorable moments. Another example is traffic accident photos and video clips that help to keep nearby police and drivers aware of the conditions. Police and insurance companies can determine who is responsible for the accident and accelerate the process, and drivers can choose alternative routes to avoid traffic jams. Yet another case is micro-climate sensors that can collect temperature, humidity and air pollution conditions for certain areas, especially in suburbs. Residents and schools can schedule outdoor activities based on such conditions.

Caching is a critical mechanism to enable such peer data sharing among edge devices. The movements of people thus devices and the varying availabilities of data and resources (e.g., battery, storage) constitute a highly fluid environment full of uncertainty and dynamics. By caching the data at willing and capable devices, the availability of data, the robustness, and latency in their retrieval can all be greatly improved. Recent content centric networks [1] even integrate caching as a fundamental component in their design.

1.2 Challenges and Contributions

Despite some earlier work [2], [3] on caching in wireless networks (MANET), the critical issue of *fairness* in caching has not been addressed. Such work focuses on reducing the contention thus latency in data retrieval. They can cause extremely unbalanced caching load, e.g., a few fixed devices are always chosen as caching nodes [4], [5]. Although this may not be an issue in MANET if all devices listen to one authority, it is simply infeasible in edge environments: each device may belong to a different owner, and caching

- *Y. Huang, F. Ye, and Y. Yang are with the Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794. E-mail: {yaodong.huang, fan.ye, yuanyuan.yang}@stonybook.edu.*
- *X. Song and X. Li are with the School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China. E-mail: {songxintong, lxm}@pku.edu.cn.*

Manuscript received 14 Nov. 2017; revised 30 Nov. 2018; accepted 19 Feb. 2019. Date of publication 27 Feb. 2019; date of current version 4 Mar. 2020. (Corresponding author: Yuanyuan Yang.)
Digital Object Identifier no. 10.1109/TMC.2019.2902090

decisions can only be voluntarily accepted, not forcefully mandated.

In this paper, we study how to ensure caching fairness among peer devices in pervasive edge computing environments. We formulate the problem in integer linear programming form, and show that it is the summation of multiple Connected Facility Location (ConFL) problems. We propose an approximation algorithm and prove that it preserves a 6.55 approximation ratio to the optimal solution. We also show that it achieves very good caching fairness when 75 percent of data are cached on 73.43 percent of nodes. We further develop a distributed algorithm where nearby devices exchange data availability information to make collaborative caching decisions. Finally, we update the definition on fairness metrics and apply that to the proposed distributed algorithm for making caching decisions continuously as data are generated or become obsolete in the network.

We make the following contributions in this paper.

- We consider caching fairness in data sharing among peer edge devices. On top of contention costs thus retrieval latency, the fairness costs are incorporated into an integer linear programming problem, which is shown to be the summation of multiple traditional ConFL problems.
- We design an algorithm by leveraging one of the existing ConFL approximation algorithms. We prove that the algorithm preserves the same approximation ratio as the original ConFL algorithm.
- We further design a distributed algorithm where devices exchange data retrieval costs among 2-hop neighbors to identify candidates with the smallest costs, and popular candidates volunteer to cache data.
- We modify fairness metrics and propose a caching strategy using proposed algorithms that continuously adapts caching decisions as new data are generated or old data become obsolete.
- We implement our algorithms and compare against other algorithms for distributed wireless network caching. Simulation results show that our algorithms significantly improve data caching fairness while keeping the contention induced latency comparable to the best existing algorithms with $O(N^3)$ complexity, where N is the number of nodes in the network.

The rest of the paper is organized as follows. In Section 2, we discuss some related work on mobile caching and facility location problems. In Section 3 we give the system model and the formulation of the problem. We provide algorithms in Section 4 to solve the problem. We evaluate our design and compare with other previous works in Section 5. Finally, we conclude the paper in Section 6.

2 Related Work

Caching is one of the classical mechanisms to improve data access robustness and performance. Cooperative caching, which shares and coordinates data caching decisions among nodes, has been applied in ad hoc networks. For cooperative caching for data, Yin et al. [6] propose two caching schemes and a method to obtain data in mobile networks. Hara et al. [7] propose a strategy to remove redundancy in the neighborhood, and Hamlet [8] minimizes access costs by leveraging the content diversity of different data in the neighborhood.

There exists some work that focuses on improving data access rates by caching in wireless networks. One basic idea is to place cache based on content popularity such that the cached content can frequently be used. WAVE [9] decides popularity based on the recommendation from upstream node request counts. Li et al. [10] dynamically place caching replicas on the en-route path in named data networks (NDN) [11], and MPC [12] caches only popular content adapted in Content-Centric Networks (CCN) [13]. Another approach is to find caching locations that minimize data access latency. Nuggehalli et al. [14] use the hop-count as the delay model to find the best places to cache data. Later, Fan et al. [3] propose a contention-aware caching algorithm, which is more accurate than the hop-count based algorithm since the packet contention is the fundamental cause of latency in MANET. Similarly, Sung et al. use contention as a key factor in determining the delay. They introduce a contention based solution on flat wireless networks [15], and extend to two-tier wireless content delivery networks [5]. Caching can also help deal with mobility in edge computing scenarios. Proactively caching data near where nodes need data can improve the data accessing rate despite the high mobility of nodes [16], [17].

The problem of determining caching locations is closely related to the classical Facility Location (FL) problem. Most caching studies map their problems into different FL problems or modify FL problems to solve caching placement. Usually, they use either Uncapacitated Facility Location (UFL) problem [18] or rent-or-buy problem [19]. The more general case for these two problems is the Connected Facility Location problem [20]. Among them, UFL does not consider the content dissemination costs, while rent-or-buy problem does not consider the facility building costs in the ConFL problem.

In this paper, we design our approximation algorithms by leveraging the ConFL problem. Since the ConFL is an NP-hard problem, previous work proposes many approximation algorithms. Jung et al. [21] achieve the best deterministic constant approximation ratio with 6.55-approximation based on the primal-dual approach. The best approximation algorithm is a 4.32-approximation randomized algorithm [22], in which the authors argue that through a derandomization process, it could reach a factor of 4.32-approximation ratio. However, the derandomization process is not polynomial, thus limiting its ability to solve the ConFL problem. There are also heuristic [23] and greedy [24] solutions for the ConFL problem. Though such algorithms may not have solid approximation bounds, they may still achieve good performance in practice. We focus on the algorithms with bounded approximation ratios (i.e., deterministic ones) and leverage them in our algorithms.

3 System Model and Problem Formulation

In this section, we introduce our system model and discuss how we quantify fairness and contention of the network. Then we provide an integer linear programming (ILP) formulation for the problem and explain its relation to the ConFL problem.

3.1 System Model

Let graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a connected undirected graph representing the network topology for a multi-hop wireless network, where vertex set ($\mathcal{V}$) represents the nodes in the network, and edge set ($\mathcal{E}$) represents the connection links between nodes. In many pervasive edge environments, nodes (e.g., customers in cafés) exhibit low to moderate mobility, and the amount of exchanged data is limited (e.g., a few MBs). Pedestrians carrying smartphones or IoT devices will not move much during such short data transmission times (e.g., a few seconds). Thus we assume that the network topology remains relatively stable while caching decisions are being made. For high mobility scenarios such as vehicles that may change the topology significantly, we have developed other algorithms and will publish them separately.

We assume that some data are interested by all nodes. The goal is to find an optimal way to distribute these data so as to achieve fairness while ensuring low data access latency. The whole process consists of two phases. First, we divide these data into multiple equal size data chunks $\mathcal{N}$ and proactively disseminate them, letting certain node $i \in \mathcal{V}$ cache certain chunk $n \in \mathcal{N}$. We call this process the dissemination phase. Then, the node which desires data chunk n will acquire the chunk from a nearby node which has already cached it. We call this process accessing phase. Thus, each node needs to acquire a copy of each data chunk. We do not assume any specific coding technique in fetching or transmission and each chunk is stored and acquired as is.

3.2 Fairness Degree Cost

Data caching fairness is a critical issue in edge computing environments where many nodes are owned by different users. Both storage and battery are important resources, and the user can decide how many resources to contribute to caching. Any fixed selection (e.g., the same group of nodes) for caching will consume excessive resources on chosen nodes. Thus their users may stop participating.

The key to achieving fair caching is to cache less data on nodes with fewer resources. We quantify the caching fairness of a node by defining a *Fairness Degree Cost* based on current node resource consumption conditions. For a node, the higher *Fairness Degree Cost* is, the fewer resources are available, and the less likely it will cache data. Intuitively, the ratio between used and remaining storage for caching can represent the usage of storage of a node. The *Fairness Degree Cost* for node i is defined as

$$f_i = \frac{S(i)}{S_{\text{tol}}(i) - S(i)}, \tag{1}$$

where $S_{\text{tol}}(i)$ is the total caching storage of node i, and $S(i)$ is the storage used. Thus, $S_{\text{tol}}(i) - S(i)$ is the storage still available. Intuitively, it represents a "penalty:" the fewer resources a node has, the more "cost" the network must pay to cache data on it. A "cost" of 0 indicates that the node has not cached anything, and ∞ means that the storage of the node is full and no further caching is possible. To reduce the cost, nodes with little storage thus high costs are unlikely to be chosen to cache. Since all chunks have the same size, we define $S_{\text{tol}}(i)$ as the total number of chunks the node can cache, and $S(i)$ as the number of chunks the node has cached.

Fairness also unifies the solution to cache multiple data items. Previous wireless caching work, e.g., [5], [14], only considers caching one piece of data. Although some of them argue that their solutions can be extended to multiple data items, the exact process is unclear. According to our understanding, these solutions may involve changes in the network topology, which may change the underlying problem into a different instance. When fairness is considered, multiple data chunks can be cached based on current available resources on nodes.

Note that the *Fairness Degree Cost* can be used to value limited resources for edge nodes such as batteries and bandwidth. The cost formulation can be similar to that of storage, which is the ratio of used and remaining resources. The total *Fairness Degree Cost* can be defined as the weighted summation of all corresponding costs. The weighted summation ensures that the probability of choosing a node decreases if any resource of this node is limited at that time. For simplicity, we only consider the *Fairness Degree Cost* for storage in this paper.

3.3 Wireless Contention Cost

Wireless contention is one of the most important factors affecting per hop latency in multi-hop wireless networks. Contentions cause packet loss and large back-off time, both increasing the data access latency. Minimizing contention is a key to reduce the latency. The contention delay model that describes the delay due to wireless contention has been considered in [3], [5], [15]. We call it *Contention-induced Delay Cost*, or simply *Contention Cost*. We use the *Node Contention Cost* w_k to denote the *Contention Cost* on a specific node $k \in \mathcal{V}$. It is affected by the number of its contending neighbors, and the amount of transmissions. In this scenario, the *Node Contention Cost* w_k is defined as the number of data packet transmissions through node k, both receiving and sending. The accurate mathematical representation of this contention makes the problem very difficult to solve, and makes it almost impossible for distributed algorithms to obtain accurate contention information. Thus, inspired by [5], we propose an estimation solution adopting a similar approach. For a node k, all neighbors will send requests to it, and the node will return all data chunks it receives to direct neighbors. Thus, w_k can be regarded as its degree, which equals the number of data chunks the node will send to its neighbors (i.e., one chunk per neighbor).

The *Path Contention Cost* between two nodes i and j is based on the *Node Contention Cost* alongside the path. We formulate it as

$$c_{ij} = \sum_{k \in \text{PATH}(i,j)} w_k[1 + S(k)], \tag{2}$$

where all *Node Contention Costs* are summed along the shortest path which the data packet will go through. Note that previously cached data chunks also affect the contention. Each of these data chunks (cached or new) increases the contention by the value of the node degree since each chunk should be transmitted to all neighbors.

The *Contention Cost* defined above focuses on the delay in the network caused by the contention in sending and

receiving messages. Yang et al. give a delay estimation of contention in [25]. Such *Contention Cost* is roughly a linear transformation of the *Contention Delay* model for evaluating the delay. Basically, it considers the processing (DCF Inter-Frame Space in 802.11) delay, back-off delay, transmission delay and collision delay for a hop. It is represented as

$$d(k,c) = DIFS + m_k\epsilon + w_k T_d + m_k^c T_c,$$

where for node k, DIFS is DCF Inter-Frame Space in 802.11, m_k is the number of back-off slots, ϵ is the length of back-off slot, w_k is the number of chunks transmitted in neighboring nodes, T_d is the transmission duration of a data chunk, m_k^c is the number of collisions and T_c is the duration of a collision. Since the back-off slot time and collision duration are relatively small, we can assume that $T_d \approx T_c \approx \epsilon$. $m_k = S(k)$ is the number of stored data chunks of nodes. $m_k^c = (w_k - 1)S(k)$ is the maximum number of collisions when all neighbors except one send their stored chunks.[1] Thus, $d(k,c) \approx DIFS + T_d(w_k + S(k) + (w_k - 1)(S(k))) \approx DIFS + T_d(w_k + w_k S(k))$, which means that the one-hop *Contention Delay* is roughly a linear transformation of the one-hop *Contention Cost*. We will use the contention cost to represent access latency in the following. Note that c_{ij} is used for *Contention Cost* for both accessing and dissemination phases where the network topology remains unchanged.

3.4 Problem Formulation

We now formulate the caching problem in pervasive edge computing environments. The basic idea is to add the *Fairness Degree Cost* and *Contention Cost* in a weighted form, and associate a decision variable to different chunks. Since the *Fairness Degree Cost* and the *Contention Cost* have different meanings, after some tests, we find that it is better to set them the same weight in the following integer linear programming formulation:

$$\min \sum_i \sum_n f_i y_{in} + \sum_i \sum_j \sum_n c_{ij} x_{ijn} + \sum_{e\in\mathcal{E}} \sum_n c_e z_{en} \tag{3}$$

$$\text{s.t.} \quad \sum_i \sum_j x_{ijn} = 1, (\forall n \in \mathcal{N}) \tag{4}$$

$$y_{in} - x_{ijn} \geq 0, (\forall i, j \in \mathcal{V}, \forall n \in \mathcal{N}) \tag{5}$$

$$\sum_{i\in Y_n} x_{ijn} \leq \sum_{e\in\delta(Y_n)} z_{en}, (\forall j \in \mathcal{V}, \forall n \in \mathcal{N}, \forall Y \subseteq \mathcal{E}) \tag{6}$$

$$x_{ijn}, y_{in}, z_{en} \in \{0,1\}.(\forall i, j \in \mathcal{V}, \forall e \in \mathcal{E}, \forall n \in \mathcal{N}). \tag{7}$$

In the above linear programming problem, $c_e = c_{ij}$ if node i and node j are the two end points of edge e, x_{ijn}, y_{in} and z_{en} are assignment variables, and x_{ijn} is the accessing variable. If $x_{ijn} = 1$, node j will access data chunk n from node i. y_{in} is the caching indicator variable. $y_{in} = 1$ means that data chunk n will be stored in node i. z_{en} is the dissemination variable. $z_{en} = 1$ means that the data chunk n will be disseminated through edge e in the dissemination phase.

1. Only one node sending causes no collision.

There are three terms in Equation (3), the objective function: the *Fairness Degree Cost* of the entire network, the *Contention Cost* for the accessing phase and the dissemination phase of the entire network. Here, f_i is defined in Equation (1) and c_{ij} is defined in Equation (2). If the storage of a node is used up, the node will not cache any more data chunks. Constraint Equation (4) ensures that every node j should receive a specific data chunk n from node i. It is clearly not optimal if a node receives the same data chunk multiple times. Constraint Equation (5) ensures that if node j receives a data chunk n from node i, node i must store that chunk. Constraint Equation (6) is the connectivity constraint. Here, Y_n is any subset of nodes, and $\delta(Y_n)$ represents all edges that connect to Y_n. This is to ensure that for any subset of chosen nodes to cache data chunk n, they are connected in a Steiner tree [26]. These nodes need to be connected to disseminate the data chunks along the Steiner tree. $z_{en} = 1$ means that edge e is chosen in the Steiner tree so that data chunk n will be disseminated through this connection link.

Our problem is an extended case from the Connected Facility Location problem. It can be transformed to the summation of multiple ConFL problems. f_i is equivalent to the construction cost for a facility in ConFL problem, which in this case represents the cost that the network is willing to pay to select nodes caching a data chunk. c_{ij} and c_e can be regarded as modified distance costs, adding the factor of contention of the network. y_i, x_{ij} and z_e have similar meanings. They represent node i as facilities or caching nodes in the respective problem and node j wants to access it through edge e. As shown in [20], the original ConFL problem is NP-hard. The summation of all different chunks is a polynomial time mapping, which maps our problem to the ConFL problem. Thus, our problem is also NP-hard.

It is very difficult to solve the ConFL problem directly. Fortunately, there are many existing approximation algorithms, among which the best algorithm has a 4.23 approximation ratio. To take advantage of approximation algorithms for ConFL, we make another transformation of our problem to

$$\sum_n \left(\min \sum_i f_i y_{in} + \sum_i \sum_j c_{ij} x_{ijn} + M \sum_{e\in\mathcal{E}} c_e z_{en} \right). \tag{8}$$

We transform from one optimization goal of achieving minimization from variables with data chunks into the summation of multiple minimization problems for each chunk individually. This way we can apply the approximation algorithm to problem Equation (8) by using it multiple times for different chunks. Although Equation (8) is different from Equation (3), we will later show that under certain conditions, we can use this iterative solution to Equation (8) to solve Equation (3), and the approximation ratio of the original approximation algorithm is preserved.

4 Caching Algorithms

In this section, we first propose an approximation algorithm for the caching problem by leveraging an existing ConFL approximation algorithm. Then we develop a distributed algorithm where devices exchange data reachability information and identify popular candidates as caching nodes.

TABLE 1
Notations Used in Approximation Algorithm

$L(n)$	Caching node set for data chunk n
P_v	Producer of data chunk v
F	FROZEN node set
T	TIGHT node pair set
A	ADMIN node set
B	INACTIVE node pair set
C_d	Direct connection pair set
C_i	Indirect connection pair set
U_x	Unit increase value of var x
R	Regional village

4.1 Approximation Algorithm

In our proposed approximation algorithm, we obtain input information from nodes in the network and leverage a ConFL approximation algorithm to solve the problem. We adopt the algorithm for the ConFL problem in [21], which has an approximation ratio of 6.55. This algorithm approximates the largest possible value of the dual problem. Basically, nodes with sufficient resources, which lie in the path between multiple nodes and an existing caching node or producer, will be selected as caching nodes. The details of our proposed algorithm are described in Algorithm 1 and the notations used are given in Table 1.

The algorithm can be explained as follows. For each chunk, we conduct one iteration (line 2). After some initialization steps, we update the *Fairness Degree Cost* for all nodes in the network (lines 5-7) and then estimate the *Contention Cost* of all links based on Equation (2) (lines 8-16). Lines 17-46 are phase 1 of the approximation algorithm. It iterates until every node finds out at which place they can obtain the data chunk, (aka the state FROZEN in the original approximation algorithm [21]). First, the algorithm increases the *price* it is willing to pay for establishing a connection to a caching node (lines 18-20). Note that here the increasing step units U_α, U_β and U_γ can be different since the increasing steps of the three parameters have different meanings on costs in the dual problem.

As we mentioned previously, choosing the parameter wisely can make the solution better. If this *price* is larger than the cost for accessing one existing caching chunk on a node, it can establish a connection to that node (lines 21-26), which is the first and second conditions in phase 1 of the original approximation algorithm in [21]. If not, it goes to the third condition. Lines 29-30 and 31-32 are the conditions in 3(a) and 3(b) respectively in [21]. Lines 33-42 deal with condition 3(c) in [21] where all direct connections, inactive node set and ADMIN set are created. This ends phase 1. Phase 2 is to connect the locations of corresponding nodes together. Many existing algorithms can be used to address the problem, from which we choose [26]. We will not discuss it in detail here. Finally, for chunk n, the ADMIN set A are the nodes that will cache the chunk. We save this set into $L(n)$ and start the next round until we find all caching nodes for all chunks.

It should be mentioned that although the best approximation ratio for ConFL problems so far is 4.23 as proposed in [22], the algorithm can only obtain a deterministic approximation ratio with a derandomization process which needs to solve an exponential size linear programming relaxation. Thus, this algorithm is practically inefficient and very hard to implement. On the other hand, the 6.55-approximation algorithm we choose has the lowest deterministic approximation ratio of polynomial time to ConFL problem for now.

Algorithm 1. Approximation Algorithm

Input: $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E}), P_v$
Output: $L(n)$
1: $L(n) \leftarrow \emptyset$
2: **for all** Data chunk n **do**
3: $F \leftarrow \emptyset, T \leftarrow \emptyset, C_d \leftarrow \emptyset, C_i \leftarrow \emptyset, A \leftarrow \emptyset, B \leftarrow \emptyset$
4: $\alpha_j \leftarrow 0, \beta_{ij} \leftarrow 0, \gamma_{ij} \leftarrow -1, \theta_{Sj} \leftarrow 0.\ \ \forall i, j, S$
5: **for all** node i **do**
6: Update $f_{in} \leftarrow S(i)/[S_{\text{tol}}(i) - S(i)]$
7: **end for**
8: **for all** node i **do**
9: **for all** node j **do**
10: Find shortest path PATH(i, j)
11: Find $c_{ij} \leftarrow \sum_{a \in \text{PATH}(i,j)} w_a[1 + S(a)]$
12: **end for**
13: **end for**
14: **for all** edge e **do**
15: Update $c_{en} \leftarrow c_{ij}$
16: **end for**
17: **while** $F \neq \mathcal{V} - P_v$ **do**
18: $\alpha_j += U_\alpha.\ \ \forall j \in \mathcal{V}, j \notin F$
19: $\beta_{ij} += U_\beta. \forall i, j \in \mathcal{V}, j \notin F, \sum_i \beta_{ij} < f_{in}$
20: $\gamma_{ij} += U_\gamma. \forall j \in \mathcal{V}, j \notin F, T[j] = i, \sum_i \beta_{ij} \geq f_{in}$
21: **for all** node i, node j **do**
22: **if** $\alpha_j \geq c_{ij}$ **then**
23: $T[j] \leftarrow i, F \leftarrow F \cup \{j\}$
24: $C_i[j] \leftarrow i (i \in A)$ or $B[i] (i \in B)$
25: **end if**
26: **end for**
27: **for all** node i, node j, location l **do**
28: **if** $\gamma_{ij} \geq c_{ij}$ **then**
29: **if** $i \in A$ **then**
30: $C_i[j] \leftarrow i, F \leftarrow F \cup \{j\}$
31: **else if** $i \in B$ **then**
32: $C_i[j] \leftarrow B[i], F \leftarrow F \cup \{j\}$
33: **else**
34: $A \leftarrow A \cup \{i\}, B[i] \leftarrow i$
35: $R \leftarrow \emptyset$
36: **if** $\gamma_{ij} \geq c_{iln}$ **or** $\beta_{ij} > 0$ **then**
37: $C_d[j] \leftarrow i, F \leftarrow F \cup \{j\}$
38: $R \leftarrow R \cup \{j\}$
39: **end if**
40: $B[k] \leftarrow i.\ \ \forall k \in T[R]$
41: $C_i[k] \leftarrow B[i].\ \ \forall k \in T[i] \notin R$
42: $F \leftarrow F \cup \{k\}.\ \ \forall k \in T[i] \notin R$
43: **end if**
44: **end if**
45: **end for**
46: **end while**
47: Construct Steiner tree between $i \in A$
48: $L(n) \leftarrow A$
49: **end for**

4.2 Complexity Analysis of Approximation Algorithm

We now discuss the time complexity of the algorithm. We assume that there are N nodes, Q chunks, and the number of iterations of line 17 is C. Then, the complexity of the

algorithm in a grid network is $O(N^3)$. Apparently, the bottleneck of the algorithm is line 27. If we simplify the locations as the location of nodes in line 27, which is practical in the real implementation, there will be $O(N^3)$ complexity by the number of loops. Note that lines 8-13 in Algorithm 1 require the shortest path between every two nodes, which costs at most $O(N^3)$ using Floyd-Warshall algorithm. If the network topology is simple, such as a grid network, the complexity might drop to $O(N^2)$. Meanwhile, Steiner tree problem has polynomial time approximation algorithms. The work like [26] can achieve 1.55 approximation ratio at the time complexity of $O(N^3)$. The number of iterations is related to the chosen unit step U_α. If the unit step is large, it might quickly finish but may select fewer nodes and increase the *Contention Cost* of accessing phase; or if the unit is small, it might take a long time. However, it will not exceed $\max\{c_{ij}\}/U_\alpha$. If we increase α_j to become larger than the maximum of all c_{ij}, data chunks can be placed anywhere. In this case, the iteration will end. So the number of iterations is no more than $C = \max\{c_{ij}\}/U$. After all, in the worst case, for each chunk, the complexity is $O(CN^3)$. For the algorithm in total, its complexity is no more than $O(QCN^3)$. For $\max\{c_{ij}\}$, it depends on the data chunks of the network as well as the connection complexity. Some networks may have less complexity. For example, for a grid network, the number of maximum hops of the network is fixed. $O(C) = O(Q)$ since the size of $\{c_{ij}\}$ is fixed. We argue that in practical edge computing environments, the number of data chunks is a constant not depending on the size of the network. Thus, the overall complexity of a grid network is $O(N^3)$. The original approximation algorithm can be executed in polynomial time, which can still offer a 6.55 approximation ratio to the ConFL problem, and our problem as well.

Now we show that this iterative solution under proper increment unit settings will achieve the same approximation ratio as Equation (3).

Theorem 1. *The above iterative algorithm can achieve a 6.55 approximation ratio to the optimization problem (8).*

Proof. Equation (8) is the summation of a series of minimization problems. To address the problem, first, we must obtain the dual problem of each chunk represented in one of the minimization problems in Equation (8). For a linear programming problem, the maximum value of the dual problem is the same as the minimum value of the primal problem. In this problem, it is easier to solve the dual problem than the original one. The dual problem formulation can be induced from [21]. For each minimization problem, we have

$$\max \quad \sum_j \alpha_{jn} - \sum_j \beta_{vjn} \tag{9}$$

$$\text{s.t.} \quad \alpha_{jn} \le c_{ijn} + \beta_{ijn} + \sum_{Y_n} \theta_{Y_n,jn}, (\forall i \ne v, j) \tag{10}$$

$$\alpha_{jn} \le c_{vjn} + \beta_{vjn}, (\forall j) \tag{11}$$

$$\sum_j \beta_{ijn} \le f_{ij}, (\forall i) \tag{12}$$

$$\sum_j \sum_{e \in \delta(Y_n)} \theta_{Y_n,jn} < c_{en}, (\forall e) \tag{13}$$

$$\alpha_{jn}, \beta_{ijn}, \theta_{Y_n,jn} \ge 0. \tag{14}$$

In Equation (9), α, β and θ are three dual variables of x_n, y_n and z_n in Equation (3). We need to find the values of dual variables to get the optimal value of the original problem. We let $Dual_n = \max(\sum_j \alpha_{jn} - \sum_j \beta_{vjn})$ be the approximate optimal value (not the real optimal value) of Equation (9). Opt_n is the optimal value for each chunk n of Equation (8). Similarly, we let $Dual(n) = n(\sum_j \alpha_{jn} - \sum_j \beta_{vjn})$ be the approximate optimal value for the dual problem of Equation (3) and $Opt(n)$ be the optimum value of Equation (3). The approximation algorithm is based on improving the dual value to approximate the objective value. We can conclude that if there was only one chunk in the network, $Dual(1) = Dual_1$. We assume that the cost is always lower if we use caching rather than directly getting all data from the producer. Thus, according to Equations (9), (10), (11), $Dual_n < \sum_j c_{vjn}$. The result is the same as $Dual(n) < \sum_n \sum_j c_{vjn}$. We then define ϵ_{1n} and ϵ_{2n}, where $\forall n, \epsilon_{1n}, \epsilon_{2n} > 0$. Let $Dual(n) = \sum_j c_{vjn} - \epsilon_{1n}$ and $Dual_n = n\sum_j c_{vjn} - n\epsilon_{2n}$. For the first chunk, $\epsilon_{1,1} = \epsilon_{2,1}$. We assume that adding a cache should lower the total cost of facility construction. If choosing U_α and U_β properly, after the first chunk, we have $\epsilon_{1n} \ge \epsilon_{2n}$. When all chunks are cached into the network, $\sum_n Dual_n = \sum_n \sum_j c_{vjn} - \sum_n \epsilon_{1n} = n\sum_j c_{vjn} - \sum n\epsilon_{1n} \le n\sum_j c_{vjn} - \sum n\epsilon_{2n} = Dual(n)$. Since there is a 6.55-approximation algorithm for the primal-dual problem, we set k as the approximation ratio. Then $Dual_n = k \times Opt_n$ and $Dual(n) = 6.55 \times Opt(n)$. Thus, we have $\sum_n Dual_n = nDual_n = nk \times Opt_n \le Dual(n) = 6.55 \times Opt(n)$. Since $nOpt_n = Opt(n)$, We have $k \le 6.55$. We conclude that it will achieve the same approximation ratio by using the algorithm with fixed approximation ratio multiple times. □

4.3 Distributed Algorithm

The algorithm proposed above is a centralized algorithm that needs the information of connection topology of the network. However, sometimes nodes may not have such information. To address this issue, we now make some extensions to the approximation algorithm and propose a distributed algorithm. The basic idea is to keep the variables associated with a node and let the node maintain them, and send control messages within the k-hop range to obtain the contention information and inform the state change of a node.

In the distributed algorithm, the initial states of nodes are the same as the approximation algorithm. Consider that there is a new data chunk to be cached in the network. First, there will be an New Packet Info (NPI) packet informing all nodes that there is a new data chunk to cache. Then, nodes will exchange information about the contention. The exchange will be limited in the k-hop range to avoid flooding. Then node i will increase a bid denoted as α_j. If the bid can cover the estimated contention cost between two nodes i and j, node i will send a TIGHT request to node j, meaning "Can I get data from you?" and start a bid for relay cost γ_j and resource cost β_j. If the bid for relay cost covers the contention cost, the node will send a SPAN request, "Can you fetch data for me from other nodes?" A node that has received enough SPAN requests will make itself an ADMIN node, and send responses back to the nodes which sent these SPAN requests and those whose bid for resource cost was

TABLE 2
Messages in the Distributed Algorithm

Packets	Content	Range
NPI	Inform there is a new data chunk to be cached	Broadcast
CC	Contention collection request	Local
TIGHT	Inform a node for tight (bid larger than contention cost)	Local
SPAN	Inform a node for span (bid larger than relay cost)	Local
FREEZE	Response message to freeze a certain node	Local
NADMIN	Inform self admin for those nodes which tights with this node	Local
BADMIN	Inform self admin for those nodes which has adequate resources	Broadcast

large enough. Nodes that receive the response message will stop bidding. Algorithm 2 shows the detailed process for a data chunk. Whenever a node receives a message, one of the processes in Algorithm 2 will be performed on that node. If a node determines itself to be an ADMIN node, it will proactively request the data chunk from the producers. Note that all types of messages, except NPI messages and BADMIN messages, are only limited in the k-hop range.

4.4 Complexity Analysis of Distributed Algorithm

We now discuss the number of messages of the distributed algorithm. All messages that will be transmitted in the network are listed in Table 2. We assume there are N nodes and Q chunks. The number of messages is of $O(QN + N^2)$. The number of NPI messages is equal to the number of chunks. Every node will send out CC packets for each chunk, so the total number is $O(QN)$ which is the number of chunks times the number of nodes. The number of TIGHT and SPAN messages are both $O(N^2)$, where the worst case is to send every other node a TIGHT message. This is unlikely to occur since there will always be nodes that have data copies and send responses. The number of FREEZE messages is $O(N)$ since every node will send at most once when it finds a node which can cache. NADMIN and BADMIN are all messages sent from a node that is selected to become a caching node. Each caching node will only send these two messages once, which is in total at most $O(N)$. We can conclude that the total number of messages is $O(QN + N^2)$, where TIGHT, SPAN and CC are the most dominating messages in the network.

4.5 Caching for Continuously Generated Data

So far our proposed algorithms are based on the assumption that every node has the caching storage capacity to store all data chunks. In reality cases, the producers may generate data chunks continuously. For example, micro-climate sensors such as thermometers and hygrometers will generate temperature and humidity data continuously for small areas, and old data will become obsolete. Over time, data chunks may become outdated as the information is no longer valid, or caching storage may be full on some nodes, thus necessitating cache replacement. As new data are being generated, the caching decisions must be adapted continuously to ensure fair and efficient data access. Thus, we propose a continuous caching strategy, which extends our original design to meet the demand of caching scenarios over long time periods.

Algorithm 2. Distributed Algorithm

Receive NEW PACKET INFO(NPI)
  Send Contention Collection (CC(i)) Request in k hops
  while True **do**
    Gradually increase α_j Until α_j larger than one of the Con_j received.
    Send TIGHT(i) Request to j.
    Gradually increase β_j, γ_j
    if γ_j larger than Con_j **then**
      Send SPAN(i) Request to j.
    end if
  end while
end Receive
Receive CCi
  $Con+ = 1$
  Send Con back
end Receive
Receive Tight/Spani
  $T = T \cup \{i\}$
  if Node is INACTIVE **then**
    Send FREEZE(a) to $\forall j \in T$
  else if Node is ADMIN **then**
    Send FREEZE(i) to $\forall j \in T$
  else if Node is ACTIVE and Message is SPAN **then**
    $c+ = 1$
  end if
  if Node is ACTIVE and $c \geq M$ **then**
    Make myself ADMIN
    Send NADMIN(i) to $\forall j \in T$
    Broadcast BADMIN(i)
    Proactively request Data chunk from Producer
  end if
end Receive
Receive Freezei
  $a = i$
  Stop increasing α_j, β_j, γ_j
end Receive
Receive Nadmini
  $a = i$
  Stop increasing α_j, β_j, γ_j
  Send FREEZE(a) to $\forall j \in T$
end Receive
Receive Badmini
  if Node is ACTIVE and $\beta_j \geq Con_j$ and $\beta_j \leq f_i$ **then**
    $a = i$
    Stop increasing α_j, β_j, γ_j
    Send FREEZE(a) to $\forall j \in T$
  end if
end Receive

The previous algorithms can only support caching of continuously generated data chunks for a short time. If the data generation rate is larger than the expiration rate, all caching storages in the network will eventually become full. Moreover, caching storage is usually available where contention costs are high, which is the reason that those nodes are not chosen for caching previously. Therefore, it is necessary to replace chunks from some of the nodes at key

locations of low contention costs, even if their storage is full. The total contention cost for accessing the new chunk will decrease if more than one node cache the new chunk. First, the contention cost is the same on the node which swaps a chunk, since the number of chunks on that node will not change; second, caching additional chunk replicas on other nodes will decrease the total contention costs. Note that the replicas of the new chunk can be cached not only on nodes with available storage, but also on nodes that are full, in which case some old chunks must be driven out.

Carefully selecting which nodes to replace existing chunks with a new chunk is crucial for maintaining low access latency in continuous caching. In the original design, the fairness degree cost of node i will be ∞ if its caching storage is full, and then this node will no longer cache any more chunk. To enable choosing nodes with full storage, we redefine the fairness degree cost as follows.

$$f_i = \begin{cases} \frac{S(i)}{S_{\text{tol}}(i)-S(i)} & \text{if } S(i) < S_{\text{tol}(i)} \\ S_{\text{tol}}(i) + \sum_{j \in \mathcal{V}\{i\}} c_{ij} & \text{if } S(i) = S_{\text{tol}(i)}. \end{cases} \quad (15)$$

When $S(i) < S_{\text{tol}}(i)$, the fairness degree cost is the same as the original design. When $S(i) = S_{\text{tol}}(i)$, the cost is the maximum possible fairness degree cost (equal to $S_{\text{tol}}(i)$ when using the number of chunks as the metric) plus the summation of all possible contention costs for other nodes to access this chunk from node i. The redefined fairness degree cost makes it possible to choose a node with full caching storage. If the system chooses a node with full storage, such a decision will produce less overall contention costs than choosing nodes with available storage. This is because that the fairness costs will be higher on nodes which are full than on nodes with available storage. After we refine the fairness costs for the situation that nodes may be full, we can apply the new definition of f_i in Equation (15) to replace the old definition in the distributed algorithm in Algorithm 2. This will enable the ability to swap out chunks from nodes which are full in some high fairness degree cost but low contention cost cases to achieve continuous caching over time.

Note that the nodes selected to swap out and cache will not always be the same for each newly generated data chunk. The new data chunk will likely have some replicas on some other nodes with available storage. Since the network selects where to cache based on the current network state, the changes of caching storage on different nodes will have different effects on the distribution of each chunk. Meanwhile, the nodes will delete chunks if they expire. The deletion will create space for caching storage. The expiration and deletion of chunks will have even greater effects on the selection of nodes for cache replacement. Each time a new chunk is generated, we call the previous algorithm with the fixed number of chunks, and utilize the new fairness degree cost in the original algorithm to support continuous caching scenarios. Thus, the proposed strategy can be used to ensure low latency and fairness for caching over time.

By using the new fairness degree cost definition, the previous algorithm can select which nodes should cache the new data chunk. On those nodes that have full storage, they must select which existing chunks to swap out. Since we assume that all nodes need to acquire all chunks in the network, all chunks are being requested equally. Thus, the cache replacement strategies based on "recently used" are not applicable for this case. Some popular applicable replacement strategies are FIFO (first in, first out), LIFO (last in, first out) and RR (random replacement). Another simple replacement strategy based on *data value* inspired by [27] can also be applied to the continuous caching scenarios. The data value can be set by the producer of the data and it continuously decreases over time. The existing chunk with the lowest value will be swapped out to make space for new chunks.

Although different cache replacement strategies will swap out different chunks, through experiments we find that there is not any significant difference on the performances of caching. First, the selected nodes to perform cache replacement are the same no matter which replacement strategies are applied, since the algorithm for deciding these nodes is orthogonal to the replacement strategies. Second, swapping out and storing a new chunk will not change the total used storage, and the impact on contention costs throughout the network is similar among the strategies. Simulations in Section 5 show that the difference on the performances is similar on two of the replacement strategies. Third, different replacement strategies have no impact on the accessibility of other unexpired chunks. Multiple nodes will store the replicas for a chunk. When swapping out one chunk from a node, there will be other replicas of this chunk on other nodes. Thus, most nodes accessing chunks stored on other nodes are not affected very much.

As will be seen in the performance evaluation section, with the above simple extensions and modifications, the algorithm can work well in the continuous caching scenarios over time.

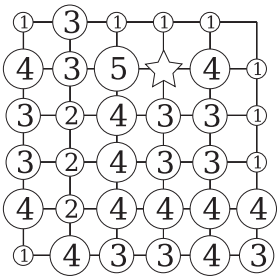
5 Performance Evaluation

In this section, we present detailed simulation and evaluation of our proposed algorithms. Our simulations focus on answering the following questions: 1) How do the proposed algorithms compare with other existing multi-hop caching algorithms on contention costs? 2) How do the proposed algorithms compare with existing algorithms on fairness? 3) What is the performance of cache replacement supported by our proposed algorithms and strategies?

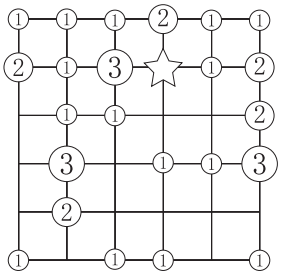
To answer these questions, we implement both our algorithms and two other existing multi-hop caching algorithms [5], [14]. To compare our algorithm with optimal solutions, we also use PuLP [28], the linear programming modulator, to implement the optimization problem. We implement these algorithms on both grid network topologies and random network topologies. We fix the producer in grid networks, and randomly choose a node as the producer in random networks. Unless otherwise specified, we put a total of 100 chunks in the network and each node is also capable of storing 100 chunks. A node will send the data to another requesting node through the shortest path in hops. We conduct our simulations on a computer equipped with an Intel Core i7-5820K and 16 GB RAM.

5.1 Comparison with Optimal Solution

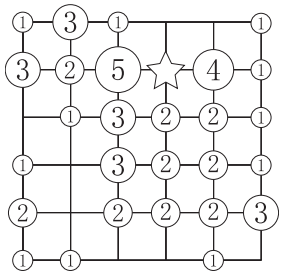
Fig. 1 shows the difference between the optimal solution and different algorithms for the first 5 chunks on a 6×6



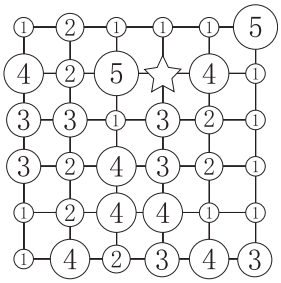
(a) Optimal solution



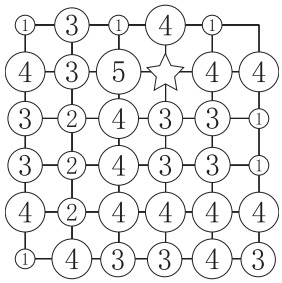
(b) Approximation algorithm



(c) Distributed algorithm



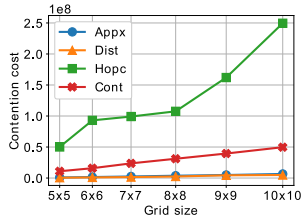
(d) Algorithm [5]



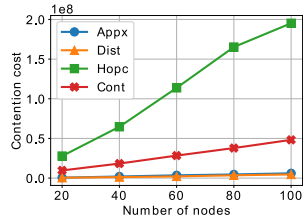
(e) Algorithm [14]

Fig. 1. The distribution of the first five data chunks in a 6×6 grid network. For (b)-(e), the area size and number in each circle show the *difference in the number of stored chunks* from the optimal solution on the respective node. The star indicates the producer. The results show that our algorithms get the closest chunk distribution to the optimal result.

grid networks. Fig. 1a shows the optimal distribution for the proposed formulation. The optimal result is obtained from the linear programming modulator. The size of the circle and the number in it show the number of chunks a node stores. The evenly distributed chunks among nodes show that most nodes participate in caching, while few nodes cached all five chunks. The more chunks are in the network, the fairer each node caches. Fig. 1b, 1c, 1d, 1e show the *difference* with optimal results for the four algorithm mentioned above. Fig. 1b and 1c are algorithms we propose, while Fig. 1d and 1e are existing algorithms to compare. The size of the circle and the number in it indicate the difference in the number of chunks that each node stores compared to the optimal results. In our algorithms, the difference with optimal solutions is smaller. The more nodes are selected to cache, the more evenly the data distribution is. Other two algorithms choose nodes based on the cost (hop count or contention) obtained from the current network topology, without considering the current state of nodes. They always choose the same group of nodes for each chunk. Thus, in these two algorithms, all 5 chunks are cached at the same group of nodes, and the difference is significantly larger.



(a) Grid networks



(b) Random networks

Fig. 2. Contention cost of four different algorithms in grid and random networks. Our algorithms perform better than both "Cont" and "Hopc" algorithms.

5.2 Contention Cost Evaluation

We now evaluate the contention cost performance of our proposed approximation algorithm (denoted in figures as Appx) and distributed algorithm (Dist). We also compare them with the two existing algorithms mentioned above, which estimate latency based on hop count (Hopc) or contention (Cont). We put 100 distinct data chunks in the networks and place these chunks by running these algorithms. We obtain the contention cost by adding all contention costs among all paths in the network. As mentioned earlier, contention cost here can be translated into data access latency.

Fig. 2a shows the contention cost for all four algorithms in different network sizes in grid networks, from 5×5 to 10×10. Our proposed algorithms perform similarly to the "Cont" algorithm, in both smaller and larger networks. The approximation algorithm is on average 10.84 percent better than the "Cont" algorithm, and the distributed algorithm is on average 7.63 percent better than the "Cont" algorithm. Meanwhile, our algorithms perform much better than the "Hopc" algorithm, with average 97.9 and 97.1 percent better respectively for approximation and distributed algorithms. We can observe that when the total number of chunks is small, the difference in contention cost is not very obvious, since the position bias strategy will provide the best place for accessing these chunks. However, as the number of chunks increases, the contention around the nodes with "best position" also increases, making the fair caching more competitive for a larger number of chunks.

We also test our algorithms in random networks. The nodes are distributed randomly in roughly the same sized area as the grid network with the nearest number of nodes. Fig. 2b shows the comparison between these four algorithms. Our algorithms also perform well in random networks, achieving significantly lower contention costs. The approximation and distributed algorithms are on average 10.26 and 7.45 percent better than "Cont" algorithm respectively.

As we mentioned in Section 4.3, in the distributed algorithm, nodes will collect local information to make estimations about the network. A good estimation is very crucial for the distributed algorithm. It gives nodes general information about the network and further guides nodes to select the values of the parameters. In general, the more information nodes have, the better estimations nodes will make. However, if the packets for information exchange traverse too many hops, it will cause more packet transmission thus more contention.

Fig. 3 plots the contention cost and the propagation scope limitation under different sizes of grid networks. We find that 2-hop limitation will give the best results in contention

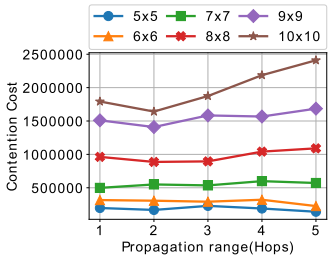
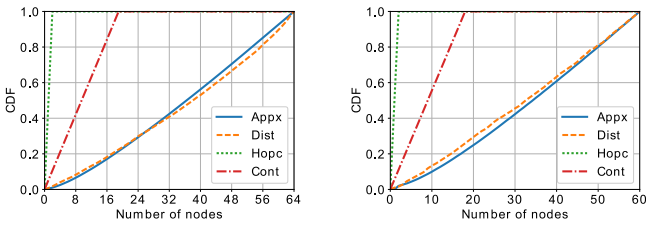


Fig. 3. Effect of message hop limitation on contention cost in the distributed algorithm. Tested on different size of grid networks. 2-Hop limit has the best result.



(a) Grid network (8 × 8) (b) Random network (60 nodes)

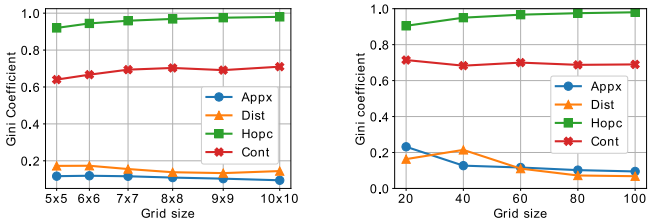
Fig. 4. Cumulative chunk distribution of four different algorithms in grid and random networks (100 chunks). Our proposed algorithms are fairer on chunk distribution.

costs. When messages are limited within 1-hop, nodes get too little information for estimation. Nodes may underestimate the contention, such that nodes at high contention locations may incorrectly decide to cache data. When the hop limitation is larger than 2, the difference between total contention cost on different propagation scopes is relatively small, since nodes have adequate information about the network. It also brings extra contention when the message propagation scope is larger. With proper information collected, nodes will have better estimations, and more nodes will be selected as caching or relay nodes. To balance between message overhead and caching performance, we therefore choose 2-hop limitation on propagation for local information exchange packets in the distributed algorithm.

5.3 Fairness Evaluation

Next, we evaluate the performance on fairness for our algorithms. As mentioned earlier, in edge computing environments, nodes are owned by individuals. With limited resources, fairness is crucial in such environments.

Fig. 4 shows the number of nodes needed to store a certain ratio of all data on an 8×8 grid network (a) and a random network contains 60 nodes (b). If more nodes participate in caching, the workload of each node will be fairer. We define p-percentile fairness as the fraction of nodes needed to cache $p\%$ of the total data. Ideally, when all nodes have the same caching load, p-percentile fairness is strictly $p\%$. The smaller it is, the more uneven the load, the less fair it will be. In the 8×8 grid network, the 75%-percentile fairness of approximation and distributed algorithms are 73 and 81 percent respectively, while 3 and 25 percent fairness for "Cont" and "Hopc" algorithms respectively. This shows that our algorithms enable more nodes to participate in caching data chunks, which makes it fairer for caching. The fair distribution is also applied in random networks, as shown in Fig. 4b. The observations are similar



(a) Grid networks (b) Random networks

Fig. 5. The Gini coefficient of four different algorithms in grid and random networks. Our proposed algorithms have a lower Gini coefficient.

on chunk distribution. It shows that our algorithms can ensure fair caching in more general cases.

Another well received measurement for fairness is the Gini coefficient. The Gini coefficient is widely used to depict income disparity, and is also used in previous works to measure fair caching [29]. The definition is as follows:

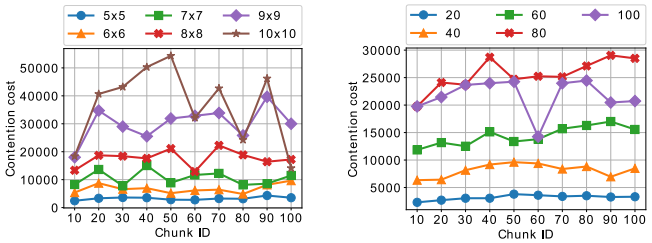
$$Gini = \frac{\sum_i \sum_j (t_i - t_j)}{2 \sum_i \sum_j t_j},$$

where t_i and t_j are the number of chunks stored in nodes i and j respectively. Note that in the denominator, t_j and t_i are commutable. It measures the inequality of cached chunks among different nodes. In general, the lower the Gini coefficient is, the fairer the network will be.

Fig. 5 depicts the Gini coefficient for all four algorithms in both an 8×8 grid network (a) and a random network that contains 60 nodes (b). Our algorithms have a Gini coefficient about 0.2. This means that the chunk distribution is fairer among all nodes. Moreover, when the network size grows, the Gini coefficient of our algorithms drops while others remain roughly the same or even increasing. The larger the number of nodes is, the more evenly chunks will be distributed in our algorithms, thus decreasing the Gini coefficient.

Dividing large data items into multiple chunks is easy to maintain, and chunks can be cached separately to enhance robustness. A chunk may have multiple copies (i.e., replicas) stored on different nodes. However, all chunks belonging to one data item must be obtained for successful data accessing. The contention cost of chunks should be roughly even, such that they can be obtained in about the same time for accessing. Otherwise, one chunk with long latency may delay the successful access of the data item. We test such per chunk fairness by putting 100 distinct chunks into the network and calculate the contention cost in accessing phase of each chunk.

Fig. 6 shows the accessing contention cost for each data chunk in different grid networks (a) and random networks (b) for the distributed algorithm. Each point depicts the average accessing cost of 10 distinct chunks on average. The accessing contention cost for each chunk is roughly the same, which shows that each chunk fairness is also maintained in our distributed algorithm. Note that in "Cont" and "Hopc" algorithms, they do not consider each chunk distribution. Thus, every chunk is stored on the same set of nodes. This will cause huge contention on a few selected nodes since they have to respond to all data requests.



(a) Grid networks (b) Random networks

Fig. 6. Contention cost (accessing phase) for 100 distinct chunks for the distributed algorithm in different sizes of networks. In general, the contention cost fluctuates in relatively small ranges for each chunk in both network settings.

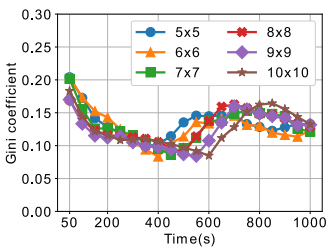


Fig. 7. The Gini coefficient in continuous scenario for different sizes of networks, each point depicts one sample of every 50 seconds. The Gini coefficient reaches an equilibrium after the rate of generating and deleting chunks stabilized.

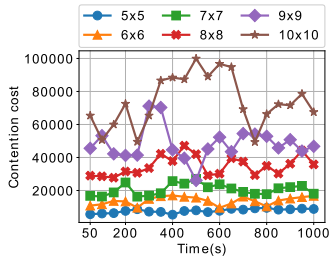
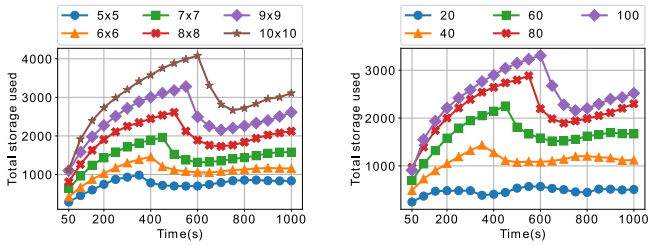


Fig. 8. Contention cost (accessing phase) for chunks in continuous scenario, each point depicts the average of every 50 seconds. The cost fluctuates for diffferent chunks in relatively small ranges.



(a) Grid networks (b) Random networks

Fig. 9. Total storage used in continuous scenario, each point depicts the average of every 50 seconds. As the Gini coefficient depicts previously, the system will reach an equilibrium as the rate of generating and deleting chunks stablilized over time.

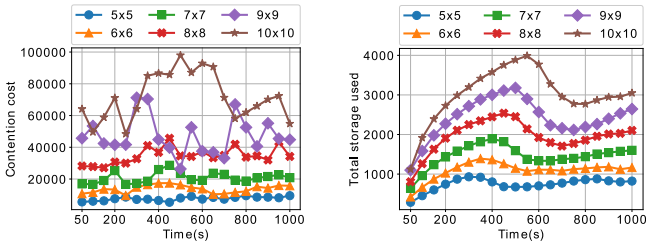
5.4 Continuous Caching Evaluation

To further understand our extended algorithms in continuous caching scenarios, we conduct the simulation over long time periods using the modified distributed algorithm. In this case, each node is capable of storing 50 chunks, and the total number of distinct data chunks is 1000. A new chunk is generated every second. Every chunk will be cached on multiple different nodes as replicas. In this scenario, every chunk will have an expiration time. Once a chunk expires, the node will delete this chunk from its caching storage, making space to store future chunks. Before expiration, if a node is already full, and the network still wants to put data onto this node, as we discussed in Section 4.5, the node will swap out one chunk. We first analyze the performance using FIFO as the replacement strategy.

Fig. 7 shows the Gini coefficient over time. As we mentioned before, it measures the inequality of cached chunks among different nodes. We conduct the simulation on different sizes of grid networks, and we sample the Gini coefficient of the entire network every 50 seconds. Overall, the Gini coefficient is limited between 0.1 to 0.2, which shows good fairness when running over time. It first gradually decreases before the first chunk starts to expire (in this case, between 300s to 600s). This shows the fact that the algorithm makes caching selections based on the current network, thus fairer over time. Meanwhile, it also agrees with the observation from the previous simulation without any replacement, where the larger size of the network, the more nodes participate in caching, the lower the Gini coefficient will be. After chunks start to expire, the Gini coefficient starts to increase and gradually achieves a stable state. Nodes swap out out-dated chunks and may not have a new chunk to store. Thus, the expiring of chunks causes the temporary increase on the Gini coefficient, since deleting the out-dated chunk is not even on every node. When the process goes on, nodes deleting expired chunks and caching new data chunk will finally achieve an equilibrated state. The change of total storage also reflects this equilibrium, which we will discuss later.

The even contention cost on accessing of each data chunk is also important in continuous caching for steady accessing. This can keep smooth data access for continuous service in the network, since all chunks of a piece of data often need to be completely collected to be useful. Fig. 8 plots the average contention cost in accessing phase for every 50 seconds. When the network size is small, the accessing cost is roughly the same. When the network size increases, the accessing cost fluctuates more, but it is still within a certain range ($5 \times n^2$ to $10 \times n^2$, where n is the total number of nodes). The fluctuation mostly occurs when chunks start to expire. After the network reaches the equilibrium, the accessing costs also remain roughly the same. In general, it shows that the proposed continuous caching can also lead to low data access latency.

The changes of the total storage used is another way to observe the behavior over long time periods. Fig. 9 depicts the total storage used throughout the entire network over time. In grid networks (a), we test from 5×5 to 10×10, and in random networks (b), we test from 20 to 100 nodes. The expiration time of every chunk is between 300s to 600s for different network sizes in both settings. Before the first chunk expires, the storage will gradually increase as more data chunks are cached. The figure also shows that our algorithms make more replicas in larger networks for easy accessing and fairness. The larger the network size, the more replicas for a chunk, thus more storage will be used. Meanwhile, the network will not be overloaded and use up all storage (despite some of the nodes will use up all storage



(a) Contention cost (accessing) (b) Total storage used

Fig. 10. The contention cost and total storage used in continuous scenarios. Caching replacement is determined by data values. The expiration threshold value is uniform random for each chunk. The result is very similar to that of Figs. 8 and 9.

and need cache replacement later). As the system running over time, after chunks start to expire, the total storage will first decrease and then increase. The network will quickly find an equilibrium state and maintain fair and efficient caching, as well as the Gini coefficient shows.

Because of these replicas, the ratio between the number of accessible chunks to the number of chunks that are not expired (hit rate) is always 100 percent for unexpired chunks. It means that even if one node swaps out a chunk in order to store a new chunk, there are other replicas in the network. The chunk can still be accessed via other replicas.

Finally, we discuss the difference in the performances of cache replacement strategies. As we mentioned in Section 4.5, the only difference on different cache replacement is which chunk the node will swap out when it is full. For FIFO, we swap out the oldest one. If we assign a data value for every chunk, the node will swap out the one with the lowest value. Other parts of the process for caching are the same for different replacement strategies.

For swapping out data chunks using data values, if it drops below a certain threshold, the chunk is marked as expired. The node will delete the chunk to make room for new chunks. We set the expiration threshold value as uniform random for each chunk, which the average time based on the value is between 300s to 600s for different network sizes, with the standard deviation of 100 seconds.

Fig. 10 depicts the total contention costs (a) and the total storage used (b) when we used the value function to decide which chunk the nodes will swap out. We test them under different sizes of grid networks. These figures are very similar to that of Figs. 8 and 9 using FIFO as the replacement strategy. The contention costs for accessing chunks are at the same level of the FIFO replacement strategy. This shows that accessing for chunks is as efficient as other replacement strategies. The total storage used is smoother when reaching the steady state of the storage. This is because the distribution of lifetime for each chunk is uniformly random. Thus, some chunks may have more lifetime. The hit rate of unexpired chunks is also 100 percent. This shows that our proposed algorithm and continuous caching strategies can also achieve the same degree of fairness and low latency despite different replacement strategies.

In summary, the above simulation results show that the extended distributed algorithm also works well on continuous caching over long time periods.

6 Conclusion and Discussion

In this paper, we propose two caching algorithms to achieve fair workload among selected caching nodes for data sharing in pervasive edge environments. We consider fairness in caching multiple data items while keeping contention cost low for data access. We propose an approximation algorithm and a distributed algorithm. Comparison with two existing algorithms on wireless network caching shows that our algorithms can achieve comparable or even lower latency while greatly improving fairness, thus data access robustness and performance.

We would like to stress that the accurate representation of contention cost or actual delay is very difficult. There are many factors that can affect them. The accurate formulation of contention cost is not the primary goal of our work in this paper. Thus, we adopt the contention-induced delay, a mature, well-studied model used in recent works to represent the contention cost. Our work focus on finding fair and efficient caching algorithms, where the topology of the network does not change significantly in short data transmission times. For high-mobility scenarios that change the topology during the data transmission, we have designed corresponding algorithms and strategies and will publish them separately.

Acknowledgments

This work is supported in part by US National Science Foundation under grant numbers CNS-1513719 and CNS-1730291.

References

[1] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)," *Relatório técnico, Telecom ParisTech*, 2011, pp. 1–6.

[2] A. I. Saleh, "An adaptive cooperative caching strategy (accs) for mobile ad hoc networks," *Knowl.-Based Syst.*, vol. 120, pp. 133–172, 2017.

[3] X. Fan, J. Cao, and W. Wu, "Contention-aware data caching in wireless multi-hop ad hoc networks," *J. Parallel Distrib. Comput.*, vol. 71, no. 4, pp. 603–614, 2011.

[4] P. Nuggehalli, V. Srinivasan, and C.-F. Chiasserini, "Energy-efficient caching strategies in ad hoc wireless networks," in *Proc. 4th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2003, pp. 25–34.

[5] J. Sung, M. Kim, K. Lim, and J.-K. K. Rhee, "Efficient cache placement strategy in two-tier wireless content delivery network," *IEEE Trans. Multimedia*, vol. 18, no. 6, pp. 1163–1174, Jun. 2016.

[6] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," *IEEE Trans. Mobile Comput.*, vol. 5, no. 1, pp. 77–89, Jan. 2006.

[7] T. Hara, "Effective replica allocation in ad hoc networks for improving data accessibility," in *Proc. 20th Annu. Joint Conf. IEEE Comput. Commun. Societies*, 2001, vol. 3, 2001, pp. 1568–1576.

[8] M. Fiore, F. Mininni, C. Casetti, and C.-F. Chiasserini, "To cache or not to cache?" in *Proc. INFOCOM*, 2009, pp. 235–243.

[9] K. Cho, M. Lee, K. Park, T. T. Kwon, Y. Choi, and S. Pack, "Wave: Popularity-based and collaborative in-network caching for content-oriented networks," in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2012, pp. 316–321.

[10] J. Li, H. Wu, B. Liu, J. Lu, Y. Wang, X. Wang, Y. Zhang, and L. Dong, "Popularity-driven coordinated caching in named data networking," in *Proc. 8th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2012, pp. 15–26.

[11] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, et al., "Named data networking (ndn) project," *Relatório Técnico NDN-0001, Xerox Palo Alto Research Center-PARC*, vol. 157, p. 158, 2010.

[12] C. Bernardini, T. Silverston, and O. Festor, "MPC: Popularity-based caching strategy for content centric networks," in *Proc. IEEE Int. Conf. Commun.*, 2013, pp. 3619–3623.
[13] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proc. 5th Int. Conf. Emerging Netw. Exp. Technol.*, 2009, pp. 1–12.
[14] P. Nuggehalli, V. Srinivasan, C.-F. Chiasserini, and R. R. Rao, "Efficient cache placement in multi-hop wireless networks," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 1045–1055, Oct. 2006.
[15] J. Sung, M. Kim, K. Lim, and J.-K. K. Rhee, "Efficient cache placement strategy for wireless content delivery networks," in *Proc. Int. Conf. ICT Convergence*, 2013, pp. 238–239.
[16] H. Farahat and H. Hassanein, "Optimal caching for producer mobility support in named data networks," in *Proc. IEEE Int. Conf. Commun.*, 2016, pp. 1–6.
[17] N. Kumar and J.-H. Lee, "Peer-to-peer cooperative caching for data dissemination in urban vehicular communications," *IEEE Syst. J.*, vol. 8, no. 4, pp. 1136–1144, Dec. 2014.
[18] G. Cornuéjols, G. L. Nemhauser, and L. A. Wolsey, "The uncapacitated facility location problem," Carnegie-Mellon Univ., Pittsburgh, PA, USA, DTIC Document, Cornell University Operations Research and Industrial Engineering, 1983, https://ecommons.cornell.edu/bitstream/handle/1813/8491/TR000605.pdf?sequence=1
[19] A. Gupta, A. Kumar, and T. Roughgarden, "Simpler and better approximation algorithms for network design," in *Proc. 35th Annu. ACM Symp. Theory Comput.*, 2003, pp. 365–372.
[20] C. Swamy and A. Kumar, "Primal-dual algorithms for connected facility location problems," in *Proc. Int. Workshop Approximation Algorithms Combinatorial Optimization*, 2002, pp. 256–270.
[21] H. Jung, M. K. Hasan, and K.-Y. Chwa, "A 6.55 factor primal-dual approximation algorithm for the connected facility location problem," *J. Combinatorial Optimization*, vol. 18, no. 3, pp. 258–271, 2009.
[22] F. Eisenbrand, F. Grandoni, T. Rothvoß, and G. Schäfer, "Connected facility location via random facility sampling and core detouring," *J. Comput. Syst. Sci.*, vol. 76, no. 8, pp. 709–726, 2010.
[23] A. Tomazic and I. Ljubic, "A grasp algorithm for the connected facility location problem," in *Proc. Int. Symp. Appl. Internet*, 2008, pp. 257–260.
[24] S. Guha and S. Khuller, "Greedy strikes back: Improved facility location algorithms," *J. Algorithms*, vol. 31, no. 1, pp. 228–248, 1999.
[25] Y. Yang and R. Kravets, "Achieving delay guarantees in ad hoc networks by adapting ieee 802.11 contention windows," in *Proc. IEEE INFOCOM*, 2006, pp. 1–8.
[26] G. Robins and A. Zelikovsky, "Improved steiner tree approximation in graphs," in *Proc. 11th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2000, pp. 770–779.
[27] L. Turczyk, M. Groepl, N. Liebau, and R. Steinmetz, "A method for file valuation in information lifecycle management," *Proc. Americas Conf. Inf. Syst.*, 2007, Art. no. 38.
[28] S. Mitchell, M. OSullivan, and I. Dunning, "Pulp: A linear programming toolkit for python," Univ. Auckland, Auckland, New Zealand, 2011. [Online]. Available: http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf
[29] D. Wei, K. Zhu, and X. Wang, "Fairness-aware cooperative caching scheme for mobile social networks," in *Proc. IEEE Int. Conf. Commun.*, 2014, pp. 2484–2489.

Yaodong Huang received the BEdegree in computer science and technology from the University of Electronic Science and Technology of China, Chengdu, China, in 2015. He is now working toward the PhD degree in computer engineering at Stony Brook University. His research interests include mobile edge computing, with a focus on data caching, storage, and blockchain technology on edges.

Xintong Song received the BS degree from Peking University, in 2012 . He is working toward the PhD degree in the Institute of Network Computing and Information Systems (NCIS), Peking University, China. His research interest includes mobile computing and wireless networks. He joined this work during his one-year visit at Stony Brook University in 2016.

Fan Ye received the BE and MS degrees from Tsinghua University, and the PhD degree from the Computer Science Department, UCLA. He is an associate professor with the ECE Department, Stony Brook University. He has published more than 90 peer reviewed papers that have received more than 10,000 citations according to Google Scholar. He has 26 granted/pending US and international patents/applications. His research interests include mobile sensing platforms, systems and applications (LBS and health), internet-of-things, edge computing, and wireless and sensor networks.

Yuanyuan Yang received the BEng and MS degrees in computer science and engineering from Tsinghua University, Beijing, China, and the MSE and PhD degrees in computer science from Johns Hopkins University, Baltimore, Maryland. She is a SUNY distinguished professor of computer engineering and computer science with Stony Brook University, New York, and is currently on leave at the National Science Foundation as a program director. Her research interests include edge computing, data center networks, cloud computing, and wireless networks. She has published about 400 papers in major journals and refereed conference proceedings and holds seven US patents in these areas. She is currently an associate editor-in-chief for the *IEEE Transactions on Cloud Computing* and an associate editor for the *ACM Computing Surveys*. She has served as an associate editor-in-chief and associate editor for the *IEEE Transactions on Computers* and associate editor for the *IEEE Transactions on Parallel and Distributed Systems*. She has also served as a general chair, program chair, or vice chair for several major conferences and a program committee member for numerous conferences. She is a fellow of the IEEE.

Xiaoming Li is a professor in computer science and technology and former director of the Institute of Network Computing and Information Systems (NCIS), Peking University, China. His research interests include mobile computing and wireless networks, search engine and web mining, web technology enabled social sciences, reputation issues in cyber systems, and computing system virtualization. He serves as a vice president for the Chinese Computer Federation and is chairing the Advisory Subcommittee for Undergraduate Computing Education in China. He is a senior member of the IEEE and a member of Eta Kappa Nu.