Edge Service Caching with Relaying to Reduce Latency

***Index Terms*—Service caching, Multi-access Edge Computing, Delayed hits, Latency, Online algorithms**

I. Introduction

"In Multi-access Edge Computing (MEC), compute and storage are brought closer to users in order to reduce the latency and network congestion incurred while accessing the cloud. A *service* processes inputs from devices and generates outputs, such as one for multiplayer AR games [1]. A service may be downloaded to the edge and cached to reduce latency.

In MEC networks, when a request for a service arrives at an edge node, if the service is cached, the request is served from the cache. If the service is not cached, either: 1) The request is sent to the cloud (we call this *Request Forwarding*) so that it can be processed in the cloud and the service is not downloaded. This may be because this service is not expected to be requested again, it is a service that has high fetch time and high resource requirements, etc. or 2) The request is sent to the cloud or a neighbouring edge where it is cached and when it is estimated that it will be required in the future, the service is downloaded from the cloud or from a neighbouring edge and cached. When the service is being downloaded but is not yet available in cache, another service request that arrives may be buffered (we call this a *delayed hit*) or forwarded to the cloud or a neighbouring edge where it is cached."

"Recent work discusses caching of services considering the resource availability at the edge [2]. There are algorithms for service caching to minimize cost (defined as the sum of the downloading and forwarding times) [3], and those that consider relaying (sending the service request to nearby caches) and request forwarding [4]. Fan et al. [5] has the objective of finding an online service caching algorithm with provably small caching regret. *All the above discuss reducing or minimizing cost and do not consider delayed hits, to the best of our knowledge.*

A service request can be forwarded to the cloud or another edge and simultaneously a service download initiated. Since empirically the download cost is far higher than the cost of forwarding the service request [6], the *latency* in this case is due to forwarding the request to the cloud or another edge and getting a response, or waiting for the download to complete, if time incurred for that is lower than the time to forward the request and get a response. The *cost* incurred is the cost to send a request to the cloud or another edge and then to download the service. *Thus in service caching the total cost and the total latency are two different parameters* unlike in content caching [7]. *Since optimizing latency is an important requirement for low-latency edge services [8], we propose to minimize latency while keeping cost as a constraint.*"

We address the following problem: What is a deterministic online algorithm for caching and replacing services at the edge, without making any assumptions on the arrival patterns of requests, considering Delayed hits and Request forwarding, with the possibility of Relaying the request to a nearby edge or downloading from a nearby edge where it may be already cached, and with the objective of minimizing *Latency* and reducing cost ? We call this the DRL-Relay problem. Our contributions are: 1) We formulate the offline version of DRL-Relay as an optimization problem and analyse its complexity. 2) We propose an online algorithm for DRL called Online-DRL-Relay 3) We evaluate Online-DRL-Relay using synthetic and Google cluster traces and compare its latency and cost with the theoretical minimum latency.

We make the following assumptions: 1) All edges are connected to each other with a uniform latency. 2) All edges allow the same number of elements in their cache and the caches have the same upper limits for CPU, RAM and disk.

II. Motivation

All requests are routed through a central controller in a centralized solution such as that of Tan et al. [4]. This does not address the following problems: 1) How will each edge maintain the information on which edge(s) has (have) the requested service ? How accurate will this information be, as edge caches may be evicting files? In a centralized solution when a request is received at the controller, the controller may relay the request to an edge, but the edge may evict that service before the request arrives, causing a cache miss. 2) More importantly, all requests from all edges now need to be sent to the controller, for which an additional fixed cost is incurred. 3) The controller needs to have 3 hash tables indexed with the service id, which requires space. Instead, is there a way to do this in a distributed manner at each edge? For this, can we use tiered caches?

He at al. [9] discuss a way for each edge to probabilistically know which is the closest edge that has a particular service cached. However, it does not say how this information is updated across edges. That is, what exactly the protocol for this is? How frequently is it updated or when is it updated?

ISBN 978-3-903176-63-8

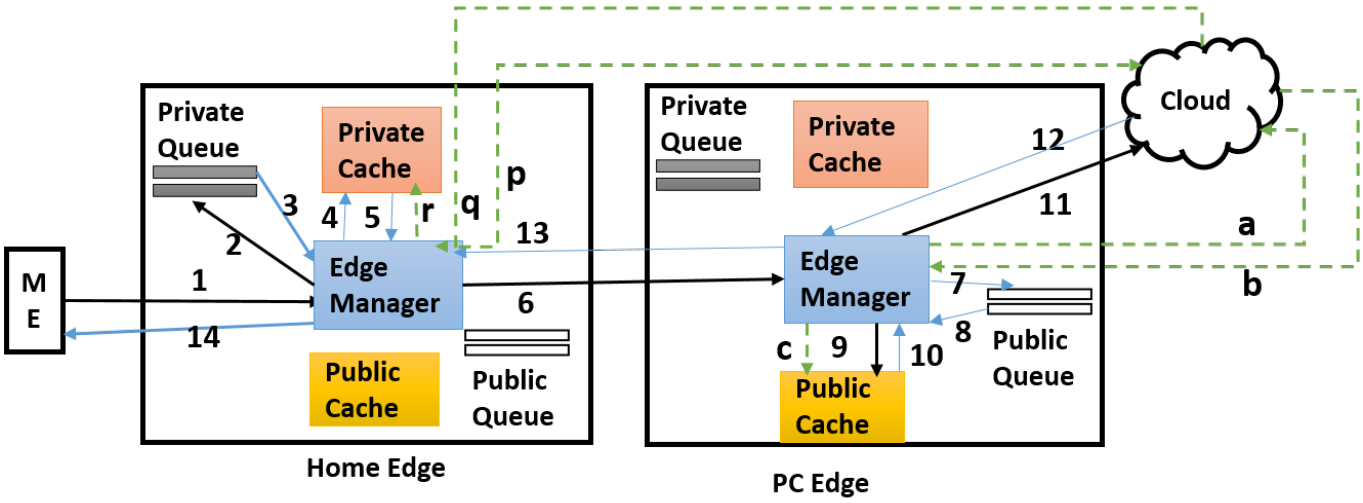

Fig. 1: Edge Caching Architecture with Multiple Caches

III. System Model

Each edge cache consists of an Edge Manager, a Private Cache and a Public Cache, as shown in Fig.1.

In the first version of the solution, assume that a service is always downloaded from the cloud and never from a nearby edge (thus it will be possible to evict a service instantaneously from an edge without checking for ongoing downloads). We also assume (optimistically) that a service is available in a neighbouring cache.

Assume that the services to be cached are $s_1, s_2, ..., s_n$. A cache may be divided over e edge servers. The hash value of s_i corresponding to each request r_i for a service s_i is computed to get a key. Each of the e caches is assigned a unique region in the key space. Corresponding to this hash, it is clear on which edge a service could be Potentially Cached (called the *PC edge*) and then the request is sent to that edge. The *home edge* (the edge where the request first arrives) can never know for certain if the edge where the service is supposed to be cached has the service cached[1]. This is because by the time the request is sent to the PC edge, the PC edge may have evicted it. The caches so far referred to are called *public caches*.

When a service request arrives at an edge from an end user, the edge queues it in its private queue ($1-2$ in Fig.1). The edge dequeues each service from that queue (3) and checks for that service in its private cache. If it is a miss ($4-5$), the following actions are taken: 1) if the service is being downloaded to the private cache, the request is buffered if the remaining time to download the service to the private cache is less than the time for the next action; that is, the time to forward the request to a neighbouring edge and get a response (lr_i). 2) if the service is not being downloaded to the private cache, it is queued in the public queue of the appropriate PC edge (the PC edge may be the home edge) ($6-7$).

The edge dequeues each service from its public queue (8) and checks for that service in its public cache. If it is a miss ($9-10$), one of the following actions is taken: 1) if the service is being downloaded to the public cache, the request is buffered if the remaining time to download the service is less than the time to forward the request to the cloud (l_i) and get a response. 2) if the service is not being downloaded to the public cache, the request is forwarded to the cloud. The response obtained from the cloud is forwarded to the user ($11-12-13-14$).

[1]prove this

IV. Optimization Formulation for a Centralized Offline Algorithm

A centralized offline algorithm may be proposed. The objective is to minimize the sum of latencies of all service requests arriving at all edges.

V. Distributed Online Algorithms

A. Tiered Caching

See Algorithm 1. A request may be downloaded and stored to the appropriate public cache ($a-b-c$) regardless of which edge requested that service under the following conditions: 1) If the elapsed time between the first request for a service i and the current request for the same service (T_i) exceeds the download time of that service from the cloud (M_i) or 2) if the total fetch time of requests (relaying time (lr_i) + time to forward the request to the cloud (l_i) and get a response[2]) for the same service ($lr_i + l_i = Lt_i$) exceeds the download time of that service from the cloud (M_i).

But this does not account for popularity at a particular edge. That is, some requests may be more popular at a particular edge compared to other edges and therefore they need not be delayed by the time required to send the request to a neigbhouring edge. Therefore, each edge can have a smaller *private cache*. A request to a particular edge from a user may be downloaded and stored to the private cache of that edge ($p-q-r$) under the following conditions: 1) If the elapsed

[2]This assumes that all requests are relayed, thus ignoring requests from the home edge to itself.

time between the first request for the service and the current request for the same service exceeds the download time of that service from the cloud (T_i) or 2) if the total relaying time of requests for the same service exceeds the download time of that service from the cloud (Lr_i).

Each edge cache will run LandLord-RR [10] (Algorithm 2) independently on its public cache to evict requests when the cache is full and a service is to be cached.Since requests are queued, there will be no issue of concurrent accesses.

Each edge cache will run LandLord-RR [10] (Algorithm 2) independently on its private cache to evict requests when the cache is full and a service is to be cached.

Each edge will measure its own request latency. The objective is to reduce the sum of latencies of the requests that arrive at all the edges.

It is possible that multiple edges download the same service. The cost incurred due to downloads must be less than a maximum value.

If a home edge does not coincide with the PC edge, the minimum worst case time to forward a request increases by the relay time compared to the case when a home edge coincides with the PC edge.

B. Tiered caching with latency-cost tradeoff

Lt_i,T_i and Lr_i may be divided by γ to tradeoff cost for latency. When $\gamma = 1$, cost reduces but latency increases. When $\gamma > 1$, latency reduces but cost increases.

C. Tiered Caching with Downloading Allowed from Neigbouring Caches

In this algorithm, a download to a private cache may be from a neigbouring cache instead of the cloud. This will require the PC edge to check if a service to be evicted is being downloaded and if so, delay eviction until the download is complete. Additionally, if a PC edge does not have the service to be downloaded, the request will be forwarded to the cloud, which can make the download process also have false positives. This will be future work.

VI. Investigations

1) What are the tradeoffs between edges informing each other of their cache status versus fixing the location of the items to be cached? Our solution requires no information about the status of other caches, but complete information about the potential location of each service.
2) In the long run, each private cache will have the MRU items cached. The public cache will have the remaining items cached. Is this correct? Can this be proved? Does this depend on the queue service rate?
3) Is there a way to estimate the probability of a miss in the private and public caches? Is it worth estimating?
4) How to estimate the size of the request queue? How will this affect latencies? The queueing delay can be added to the relaying time, but how (this can be done in a real network and then the relay time will be variable)?

Algorithm 1 Online algorithm to reduce latency: Online-DRL-Relay

procedure GetPenalty(r_i,$tstamp$,l_i,$miss_i$)
  $T_i = tstamp - miss_i$
  if $flag = priv$ **then**
    $L_i = lr_i$*Number of request forwards to neighbouring edges from the time $miss_i$, including $miss_i$ ▷ Assuming PC edges always cause a hit
  else
    $L_i = lr_i + l_i$*Number of request forwards to the cloud from the time $miss_i$, including $miss_i$
  end if
  return (T_i, L_i)
end procedure
procedure ProcessPending(r_i,M_i,cpu, ram,$disk$,$pending$,$miss_i$)
  for each $s_i \in pending$ **do**
    if s_i has completed download **then**
      LandLord-RR(s_i,M_i,cpu,ram,$disk$)
      Remove s_i from $pending$
      $miss_i = -1$
    end if
  end for
end procedure
procedure Main
  ▷ r_i is a service request, s_i is a service corresponding to it and l_i the latency incurred in forwarding the request to the cloud and getting a response. M_i is the download cost of service s_i.
  ▷ $priv\_pending = pub\_pending = NULL$.
  ▷ $privmiss_i = pubmiss_i = -1$ when a service is requested first or when a service is evicted from cache.
  Start a thread that processes $priv\_q$: HANDLE_SERVICE_REQUEST(s_i, lr_i, r_i, $tstamp$, $privmiss_i$,$priv\_q$,priv,$priv\_cache$,pub_q)
  Start another thread that processes pub_q : HANDLE_SERVICE_REQUEST(s_i, l_i, r_i, $tstamp$, $pubmiss_i$,pub_q,pub,pub_cache,$cloud$)
  while True **do**
    ProcessPending(r_i,M_i,cpu,ram,$disk$, $priv\_pending$, $privmiss_i$)
    ProcessPending(r_i,M_i,cpu,ram,$disk$, $pub\_pending$, $pubmiss_i$)
    if request r_i arrives for service s_i from a user **then**
      Queue it in $priv\_q$
    else ▷ The request is from another edge
      Queue it in pub_q
    end if
  end while
end procedure

procedure HANDLE_SERVICE_REQUEST(s_i, f_i, r_i, $tstamp$, $miss_i$,q,$flag$,$cache$,$dest$)
▷ $dest$ is the cloud if there is a public cache miss and it is a public cache indicated by pc if it is a private cache miss
s_i=Dequeue(q) ▷ Dequeue one request from the queue
if $flag = priv$ **then**
$pc = region(hash(s_i))$ ▷ Find which public cache may host this service
end if
if $s_i \notin cache$ **then**
If $miss_i$ is -1, $miss_i = tstamp$ ▷ Time stamp of the very first miss or the first miss after an eviction
if $flag = priv$ **then**
(T_i, L_i)=GETPENALTY(r_i,$tstamp$,f_i,$miss_i$,$priv$)
else
(T_i, L_i)=GETPENALTY(r_i,$tstamp$,f_i,$miss_i$,pub)
end if
if ($Ti \geq Mi$ or $Li \geq Mi$) and $s_i \notin pending$ **then** ▷ Consider relay time instead of forwarding time to cloud
Initiate download
Add s_i to $pending$
Forward r_i to $dest$ ▷ A miss
else
if ($T_i < M_i$ or $L_i < M_i$) and $s_i \notin pending$ **then**
Forward r_i to $dest$ ▷ A miss
else if $s_i \in pending$ **then**
if remaining time to download$\leq f_i$ **then**
Buffer r_i ▷ A delayed hit
else
Forward r_i to $dest$ ▷ A miss
end if
end if
end if
else
LANDLORD-RR(g, $cost$, cpu, ram, $disk$)
end if
end procedure

5) An item that is popular across caches will get downloaded to multiple private caches. What is the implication of this?
6) What is the optimal size of the private cache with respect to the public cache?
7) When an item is cached or evicted, the information is broadcast. Each edge builds and maintains a Hierarchical Counting Bloom Filter tree with this information.
8) What is the impact of various arrival distribution patterns at various edges?
9) It is assumed that accessing each edge from another edge has the same cost. If that is not so, how to know the nearest cache?
10) What should the sizes of each cache be? What if they are variable?
11) Suppose each edge has a different access pattern. How

Algorithm 2 Landlord With Resource Checks

This algorithm is the same as in [10]
Each service g has a real value $credit[f]$ and CPU, RAM and Disk resources associated with it. The maximum number of items in cache is C. Each of CPU, RAM and Disk resources has a maximum permissible value.
procedure LANDLORD-RR(g, $cost$, cpu, ram, $disk$)
if g is not in the cache **then**
if adding g to cache exceeds the maximum resource requirements of CPU, RAM or Disk **then**
$c = 0$, $r = 0$, $d = 0$ ▷ To store the cumulative requirements
while True **do** ▷ Keep evicting files until the resource requirements of g are met
$eligible\_for\_eviction$ = DECREASECREDIT()
for each item f in $eligible\_for\_eviction$ **do**
Subtract $max(0, cpu–c)$,$max(0, ram-r)$, $max(0, disk - d)$ from those of f. If a value is negative, store it as 0.
end for
Sort the modified $eligible\_for\_eviction$ in ascending order of the sum of cpu,ram and $disk$ and store in $sorted\_f$ ▷ As less items as possible are evicted to accommodate the new entry
for each item f in $sorted\_f$ **do**
Evict f
Add the cpu, ram and disk values of f to c, r and d respectively ▷ Update the cumulative requirements
if all resource requirements of g are met now **then**
Cache g, setting its $credit = cost$ and CPU, RAM and Disk values to cpu, ram and $disk$
return
end if
end for
end while
end if
if number of items in cache exceeds C **then**
$eligible\_for\_eviction$ = DECREASECREDIT()
Evict an item from $eligible\_for\_eviction$
Cache g, setting its $credit = cost$ and CPU, RAM and Disk values to cpu, ram and $disk$
return ▷ Only one item needs to be evicted
else
Cache g, setting its $credit = cost$ and CPU, RAM and Disk values to cpu, ram and $disk$
end if
else ▷ g is in the cache
Reset $credit[g]$ to cost(g).
end if
end procedure
procedure DECREASECREDIT()
For each item f in cache, decrease $credit[f]$ by $\Delta \cdot$ size$[f]$, where $\Delta = \min_{f \in \text{cache}} \frac{credit[f]}{size[f]}$.
$items$ = Items from $cache$ with $credit[f] = 0$
return $items$
end procedure

will the results change? Unlike the case of independent caches, the arrival distribution of requests at each cache relative to the other will matter. How?

12) What is the competitive ratio of the online algorithm?
13) How to formulate the distributed online algorithm such that it is asymptotically convergent to that of the centralized offline problem? (S. Boyd and L. Vandenberghe, Convex Optimization. Cambridge, U.K.: Cambridge Univ. Press, 2004; Distributed Online Algorithm for Optimal Real-Time Energy Distribution in the Smart Grid , Feburary 2014)

VII. Evaluation

A. Testing with synthetic data

Multiple threads are spawned with two threads each corresponding to an edge. The first thread generates requests and queues them and the second thread reads the queue, accesses the cache and decides if it is a hit or a miss. Each thread generates its own requests from the same range [0,a]. If the generated request is potentially in its own cache it queues in its own cache, else it queues in the appropriate cache. Its queue can contain requests from itself and from other threads.

B. Testing with google cluster data

Take consecutive 10K lines from the dataset and assign it to one edge. The first thread reads the data and sends to the appropriate edge.

Compare this with the case where each thread maintains its own edge cache independently, without relaying.

Compare with the optimal centralised offline algorithm. Since the requests are known in advance each edge knows what to evict.

Compare with each edge having a different request arrival pattern. That is, the inter-arrival distance changes.

VIII. Future Work

IX. Acknowledgements

This work was funded by the Science and Engineering Research Board (SERB), DST, Govt. of India, under the Project Code SPG/2021/002505.

References

[1] R. Singh, R. Sukapuram, and S. Chakraborty, "Mobility-aware multi-access edge computing for multiplayer augmented and virtual reality gaming," in *NCA*, vol. 21. IEEE, 2022, pp. 191–200.

[2] C.-K. Huang and S.-H. Shen, "Enabling service cache in edge clouds," *ACM Trans. on Internet of Things*, vol. 2, no. 3, pp. 1–24, 2021.

[3] T. Zhao, I.-H. Hou, S. Wang, and K. Chan, "Red/led: An asymptotically optimal and scalable online algorithm for service caching at the edge," *IEEE JSAC*, vol. 36, no. 8, pp. 1857–1870, 2018.

[4] H. Tan, S. H.-C. Jiang, Z. Han, and M. Li, "Asymptotically optimal online caching on multiple caches with relaying and bypassing," *IEEE/ACM Trans. on Netw.*, vol. 29, no. 4, pp. 1841–1852, 2021.

[5] S. Fan, I.-H. Hou, V. S. Mai, and L. Benmohamed, "Online service caching and routing at the edge with unknown arrivals," in *ICC*. IEEE, 2022, pp. 383–388.

[6] C. Zhang, H. Tan, G. Li, Z. Han, S. H.-C. Jiang, and X.-Y. Li, "Online file caching in latency-sensitive systems with delayed hits and bypassing," in *INFOCOM*. IEEE, 2022, pp. 1059–1068.

[7] J. Yao, T. Han, and N. Ansari, "On mobile edge caching," *IEEE Commn. Surveys & Tutorials*, vol. 21, no. 3, pp. 2525–2553, 2019.

[8] H. A. Alameddine, M. H. K. Tushar, and C. Assi, "Scheduling of low latency services in softwarized networks," *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 1220–1235, 2019.

[9] Q. He, S. Tan, F. Chen, X. Xu, L. Qi, X. Hei, H. Jin, and Y. Yang, "Edindex: Enabling fast data queries in edge storage systems," in *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2023, pp. 675–685.

[10] S. Deka and R. Sukapuram, "Edge service caching with delayed hits and request forwarding to reduce latency," 2024.