A service caching method for mobile edge computing

Xiaojun Wu¹, Yong Wang¹, Sheng Yuan*, Peiru Xie¹
¹School of Software Engineering, Xi’an Jiaotong University, Xi’an, Shaanxi, 710049, China
*sheng@xjtu.edu.cn

Abstract. Mobile Edge Computing (MEC) is considered as a key technology to support 5G networks. It can push cloud computing functions to the edge of the network, and achieve the purpose of reducing latency and reducing energy consumption through computing offloading and edge caching. In the MEC system, by caching computing services from the cloud center to the network edge, MEC can provide users with more services, and also be able to make greater use of network resources and improve system performance. However, which services to cache at edge nodes and how to cache these services have become an important challenge of the research. In order to solve the aforementioned problems, in this paper, we analyze the popularity of each cached object firstly, and propose a novel service caching algorithm, which aims to reduce system delay and improve resource utilization. In addition, we design a novel cache replacement algorithm that successfully predict future requests by analyzing historical data, thereby greatly improves the cache hit rate. Finally, experiments prove that our algorithms have greatly improved system performance and hit rate.

1. Introduction

Mobile edge computing (MEC) is a new network architecture, which aims to reduce delay by making the cloud computing function closer to the Internet of Things and mobile users [1]. MEC pushes the function of cloud computing to the network edge which is closer to users, and provides users with cloud computing and caching functions through edge nodes [2]. A large number of researches focus on the offloading in MEC, which has the following premise: MEC server already has the ability to execute all tasks. In other words, the MEC server has all the resources and services before the task is offloading. However, in actual application scenarios, it cannot be guaranteed that all the edge servers have all the resources and services required to execute each task. Because the execution of some tasks may require specific services, and the execution of some tasks has dependency. In this case, it is necessary to cache these services to the edge nodes in advance. In fact, both service cache and dependency will affect the performance of task offloading. If the service cache or dependency is not considered when application is offloading, these applications may not execute successfully [3].

In the existing research on service caching, such as [4], it pointed that service caching referred to caching application services and their related databases/libraries in the edge server (e.g. MEC-enabled BS), thereby enabling corresponding computation tasks to be executed. It proposed an efficient online algorithm, which jointly optimized dynamic service caching and task offloading to address a number of key challenges in MEC systems. In [5], it proposed a novel adaptive user-managed service placement mechanism, which jointly optimized a user's perceived-latency and service migration cost, weighted by user preferences. In [6], it proposed a two-time-scale framework that jointly optimized service placement and request scheduling, under storage, communication, computation, and budget constraints. In [7], it considered a heterogeneous MEC system, and focused on the problem of placing
multiple services in the system to maximize the total reward. And it showed that the problem was NP-hard via reduction from the set cover problem, and proposed a deterministic approximation algorithm to solve the problem.

These studies have effectively solved many challenges of service caching, but they only considered from the perspective of the system, and ignored the state of each service object before requesting and the changes in the caching process. It will lead to lower hit rate. In order to solve this problem, this paper focuses on service objects in order to study service placement and cache replacement.

2. System architecture
In order to cache and response services clearly, we design a caching system based on hierarchical structure. Figure 1 shows the architecture of the edge cache. The “cloud” is the cloud center of MEC system, which has powerful storage and computing capabilities, and stores the services of the whole system in the edge cache. The “edge caching” is composed of edge servers and base stations. Each server connects to a base station in order to establish wireless connection with mobile users. The “users” is the mobile users, which deploys mobile devices.

Our work focuses on the MEC layer, and addresses which popular services are cached on the edge nodes and how to cache these services on the edge nodes.

2.1. Hierarchical architecture
User layer: Each user connects to a MEC server through the uplink. As the task requester, users can execute computing through connected MEC server. The system assumes that there are \( n \) users, which form a set \( \mathcal{U} = \{U_1, U_2, ..., U_n\} \), called user set. Suppose that there are \( k \) users in the service area of each server, and these \( k \) users form a set \( \mathcal{U} = \{U_1, U_2, ..., U_k\} \). Herein, each user is represented as a one-dimensional vector \( u = \{\varepsilon, S_j, r\} \) and \( 1 \leq j \leq m \). The definition of \( m \) is given in the edge layer. The symbol \( \varepsilon \) represents user number, which is used to distinguish the user who send the task requests. \( S_j \) is the edge server connected to users, and \( r \) is the distance between users and servers.

Edge layer: As service nodes, each edge server has cache function, and executes tasks, and sends the computing results to users. Assume that there are \( m \) edge servers in the system, which form a set \( \mathcal{S} = \{S_1, S_2, ..., S_m\} \). Herein, each server is represented as a one-dimensional vector \( s = \{\delta, h, c\} \). The symbol \( \delta \) represents server number, which is used to distinguish each server. \( h \) is the number of users connected to servers, and \( c \) is the service objects cached by the server. All servers are connected with each other to share information.

Cloud layer: As cloud center, it has powerful storage capacity and computing capacity. Therefore, it stores all the service objects for executing tasks. Simultaneously, it can execute any request from users. The cloud central database is represented by \( \mathcal{DB}^{cloud} \), and all the services form a set \( \Psi \).

Figure 1. System architecture.
2.2. Cooperative caching

In order to use the storage resources and computing resources of edge nodes effectively, each edge node needs to cooperate. In the existing research, one of the most popular cooperative caching schemes is to create a central storage for servers, so that all servers can obtain the required cache objects from the central storage [8]. The disadvantage of this is that it increases the extra time and energy consumption to get data from the central storage. Another existing solution is not only keeping cache objects by itself, but also having a synchronous database in each server, so as to obtain the services from other servers [9]. The disadvantage of this design scheme is that it will waste a lot of cache space and generate a lot of redundancy. Our approach is that all servers keep only one synchronized lookup table, which records all the services cached by each server, and it can quickly locate which server has cached which services through the lookup table. This solution overcomes the shortcomings of remote central storage and synchronized databases, and achieves the best space occupation and search time.

2.3. Cache Replacement

When a request is missed, traditional algorithms such as First Input First Output (FIFO), Least Frequently Used (LFU), Least Recently Used (LRU) all assume that every object has the same state before caching. For example, FIFO only considers the arrival order of cache objects, LFU only considers the number of requests of cache objects, LRU considers both the arrival time and the number of requests of cache objects, and most other studies don’t consider the difference of cache objects before arrival. If these algorithms are directly applied to the service cache, it is obviously inefficient. In order to solve the above problems, so as to adapt to the cache in edge computing to the greatest extent, and improve the hit rate as much as possible, this paper designs a caching algorithm that not only predicts future requests based on the historical state, but also considers the differences between each cached object before being cached to edge nodes.

3. Problem formulation

In the scenario of this paper, each user initially sends a task to the connected MEC server, and executes the task through the service scheduling of the MEC layer. After the task is executed, the result is fed back to users. In order to evaluate the performance of the caching system, we need to consider the time cost and space cost. This paper assumes that the MEC server has sufficient cache space, so only the time cost is considered, that is, the delay consumed during the execution of tasks.

Because each user needs to send tasks to MEC servers, there are three cases in the process of executing the task. The first case is user $U_{i}$ sends the task to the connected MEC server $S_{i}$, and $S_{i}$ has cached the required services locally. In this case, the task is executed in $S_{i}$, and the results are fed back to $U_{i}$. The second case is that user $U_{2}$ sends the task to the connected MEC server $S_{i}$, but $S_{i}$ does not cache the required services locally, and find other server $S_{j}$ through synchronous look-up table. The required services are cached in $S_{j}$, then $S_{i}$ send the task to $S_{j}$ and this task is executed in $S_{j}$, and the result will feed back to the $U_{2}$. The third case is that user $U_{3}$ sends the task to the connected MEC server $S_{i}$, and the whole MEC layer doesn’t cache the required services. In this case, $S_{i}$ send the task to the cloud center for executing, and return the result to $U_{3}$ directly after execution.

The waiting time of task in MEC server is $t_{\omega} = 1/(\mu_{i} - \lambda_{i})$, where $\lambda_{i}$ represents the request arrival rate of the i-th edge server task, and $\mu_{i}$ represents the average service rate of each server processing requests. The transmission delay between server and user is $t_{a} = d_{l}/r_{s,u}$, where $d_{l}$ is the number of tasks, $r_{s,u}$ is the transmission rate form $S_{i}$ to user. The transmission delay between servers and services is $t_{b} = d_{l}/r_{s} + L_{s}/\Lambda$, where the transmission rate is $r_{s}$, the length of optical fiber is $L_{s}$, and the refractive index of the fiber is $\Lambda$. The transmission delay from cloud center to user is $t_{c} = d_{l}/r_{c,u}$, where $r_{c,u}$ is the transmission power of the server. In the first case, the system delay consumed by requesting the i-th service is $W_{i} = t_{0} + t_{a}$. In the second case, the system delay consumed by
requesting the i-th service is $W_i = t_0 + t_a + t_b$. In the third case, the system delay consumed by requesting the i-th service is $W_i = t_0 + t_c$. Then the total delay of the system can be described as:

$$W = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{z} \rho(i)W_i$$

(1)

where $\rho(i)$ is the probability of the i-th service being requested, $m$ is the number of servers in the system, and $z$ is the number of services in the system.

4. Problem formulation Service placement and cache replacement algorithm

4.1. Service placement algorithm

When the MEC server receives tasks, it firstly looks for the required service in the local cache. In order to describe cache placement clearly, we need to analyze the popularity of each task. In the cache system, popularity is used to measure the possibility of an object being requested in a certain period of time generally, which is an important indicator of the cache system. According to this indicator, if the edge cache can cache the popular objects in time, the hit rate will be improved greatly, and the cache efficiency of the system will also be increased. Many studies have shown that the popularity ofcached objects generally follows the Zip-f distribution [10], which can be expressed as:

$$p(k) = \frac{k^{-\theta}}{\sum_{k=1}^{z} k^{-\theta}}$$

(2)

In the formula, $\theta$ is Zip-f index which satisfies $0 < \theta < 2$ and $\theta \neq 1$, and $z$ is the total number of cached objects. In other words, $p(k)$ is the possibility of the k-th cache object being requested. Herein, it is assumed that the size of each cached object is $d_k$. In addition, the popularity of the object is constant in a certain period of time.

Definition 1: The set of $\Phi$ is the set of all services in the local cache of MEC layer, and is a subset of all services in the cloud central database.

Definition 2: The set of $\Omega$ is the most popular set of objects, and is a subset of set of $\Phi$.

Next, we can model the cached object as a vector $c = \{\sigma, p, v\}$, where $\sigma$ represents cached object number which is used to distinguish each cached object, $p$ represents the popularity of the object, and $v$ represents the indicator variable of cache placement. The specific meaning is as follows:

$$\begin{cases} 
\delta, \text{number of MEC server} \\
0, c_i \notin DB \\text{edge} \\
-1, c_i \in \Phi \text{ and } c_i \notin \Omega
\end{cases}$$

(3)

The specific cache placement process is as follows: firstly, all cached objects in the cloud central database are sorted in descending order of popularity, and the top $f\%$ of the objects are taken out and put in the set of $\Phi$. Take the top part from set of $\Phi$ and put them into the set of $\Omega$. Cache the objects in set of $\Omega$ to all MEC servers. Then, cache the remaining objects to every MEC servers in the circular way. For example, there are three MEC servers in a specific scenario. There are 100 services in the cloud central database. If we need to cache the top 14% of services to the MEC layer, so there are 14 objects in the set of $\Phi$ need to be cached. Specifically, $\Phi = \{c_1, c_2, ..., c_{14}\}$. And we put top two objects into $\Omega$ from $\Phi$, specifically $\Omega = \{c_1, c_2\}$. After placement according to the above scheme, the service objects in the cache area of each MEC server are as follows:

$$\begin{cases} 
S_1: c_1, c_2, c_3, c_6, c_9, c_{12} \\
S_2: c_1, c_2, c_4, c_7, c_{10}, c_{13} \\
S_3: c_1, c_2, c_5, c_8, c_{11}, c_{14}
\end{cases}$$

(4)

4.2. Cache replacement algorithm

In the process of service request, when a task is offloading to an MEC server $S_i$, it will be searched in the local cache firstly. If the local has already cached the required service, this service will be assigned
to the task. If the local doesn’t cache the required services, then we need to find other servers through synchronous look-up table. If the required services are already cached in other servers, locate the server $S_j$ which cache required service. Then $S_j$ offloads the task to $S_j$ which assigns the required services to execute the task. If all other MEC servers do not have the required services for caching, the task is sent to the cloud center for execution. The specific execution process can refer to Algorithm 1.

Table 1. Service caching algorithm

| Algorithm 1 | Service Caching Algorithm Based on Popularity (SCAP) |
|--------------|------------------------------------------------------|
| **Input:**   | the set of MEC servers, $\mathcal{DB}^{\text{cloud}}$, $\mathcal{DB}^{\text{edge}}$, the set of $\Psi$. |
| **Output:**  | the objects of $\Phi$ set are cached in the MEC layer. |
| **Initialize:** | $c_i \in \Psi$ for $\forall i$, $\Psi \leftarrow \mathcal{DB}^{\text{cloud}}$. |
| 1: | for all $c_i \in \Psi$ do |
| 2: | sort $\Psi$ in descending and number $\text{num}_i$ for each $c_i$ |
| 3: | Take out the $c_i$ at the top and put them in set of $\Phi$ and set of $\Omega$ in order to $\Omega \subseteq \Phi \subseteq \Psi$ |
| 4: | for each $MEC_i$ do |
| 5: | if $c_i \in \Omega$ |
| 6: | put it into $MEC_i$ and $c_i \in \mathcal{DB}^{\text{edge}}$ |
| 7: | else |
| 8: | put it into the next $MEC$ as circular way and $c_i \in \mathcal{DB}^{\text{edge}}$ |

When there is a miss, we need to perform the above process. In addition, the missed services need to be cached in the server. If the server still has the remaining cache space before caching, the service can be cached directly. However, if there is no remaining cache space, a caching algorithm must be executed to select and replace, which is inevitable.

The specific implementation process is as follows: firstly, we sort all the services in the set of $\Psi$ in descending order of popularity. The number for each service is from large to small, and the number of the last service is number 1. Set a value for each service at the same time, which is called activity $\gamma$. It is used to indicate the activity degree of the service during the caching process of the system. The more activity it is, the more likely it will be requested again in the future. Initially, the activity of the service is the service number in the set of $\Psi$. When a request for a service hits, then $\gamma+1$, otherwise $\gamma-1$. When replacement is needed, the minimum $\gamma$ will be selected. After the replacement, check all the services in the cache, and the service which $\gamma=0$ is eliminated. However, when it’s in the process of replacement, request for replacement $\gamma$ less than the minimum $\gamma$, there will be problems. Therefore, our algorithm will not replace in this process. The following algorithm 2 is the specific process.

### 5. Experiments and results

In the experiments, we will simulate the caching process of the edge nodes. In the simulation environment, cache and network resources can be considered at the same time. Then we analyze the data generated in a specific working environment to conduct experimental comparison and optimization, and then reflect the superiority of our algorithm. In the experiments, all servers and users are distributed randomly and discretely. Virtual Machine (VM) instances are deployed in a simulated cloud environment, and each MEC node uses a lightweight Docker container. Each user randomly assigns an independent task, and sends the task to the MEC server to execute. The experiments require that the tasks of each user offload to the MEC server for executing, and we evaluate the impact of the caching strategy on system performance.
Table 2. Cache replacement algorithm

**Algorithm 2** Cache Replacement Algorithm Based on Activity (CRAA)

**Input**: MEC servers, $\Psi, \Phi, \Omega$, the number of each $c_i$ as $num_i$, service requires set $RQ$.

**Output**: number of requests and hits

**Initialize**: $c_i \in \Psi$ for $\forall i, \Psi \leftarrow DB^{\text{cloud}} \Omega \subseteq \Phi \leftarrow DB^{\text{edge}}$, activity $\gamma_i = num_i$

1: for each $MEC_i$ do
2:    get $r_q \in RQ$
3:    for each $c_i$ do
4:        if $c_i$ is hit
5:            $\gamma_i++$
6:        else //miss
7:            $\gamma_i--$
8:    if $\gamma_i==0$
9:        $c_i \notin DB^{\text{edge}}$ // $c_i$ is eliminated
10: if remaining cache space $\text{Size}==0$ do
11:    for each $c_f$
12:        get minimum $\gamma_{min}$
13:        if $\gamma_k < \gamma_{min}$
14:            reject caching
15:        else
16:            cache $c_j$ and $c_{min} \notin DB^{\text{edge}}$;

In this section, we evaluate our algorithm proposed. In order to show the advantages of our algorithms, we introduce three comparison strategies. They are Random Cache Strategy (RCS), Independent Cache Strategy (ICS), and Cooperative Caching Not Sharing (CCNS). Each algorithm builds the same MEC servers, users and tasks of the same size. The comparison between our algorithm and the other three algorithms in order to show the superiority of our algorithm. In order to distinguish the comparison in existing researches, our algorithm considers the popularity of the object, and the other three algorithms do not consider the popularity. Thus, it can show advantages of our algorithm.

Based on the results in Figure 2, it shows the relationship between system delay and number of tasks. The result shows that system delay all increase as the number of tasks in these four algorithms. The reason is that the more tasks, the busier system, thus system delay will increase. At the same time, our algorithm keeps the lowest system delay; next from low to high are CCNS, ICS and RCS. The reason is as follows: CCSN does not share cache objects between MEC servers, but it uses cooperative cache, so the system performance gap with our algorithm is the smallest. RCS and ICS do not consider both cooperation and sharing, and do not consider the popularity as well, so they have the highest system latency.

In Figure 3, system delay all decrease as the cache space increases in these four algorithms. It can be explained as that the larger storage space, the more objects are cached, so the system delay will be reduced. Simultaneously, our algorithm has the smallest delay; next from low to high are CCNS, ICS
and RCS. The reason can be expressed as follows: when the storage space increases, our algorithm can exert the highest advantage. CCNS can also play to its advantage. The cache space has little effect on ICS and RCS, and they have the highest latency.

Figure 2. As the number of tasks change, comparison between other three algorithms and our algorithm in system delay.

In order to evaluate the superiority of our algorithm, we introduced three competitors. They are FIFO, LFU and CLFU (It is improved LFU algorithm which proposed in another study) [9]. These three competitors execute after the services are deployed, so the state of each cached object before the request is considered. In Figure 4, as the time slot increases, the hit rate will decrease. The hit rate of our algorithm can keep a relatively high level, followed by CLFU, but there is little difference between them. The reason is that CLFU does not consider the impact of changes in popularity on hit rate during the system operation, but it considers the request frequency, so it has a higher hit rate. Because FIFO and LFU have no improvement in cache replacement, they have the lowest hit rate.

Figure 4. As the time slot changes, comparison between other three algorithms and our algorithm in hit rate.

Based on the results in Figure 5, as the cache space increases, the hit rate increases gradually. Our algorithm has the highest hit rate and FIFO has the lowest hit rate. The reason is as follows: when the cache is placed, it is according to the order of popularity from high to low. Therefore, the earliest placed service has the highest popularity. Our algorithm considers the popularity fully, so it has the highest hit rate. Whereas, FIFO has the priority to eliminate the most popular service when it is eliminated, therefore it has the lowest hit rate.

Figure 5. As the cache capacity changes, comparison between other three algorithms and our algorithm in hit rate.

6. Conclusion

In the MEC scenario, caching services to edge nodes in advance can greatly improve the efficiency and performance of the system. We study the service caching in MEC, and propose a service placement algorithm based on popularity and a novel cache replacement algorithm. Simultaneously, cooperation between MEC servers, the sharing of all cache objects, and the activity of each cache
object are considered. Because the system considers the state of each cached object before requesting, it has a great benefit. In the experiment, we compare the differences between different service placement algorithms and different cache algorithms. The results show that our algorithm achieves remarkable results in system delay and hit rate.

References
[1] 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, Aug. 2017.
[2] ETSI, GS MEC 002 V2.1.1, “Multi-Access Edge Computing (MEC); Phase 2: Use Cases and Requirements”, Oct. 2018.
[3] S. Sundar and B. Liang, “Offloading dependent tasks with communication delay and deadline constraint,” in IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 2018, pp. 37–45.
[4] J. Xu, L. Chen and P. Zhou, “Joint service caching and task offloading for mobile edge computing in dense networks”, IEEE INFOCOM 2018-IEEE Conference on Computer Communications., 2018, pp. 207-215.
[5] T. Ouyang, R. Li, X. Chen, Z. Zhou and X. Tang, “Adaptive user-managed service placement for mobile edge computing: An online learning approach”, IEEE INFOCOM 2019-IEEE Conference on Computer Communications., 2019, pp. 1468-1476.
[6] V. Farhadi, F. Mehmeti, T. He, T. La Porta, H. Khamfroush, S. Wang, et al., “Service placement and request scheduling for data-intensive applications in edge clouds”, IEEE INFOCOM 2019-IEEE Conference on Computer Communications., 2019, pp. 1279-1287.
[7] S. Pasteris, S. Wang, M. Herbster and T. He, “Service placement with provable guarantees in heterogeneous edge computing systems”, IEEE INFOCOM 2019-IEEE Conference on Computer Communications., 2019, pp. 514-522.
[8] A. Abouaomar, A. Filali, and A. Kobbane, “Caching, device-to-device and fog computing in 5G cellular networks generation: Survey,” in 2017 International Conference on Wireless Networks and Mobile Communications (WINCOM).
[9] E. E. Ugwuanyi, S. Ghosh, M. Iqbal, T. Dagiuklas, S. Mumtaz and A. Al-Dulaimi, "Co-Operative and Hybrid Replacement Caching for Multi-Access Mobile Edge Computing," 2019 European Conference on Networks and Communications (EuCNC), 2019, pp. 394-399.
[10] Y. Li, H. Xie, Y. Wen, C. Y. Chow, and Z. L. Zhang, “How much to coordinate? optimizing in-network caching in content-centric networks,” IEEE Transactions on Network & Service Management, vol. 12, no. 3, pp. 420–434, Jul. 2015.