RetroRenting: An Online Policy for Service Caching at the Edge

Lakshmi Narayana V S Ch, Sharayu Moharir, and Nikhil Karamchandani
Department of Electrical Engineering, Indian Institute of Technology Bombay

***Abstract*—The rapid proliferation of shared edge computing platforms has enabled application service providers to deploy a wide variety of services with stringent latency and high bandwidth requirements. A key advantage of these platforms is that they provide pay-as-you-go flexibility by charging clients in proportion to their resource usage through short-term contracts. This affords the client significant cost-saving opportunities, by dynamically deciding when to host (*cache*) its service on the platform, depending on the changing intensity of requests.**

A natural caching policy for our setting is the Time-To-Live (TTL) policy. We show that TTL performs poorly both in the adversarial arrival setting, i.e., in terms of the competitive ratio, and for i.i.d. stochastic arrivals with low arrival rates, irrespective of the value of the TTL timer.

We propose an online caching policy called RetroRenting (RR) and show that in the class of deterministic online policies, RR is order-optimal with respect to the competitive ratio. In addition, we provide performance guarantees for RR for i.i.d. stochastic arrival processes and prove that it compares well with the optimal online policy. Further, we conduct simulations using both synthetic and real world traces to compare the performance of RR and its variants with the optimal offline and online policies. The simulations show that the performance of RR is near optimal for all settings considered. Our results illustrate the universality of RR.

I. Introduction

Widespread adoption of smartphones and other handheld devices over the last decade has been accompanied with the development of a wide variety of mobile applications providing a plethora of services. These applications often rely on cloud computing platforms [1] to enable delivery of high-quality performance anytime, anywhere to resource-constrained mobile devices. However, the last few years have seen the emergence of applications based on machine learning, computer vision, augmented/virtual reality (AR/VR) etc. which are pushing the limits of what cloud computing platforms can reliably support in terms of the required latency and bandwidth. This is largely due to the significant distance between the end user and the cloud server, which has led the academia and the industry to propose a new paradigm called edge computing [2] whose basic tenet is to bring storage and computing infrastructure closer to the end users. This can help enable applications with ultra-small network latency and/or very high bandwidth requirements, which cannot be reliably supported by the backhaul connection. As a concrete example, consider a user in a wildlife sanctuary, capturing the scene around her live on a mobile device, which relays the image/video to an edge server. Using its much higher computational and storage capabilities, an application on the edge server can continually detect species of plants, animals, birds and relay this information back to the end user device where it can be overlaid onto the live stream to provide a much richer viewing experience. Broader applications of edge computing include industrial robotics/drone automation, AR/VR-based infotainment and gaming, autonomous driving and the Internet of Things (IoT). While there are now several industry offerings of dedicated edge computing platforms, e.g., Amazon Web Services [3] and Microsoft Azure [4], there have also been proposals to augment cellular base stations [5] and WiFi access points [6] so that they can act as edge servers.

Edge computing platforms enable an application provider to 'cache' its service at servers close to the end users. In this paper, we say that a service is cached at an edge server, if all the data and code needed to run the service has been downloaded from a remote/back-end server (possibly in the cloud) and cached on the edge server. Thus the edge server can handle service requests on its own without requiring to communicate with the back-end server. Edge servers are often limited in computational capability as compared to cloud servers [7], and hence there might be a limit on the number of parallel requests they can serve for the cached service. An application provider can avail this ability to cache on the edge server in return of a cost which is in proportion to the amount of resources used and/or the duration of rental. Since computing platforms usually provide pay-as-you-go flexibility [8], the client can dynamically decide when to cache or evict the service at the edge, depending on the varying number of arriving service requests. The application provider needs to design an efficient service caching policy which can help minimize the overall cost of deploying the service.

Most of the literature on caching policies has focused on the related *content caching* problem [9], which deals with the problem of delivering content (for example video or music) to end users by deploying storage caches close to the end user. There are several key differences between the content caching and the service caching problem. In the former, if a content is not currently cached and a request arrives for it (*cache miss*), the content has to be fetched from a back-end server. On the other hand, in the latter, each time a service request arrives at the edge server and the service is not cached, there are two options: *(a) request forwarding* which simply forwards the service request to the back-end server, which then carries out all the relevant computation for addressing the request; and *(b) service download* which downloads all the data and code needed for running the service from the back-

ISBN 978-3-903176-29-4 © 2020 IFIP

end server and caches it on the edge server. The cost for these two actions is different, depending for example on the amount of network bandwidth needed or the latency incurred for each of them. Motivated by empirical evidence [2], [10], a natural assumption is that the cost of forwarding a single request to the back-end server is lower than the cost of downloading the entire service to cache it on the edge server [11]. Our goal in this work is to design online service caching policies for the application provider which aim to minimize the total cost it incurs for serving requests, which is a combination of the request forwarding cost, the service download cost and the edge server rental cost. We consider two classes of request arrival processes: (i) *adversarial arrivals*: the request sequence is arbitrary and the performance of any online policy is measured by its competitive ratio, which provides a worst-case guarantee on its performance for any request arrival sequence in comparison to the optimal offline policy, which has knowledge of the entire arrival sequence a-priori. (ii) *i.i.d. stochastic arrivals*: requests are generated according to an i.i.d. stochastic process and we compare the expected cost of a proposed policy with that of the optimal online policy.

A. Our Contributions

A natural policy for our setup is the Time-To-Live (TTL) policy [12], which is popular in the content caching literature. Under the TTL policy, each cache miss triggers a download of the service to the edge server where it is then retained for some fixed amount of time. We show that TTL performs poorly both in terms of the competitive ratio for arbitrary arrivals and for i.i.d. stochastic arrivals with low arrival rates, irrespective of the value of the TTL timer. Given the limitations of TTL, we propose an online caching policy called RetroRenting (RR) which uses the history of request arrivals to decide when to cache or evict the service at the edge server. Under the adversarial request setting, we show that RR is order-optimal with respect to the competitive ratio in the class of deterministic online policies. In addition, we also provide performance guarantees for RR under i.i.d. arrivals and prove that it compares very well with the optimal online policy in this setting. In addition to our analytical results, we conduct simulations using both synthetic and real world traces to compare the performance of RR and its variants with the optimal offline and online policies. Our simulations show that the performance of RR is near optimal for all settings considered. These results combined illustrate the universality of RR.

B. Related Work

The emergence of such edge computing platforms [13]–[15] has been accompanied with various academic works which model and analyse the performance of such systems. We briefly discuss some of the relevant works in the literature.

One approach towards designing efficient edge computing systems is to formulate the design problem as a large one-shot static optimization problem which aims to minimize the cost of operating the edge computing platform [5], [7], [16]–[18]. Our work differs from this line of work in that we are interested in designing online algorithms which adapt their service placement decisions over time depending on the varying number of requests.

An approach to modeling time-varying requests is to use a stochastic model as done in [19]–[21].Our work differs from these works in that in addition to stochastic request models, we also focus on the case of arbitrary request arrival processes and provide ‘worst-case’ guarantees on the performance of our proposed schemes instead of ‘average’ performance guarantees. This can be vital in scenarios where the arrival patterns change frequently over time, making it difficult to predict demand or model it well as a stochastic process.

The work closest to ours is [11] which considers an edge server with limited memory K and an arbitrary request process for a catalogue of services. This work studies the design of service caching policies which minimize the cost incurred by the edge server for deploying the various services. The authors propose an online algorithm called ReD/LeD and prove that the competitive ratio of the proposed scheme is at most $10K$. Unlike [11], we study the problem from the perspective of an application provider and design cost-efficient service caching policies which dynamically decide when to cache or evict the service at the edge.

Finally, as mentioned before, the problem of service caching does resemble the content caching problem but with some key differences. Content caching has a rich history, see for example [9], [22]–[26]. A popular class of online caching policies is the Time-To-Live (TTL) policy [12], which downloads a content to the cache upon a cache miss and then retains it there for a certain fixed amount of time. In this work, we consider a variant of the TTL policy for service caching and demonstrate that it performs poorly in several cases.

II. System Setup

A. Network Model

We study a system consisting of a back-end server and an edge server in proximity to the end-user. The back-end server always stores the service. The service can be cached on the edge server by paying a renting cost. On paying this cost, requests can be served at edge free of cost, subject to an upper limit on the number of concurrent requests being served at the edge. In addition, requests can be served by the back-end server at a non-zero cost. The back-end server can serve all the requests that are routed to it.

B. Arrival Process

We consider a time-slotted system and consider both adversarial and stochastic arrival processes. In the adversarial setting, we make no assumptions on the arrival sequence. In the stochastic setting, we make the following assumption.

Assumption 1: (i.i.d. stochastic arrivals) The number of requests arriving in a time-slot is independent and indentically distributed across time-slots. More specifically, let X_t be the number of requests arriving in time-slot t. Then, for all t,

$$\mathbb{P}(X_t = x) = p_x \text{ for } x = 0, 1, 2, \cdots .$$

C. Sequence of Events in a Time-slot

The following sequence of events occurs in each time-slot. We first have request arrivals. If the service is cached on the edge server, requests are served locally subject to the constraints on the computation power of the edge server, else requests are forwarded to the back-end server. The system then makes a caching decision (fetch/evict/no change).

D. Cost Model and Constraints

Our cost model builds on the model proposed in [11] and extends it to the setting where cache space can be rented in a dynamic manner by paying a renting cost. For a given caching policy $\mathcal{P}$, the total cost incurred in time-slot t, denoted by $C_t^{\mathcal{P}}$, is the sum of the following three costs.

- *Service cost* ($C_{S,t}^{\mathcal{P}}$): Each request forwarded to the back-end server is served at the cost of one unit.
- *Fetch cost* ($C_{F,t}^{\mathcal{P}}$): On each fetch of the service from the back-end server to cache on the edge-server, a fetch cost of $M(> 1)$ units is incurred.
- *Rent cost* ($C_{R,t}^{\mathcal{P}}$): A renting cost of $c(\geq 0)$ units is incurred to cache the service on the edge server for a time-slot.

The number of requests that can be served by the edge server in a time-slot is limited to $\kappa \in \mathbb{Z}^+$, where $\mathbb{Z}^+$ is the set of all positive integers. Let r_t be an indicator of the event that the service is cached on the edge server during time-slot t. It follows that

$$C_t^{\mathcal{P}} = C_{S,t}^{\mathcal{P}} + C_{F,t}^{\mathcal{P}} + C_{R,t}^{\mathcal{P}}, \tag{1}$$

$$\text{where, } C_{S,t}^{\mathcal{P}} = \begin{cases} X_t - \min\{X_t, \kappa\} & \text{if } r_t = 1 \\ X_t & \text{otherwise,} \end{cases}$$

$$C_{F,t}^{\mathcal{P}} = \begin{cases} M & \text{if } r_t = 0 \text{ and } r_{t+1} = 1 \\ 0 & \text{otherwise,} \end{cases}$$

$$C_{R,t}^{\mathcal{P}} = \begin{cases} c & \text{if } r_t = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Remark 1: We limit our discussion to the case where $c \in [0, \kappa)$ because, for $c \geq \kappa$, it is optimal to forward all requests to the back-end server, irrespective of the value of M and the arrival sequence.

E. Algorithmic Challenge

The algorithmic challenge is to design a policy which decides when to cache the service on the edge server. Caching policies can be divided into the following two classes.

Definition 1: (Types of Caching Policies)

- *Offline Policies*: A policy in this class knows the entire request arrival sequence a-priori.
- *Online Policies*: A policy in this class does not have knowledge of future arrivals.

We design an online policy which makes caching decisions based on the request arrivals thus far and the various costs and constraints, i.e., the rent cost (c), the fetch cost (M), and the edge service constraint (κ).

F. Metric and Goal

The optimal offline and online policies serve as benchmarks to evaluate the performance of the proposed policy. We use different cost metrics for the adversarial and stochastic request arrival settings.

1) Adversarial arrivals: For the adversarial setting, we compare the performance of a policy $\mathcal{P}$ with the performance of the optimal offline policy (OPT-OFF). The goal is to design a policy $\mathcal{P}$ which minimizes the competitive ratio $\rho^{\mathcal{P}}$ defined as

$$\rho^{\mathcal{P}} = \sup_{a \in \mathcal{A}} \frac{C^{\mathcal{P}}(a)}{C^{\text{OPT-OFF}}(a)}, \tag{2}$$

where $\mathcal{A}$ is the set of all possible finite request arrival sequences, $C^{\mathcal{P}}(a)$, $C^{\text{OPT-OFF}}(a)$ are the overall costs of service for the request arrival sequence a under online policy $\mathcal{P}$ and the optimal offline policy respectively.

2) i.i.d. stochastic arrivals: For i.i.d. stochastic arrivals (Assumption 1), we compare the performance of a policy $\mathcal{P}$ with the performance of the optimal online policy (OPT-ON) . The goal is to minimize $\sigma_T^{\mathcal{P}}$, defined as the ratio of the expected cost incurred by policy $\mathcal{P}$ in T time-slots to that of the optimal online policy in the same time interval. Formally,

$$\sigma^{\mathcal{P}}(T) = \frac{\mathbb{E}\left[\sum_{t=1}^{T} C_t^{\mathcal{P}}\right]}{\mathbb{E}\left[\sum_{t=1}^{T} C_t^{\text{OPT-ON}}\right]}, \tag{3}$$

where $C_t^{\mathcal{P}}$ is as defined in (1).

III. Main Results and Discussion

Next, we state and discuss our main results. We provide an outline of the proof of Theorem 1 in Section V. Refer to [27] for detailed proofs of all results presented in this section.

A. Our Policy: RetroRenting (RR)

A caching policy determines when to fetch and cache the service and when to evict the service from the cache. The RR policy makes these decisions in each time-slot by evaluating if it made the right decision in hindsight. We first provide an overview of the RR policy.

To fetch: Let the service not be cached at the beginning of time-slot t and $t_{\text{evict}} < t$ be the time when the service was most recently evicted by RR. The RR policy searches for a time-slot τ such that $t_{\text{evict}} < \tau < t$, and the total cost incurred is lower if the service is fetched in time-slot $\tau - 1$ and cached during time-slots τ to t than if the service is not cached during time-slots τ to t. If there exists such a time τ, the RR policy fetches the service in time-slot t.

To evict: Let the service be in the cache at the beginning of time-slot t and $t_{\text{fetch}} < t$ be the time when the service was most recently fetched by RR. The RR policy searches for a time-slot τ such that $t_{\text{evict}} < \tau < t$, and the total cost incurred is lower if the service is not cached during time-slots τ to t and fetched in time-slot t than if the service is cached during time-slots τ to t. If there exists such a time τ, the RR policy evicts the service in time-slot t.

Refer to Algorithm 1 for a formal definition of the RR policy. The notation used in Algorithm 1 is summarized in Table I.

Symbol	Description
t	Time index
M	Fetch cost
c	Rent cost per time-slot
κ	Maximum number of requests that can be served by the edge server in a time-slot
x_t	Number of requests arriving in time-slot t
r_t	Indicator variable; 1 if the service is cached in time-slot t and 0 otherwise
$(x_l - \kappa)^+$	$\max\{x_l - \kappa, 0\}$

TABLE I: Notation used in Algorithms 1

Algorithm 1: RetroRenting (RR)

```
1  Input: Fetch cost $M$ units, rent cost $c$ units per time-slot, request arrival sequence: $x_t$, $t > 0$
2  Output: Caching strategy $r_t$, $t > 0$
3  Initialize: Caching variable $r_1 = t_{\text{fetch}} = t_{\text{evict}} = 0$
4  for each time-slot $t$ do
5      $r_{t+1} = r_t$
6      if $r_t = 0$ then
7          for $t_{evict} < \tau < t$ do
8              if $\sum_{l=\tau}^{t} x_l \geq (t-\tau+1)c + M + \sum_{l=\tau}^{t}(x_l - \kappa)^+$, then
9                  $r_{t+1} = 1$, $t_{\text{fetch}} = t$
10                 break
11             end
12         end
13     end
14     if $r_t = 1$ then
15         for $t_{fetch} < \tau < t$ do
16             if $\sum_{l=\tau}^{t} x_l + M < (t-\tau+1)c + \sum_{l=\tau}^{t}(x_l - \kappa)^+$, then
17                 $r_{t+1} = 0$, $t_{\text{evict}} = t$
18                 break
19             end
20         end
21     end
22 end
```

Remark 2: Note that in time-slot t, the computation and storage complexities of the RR policy scale as O(t) (if either $t_{\text{fetch}} = 0$ or $t_{\text{evict}} = 0$). This is indeed a limitation of the RR policy since, in the worst case, the computational and storage complexities increase linearly with time.

To overcome this limitation, we propose an efficient variant of the RR policy called the RR$_u$ policy. The only difference between the RR and RR$_u$ policies is that, at time t, the RR$_u$ policy considers the arrival sequence in the previous at most u time-slots to make its caching decision whereas the RR policy can potentially look at the entire arrival sequence from $t = 0$ to make its caching decision for each time-slot (refer to lines 7 and 15 in Algorithm 1). Therefore, under RR$_u$, the range for τ in lines 7 and 15 in Algorithm 1 are replaced with $\max\{t_{\text{evict}}, t-u\} < \tau < t$ and $\max\{t_{\text{fetch}}, t-u\} < \tau < t$ respectively.

Remark 3: Note that the computational and storage complexity of the RR$_u$ policy is $O(u)$ and does not scale with time as was the case for the RR policy. Since $x_l > 0$, for all l, for the conditions in lines 7 and 15 of RR$_u$ to be satisfied, $u > \frac{M}{\kappa - c}$ and $u > \frac{M}{c}$ respectively. Therefore, we impose the condition that $u > \max\left\{\frac{M}{\kappa-c}, \frac{M}{c}\right\}$.

B. Performance guarantees for RR and RR$_u$

1) Adversarial arrivals: Our first theorem characterizes the performance of RR in the adversarial arrivals setting.

Theorem 1: Let ρ^{RR} be the competitive ratio of RR as defined in (2). Then,

$$\rho^{\text{RR}} \leq \left(5 + \frac{\kappa}{M} - \frac{4c}{\kappa}\right).$$

Since this result holds for all finite request arrival sequences, Theorem 1 provides a worst-case guarantee on the performance of the RR policy as compared to that of the optimal offline policy. Recall that unlike the RR policy, the optimal offline policy knows the entire arrival sequence a-priori.

The competitive ratio of RR improves as the fetch cost (M) and rent cost (c) increase, however, it increases linearly with κ. Our next result shows that the competitive ratio of *any* deterministic online policy increases linearly with κ.

Theorem 2: Let $\mathcal{P}$ be any deterministic online policy and let $\rho^{\mathcal{P}}$ be the competitive ratio of this policy as defined in (2). Then,

$$\rho^{\mathcal{P}} \geq \begin{cases} 1 + \dfrac{\kappa}{c+M} & \text{if } \kappa \geq \dfrac{c(c+M)}{M} \\ \dfrac{\kappa}{c} & \text{otherwise.} \end{cases}$$

From Theorems 1 and 2, we conclude that the RR policy is order optimal with respect to the edge server computation constraint (κ) and the service fetch cost (M) for the setting considered. This is one of the key results of this work. There is a gap between the performance of RR and the bound on the optimal policy with respect to the rent cost c.

While Theorem 1 gives a worst-case guarantee on the performance of the RR policy, in our subsequent analytical and simulation results, we observe that for the request sequences considered, the performance of the RR policy is significantly closer to that of the offline optimal policy than the bound in Theorem 1 suggests.

2) Stochastic arrivals: Next we characterize the performance of the RR and RR$_u$ policies for i.i.d. stochastic arrivals (Assumption 1, Section II). Recall that, under Assumption 1, in each time-slot, the number of request arrivals is x with probability p_x for $x = 0, 1, \cdots$.

Our next theorem charaterizes the performance of our policies for stochastic arrivals.

Theorem 3: Let $\nu = \mathbb{E}[X_t]$, $\mu = \mathbb{E}[\min\{X_t, \kappa\}]$ and the function f and g are defined as follows

$$f(\kappa,\lambda,M,\mu,c) = (M+\mu)\Bigg(\Big\lceil\frac{\lambda M}{\mu-c}\Big\rceil \frac{\exp\left(-2\frac{(\mu-c)^2}{\kappa^2}\frac{M}{c}\right)}{1-\exp\left(-2\frac{(\mu-c)^2}{\kappa^2}\right)} + \exp\left(-2\frac{(\lambda-1)^2M(\mu-c)}{\lambda\kappa^2}\right)\Bigg), \text{ and}$$

$$g(\kappa,\lambda,M,\mu,c) = (c+M)\Bigg(2\Big\lceil\frac{\lambda M}{c-\mu}\Big\rceil \frac{\exp\left(-2\frac{(c-\mu)^2}{\kappa^2}\frac{M}{\kappa-c}\right)}{1-\exp\left(-2\frac{(c-\mu)^2}{\kappa^2}\right)} + \exp\left(-2\frac{(\lambda-1)^2(c-\mu)M}{\lambda\kappa^2}\right)\Bigg).$$

Recall the definition of $\sigma_T^{\mathcal{P}}$ given in (3).

– Case $\mu > c$:

$$\sigma^{\text{RR}}(T) \le \min_{\lambda>1}\left(1 - \frac{\lceil\frac{\lambda M}{\mu-c}\rceil\left(\frac{M+c+\nu}{c+\nu-\mu}+1\right)}{T} + \frac{T-\lceil\frac{\lambda M}{\mu-c}\rceil}{T(c+\nu-\mu)} f(\kappa,\lambda,M,\mu,c)\right),$$

$$\sigma^{\text{RR}_u}(T) \le \min_{1<\lambda<\frac{(\mu-c)u}{M}}\left(1 - \frac{\lceil\frac{\lambda M}{\mu-c}\rceil\left(\frac{M+c+\nu}{c+\nu-\mu}+1\right)}{T} + \frac{T-\lceil\frac{\lambda M}{\mu-c}\rceil}{T(c+\nu-\mu)} f(\kappa,\lambda,M,\mu,c)\right).$$

– Case $\mu < c$:

$$\sigma^{\text{RR}}(T) \le \min_{\lambda>1}\left(1 - \frac{\lceil\frac{\lambda M}{c-\mu}\rceil\left(\frac{M+c+\nu}{\nu}+1\right)}{T} + \frac{T-\lceil\frac{\lambda M}{c-\mu}\rceil}{T\nu} g(\kappa,\lambda,M,\mu,c)\right),$$

$$\sigma^{\text{RR}_u}(T) \le \min_{1<\lambda<\frac{(c-\mu)u}{M}}\left(1 - \frac{\lceil\frac{\lambda M}{c-\mu}\rceil\left(\frac{M+c+\nu}{\nu}+1\right)}{T} + \frac{T-\lceil\frac{\lambda M}{c-\mu}\rceil}{T\nu} g(\kappa,\lambda,M,\mu,c)\right).$$

Remark 4: The bounds obtained Theorem 3 hold for all i.i.d. stochastic arrival processes. We note that the bounds worsen as κ increases. This is a consequence of using Hoeffding's inequality to bound the probability of certain events. It is important to note that significantly tighter bounds can be obtained for specific i.i.d. processes by using the Chernoff bound instead of Hoeffding's inequality. In the next section, via simulations, we show that the performance of RR and its variants does not worsen as κ increases. We thus conclude that the deterioration of the performance guarantees with increase in κ is a consequence of the analytical tools used and not fundamental to RR and its variants.

We use Theorem 3 to conclude that for T large enough, the bound on the ratio of the expected cost incurred by RR/RR$_u$ in T time-slots to that of the optimal online policy (OPT-ON) in the same time interval decays as M increases. Lemma 4 in [27] is an intermediate result which characterizes the difference between the expected cost incurred in a time-slot by our policies and the optimal online policy. From the lemma we conclude that for t large enough, the difference between the cost incurred by RR/RR$_u$ and the optimal online policy in time-slot t, decays exponentially with M and $|\mu - c|$.

Often, policies designed with the objective of minimizing the competitive ratio tend to perform poorly on average in typical stochastic settings. Similarly, polices designed for specific stochastic arrival processes can have poor competitive ratios if they perform poorly for certain 'corner case' arrival sequences. The performance guarantees for RR obtained in this section show that RR performs well in both the adversarial and the i.i.d. stochastic setting. This is a noteworthy feature of the RR policy.

C. Performance of TTL

In this section, we focus on the TTL policy which is widely used and studied in the classical caching literature. TTL serves as a benchmark to compare the performance of RR and RR$_u$.

The TTL policy fetches and caches the service whenever there is a miss, i.e., the service is requested but is not cached on the edge server. There is a timer associated with the fetch, which is set to a fixed value (L) right after the service is fetched. If the service is not requested before the timer expires, the service is evicted from the cache. If a request arrives while the service the cached, the timer is reset to its initial value of L. Refer to Algorithm 2 for a formal definition.

Algorithm 2: TTL Policy

```
1  Input: Request arrival sequence $x_t,\ t > 0$, TTL value
     $L$
2  Output: Caching Strategy $r_t,\ t > 0$
3  Initialize: Caching variable $r_1 = 0$ and timer = 0
4  for each time-slot $t$ do
5      if $x_t = 0$ then
6          if timer = 0 then
7              $r_{t+1} = 0$
8          else
9              $r_{t+1} = 1$, timer = timer − 1
10         end
11     else
12         $r_{t+1} = 1$, timer = $L$
13     end
14 end
```

Our next result provides a lower bound on the competitive ratio of the TTL policy.

Theorem 4: Let ρ^{TTL} be the competitive ratio of the TTL policy with TTL value L as defined in (2). Then,

$$\rho^{\text{TTL}} \geq \begin{cases} 1 + Lc + M & \text{if } 1 \leq \kappa < M + c, \\ \dfrac{\kappa + Lc + M}{c + \min\{Lc, M\}} & \text{otherwise.} \end{cases}$$

The key takeaway from Theorem 4 is that unlike the RR policy, the performance of the TTL policy deteriorates as the fetch cost (M) increases. This is a consequence of the fact that the TTL policy fetches and caches the service on a miss irrespective of the value of M, whereas for high values of M, RR and the optimal offline policy might choose not to fetch the service at all. Note that the performance of both RR and TTL deteriorates with increase in κ.

Next, we characterize the performance of TTL for i.i.d. stochastic arrivals.

Theorem 5: Let $\nu = \mathbb{E}[X_t]$ and $\mu = \mathbb{E}[\min\{X_t, \kappa\}]$. Recall the definition of $\sigma_T^{\mathcal{P}}$ given in (3). If $\mu < c$,

$$\sigma^{\text{TTL}}(T) \geq \left(1 - \frac{1}{T}\right) \frac{\min\{(1-p_0)(p_0 M + c - \mu), c - \mu\}}{\nu}.$$

From Theorem 5, we conclude that for low request arrival rates ($\mu < c$), the performance of TTL deteriorates with increases in M. A lemma used to prove this theorem show that for low arrivals rates, the difference between the cost incurred by TTL and the optimal online policy in time-slot t, increases with M and $c - \mu$. Contrary to this, the performance of RR and RR$_u$ approaches the performance of the optimal online policy as M increases (Theorem 3). We thus conclude that for low arrival rates and high fetch cost, TTL is sub-optimal.

TTL policies perform well in content caching, where on a miss, the requested content is fetched from the back-end server by all policies including TTL. However, as discussed above, on a miss in our service caching setting, there are two options: *(a) request forwarding* which forwards the request to the back-end server for service; and *(b) service fetch* which fetches all the data and code needed for running the service from the back-end server and caches it on the edge server. The cost for these two actions is different. By definition, TTL always takes the second option, whereas, for low request arrival rates and high fetch cost, RR, its variants and the optimal online and offline policies use the first option. This explains the poor performance of TTL for service caching.

IV. Simulation Results

In this section, we compare the performance of various caching policies via simulations. From our analytical results, we know that the Fixed TTL policy performs poorly for our setting. Therefore, we compare our policy with an online variant of the TTL policy proposed in [12]. The simulation parameters are provided in the figure captions.

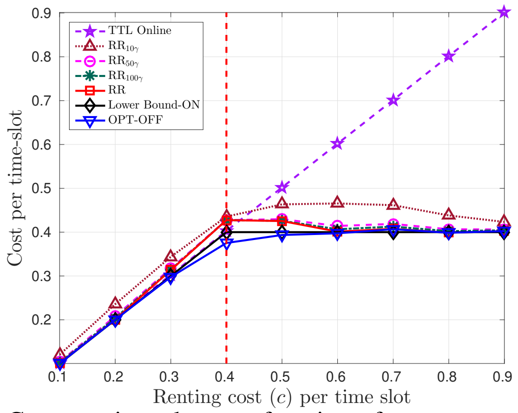

Fig. 1: Cost per time-slot as a function of storage cost (c) for $M = 4$ for i.i.d. Bernoulli(p) arrivals with $p = 0.4$

A. Stochastic Arrivals

For the first set of simulations, we focus on arrival processes satisfying Assumption 1. We compare the performance of RR, RR$_u$, TTL online, and the optimal offline policy. In addition to these, we plot the lower bound on the cost incurred by any online policy (Lemma 17 in [27]). Each data-point in the plots is averaged over 10000 requests.

1) i.i.d. Bernoulli Arrivals: In Figures 1-3, we consider Bernoulli arrivals with parameter p, i.e., $X_t = 1$ with probability p and $X_t = 0$ otherwise. Recall that $\kappa \geq 1$ and $\mu = \mathbb{E}[\min\{X_t, \kappa\}]$. Since $X_t \leq 1$ in this case, therefore, $\mu = p$. We fix $\kappa = 1$ and define $\gamma = \max\left\{\frac{M}{\kappa - c}, \frac{M}{c}\right\}$. We compare the performance of RR, RR$_{10\gamma}$, RR$_{50\gamma}$, RR$_{100\gamma}$ with the optimal offline and TTL online policies. The gap between RR$_u$ and RR decreases with increase in u. The performance of the RR policy is quite close to the lower bound on online policies for all parameter values considered. The performance gap between the optimal offline policy and the RR policy is very small compared to the bound on competitive ratio obtained in Theorem 1. We see that the gap between the performance of the RR and optimal online policy increases as M and/or $|\mu - c|$ decrease. This can be explained as follows. If $\mu < c$, the optimal online policy does not fetch/store the service and forwards all the requests to the back-end server. However, for small values of $c - \mu$, and M, the condition the RR policy checks to fetch and cache the service (Step 8 in Algorithm 1) is not very unlikely. This leads to multiple fetch–store–evict cycles and therefore a higher cost than the optimal online policy. As M and/or $c - \mu$ increase, this event becomes less probable. The case when $\mu > c$ can be argued along similar lines.

2) i.i.d. Poisson Arrivals: In Figure 4, we consider the case where the arrival process is Poisson with parameter λ. We vary κ and define $\gamma = \max\left\{\frac{M}{\kappa - c}, \frac{M}{c}\right\}$. We see that the performance of all policies improves with increase in κ. The performance of RR is very close that of the optimal offline policy and the lower bound on online policies. As before, the gap between RR$_u$ and RR decreases with increase in u.

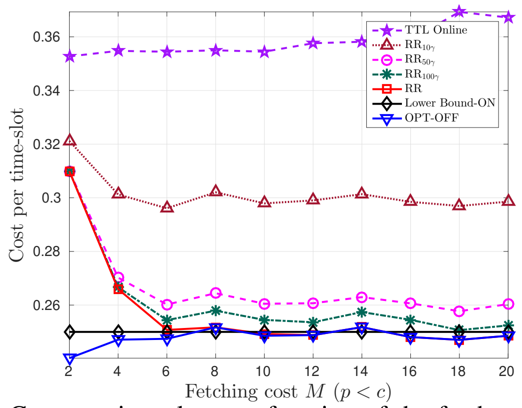

Fig. 2: Cost per time-slot as a function of the fetch cost (M) for i.i.d. Bernoulli(p) arrivals with $p = 0.25$ and $c = 0.35$

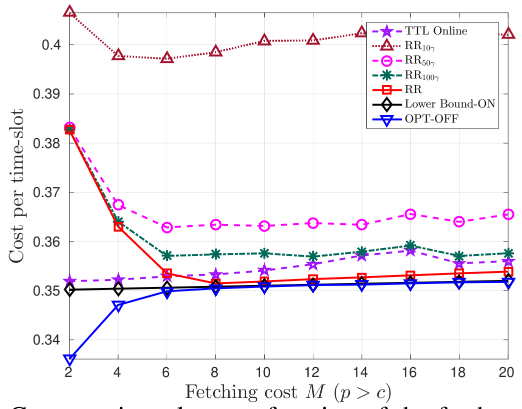

Fig. 3: Cost per time-slot as a function of the fetch cost (M) for i.i.d. Bernoulli(p) arrivals with $p = 0.35$ and $c = 0.25$

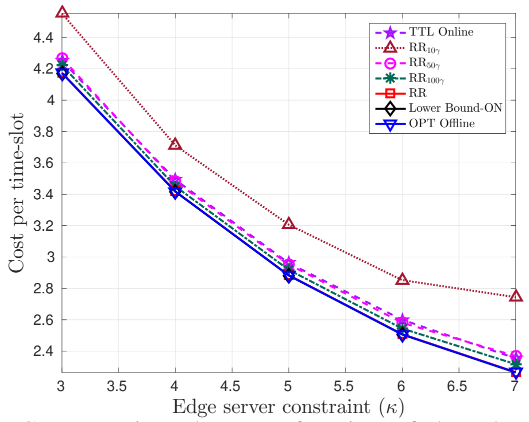

Fig. 4: Cost per time-slot as a function of the edge server constraint (κ) for i.i.d. Poisson arrivals with parameter $\lambda = 5$, $M = 10$, and $c = 2$

B. Trace-driven Simulations

For the next set of simulations, we use trace-data obtained from a Google Cluster [28]. We use a time-slot duration small enough to ensure that there is at most one request in a time-slot. This trace-data has requests for four types of jobs/services identified as "Job 0", "Job 1", "Job 2", and "Job 3". In this section, we present results for "Job 2" (Figures 5 and 6). Results for the other jobs are qualitatively similar.

For this set of simulations, we fix $\kappa = 1$ and $\gamma = \max\left\{\frac{M}{\kappa - c}, \frac{M}{c}\right\}$. We compare the performance of RR and its variants with the optimal offline policy and TTL online. The performance of TTL online is the worst among these polices, and the performance gap between TTL online and RR is significant. We note that the performance gap between RR and $RR_{10\gamma}$, $RR_{50\gamma}$ is significant whereas performance of the $RR_{100\gamma}$ policy is very close to that or the RR policy. We also note that the performance gap between the RR and optimal offline policy is significantly lower than the the worst case bound obtained in Theorem 1. For example, in Figure 6, for $M = 2$, by Theorem 1, $\rho^{\text{RR}} \leq 3.7$, while the ratio of the costs in simulations is 1.2074.

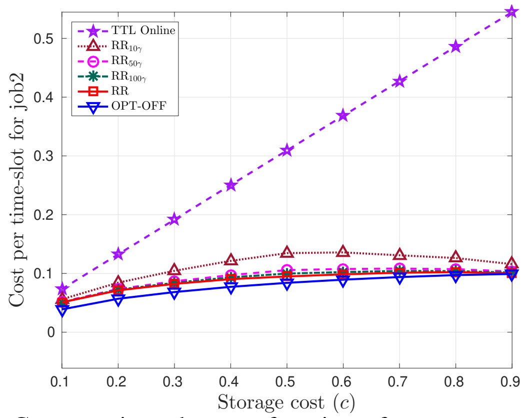

Fig. 5: Cost per time-slot as a function of storage cost (c) for $M = 10$ for Job 2

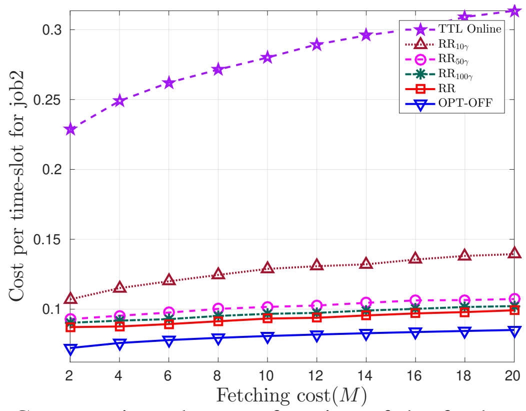

Fig. 6: Cost per time-slot as a function of the fetch cost (M) for $c = 0.45$ for Job 2

V. Proof Outline for Theorem I

We divide time into frames such that Frame i for $i \in \mathbb{Z}^+$ starts when OPT-OFF downloads the service for the i^{th} time. We refer to the time interval before the beginning of the first frame as Frame 0. Note that, in all frames, except maybe the last frame, there is exactly one eviction by OPT-OFF.

We use the properties of RR and OPT-OFF to show that each frame in which OPT-OFF evicts the service has the following structure (Figure 7). RR fetches and evicts the service exactly once each, RR does not cache the service at the beginning of the frame, the fetch by RR in Frame i is before OPT-OFF evicts the service in Frame i, and the eviction by RR in Frame i is after OPT-OFF evicts the service in Frame i. We note that

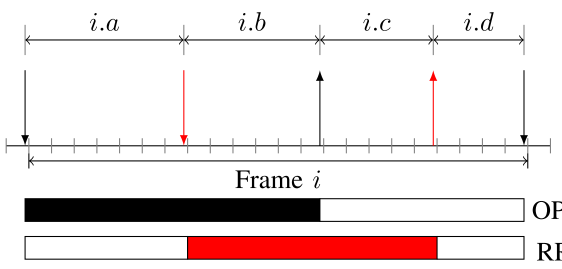

Fig. 7: Illustration of the i^{th} frame. Downward/upward arrows represent fetches/evictions. Black and red arrows correspond to the OPT-OFF and RR policy respectively. The two bars below the timeline indicate the state of the cache under OPT-OFF and RR. The solid black and solid red portions represent the intervals during with OPT-OFF and RR cache the service respectively

both RR and OPT-OFF fetch exactly once in a frame and therefore, the fetch cost under RR and OPT-OFF is identical for both policies.

We divide Frame i into four sub-frames (i.a, i.b., i.c. and i.d.) as shown in Figure 7 and upper bound the difference between the service and rent costs incurred by RR and OPT-OFF in each sub-frame. The cost incurred by RR and OPT-OFF in Frame 0 is equal. We then focus on the last frame. If OPT-OFF does evict the service in the last frame, the analysis is identical to that of the previous frame. Else, we upper bound the ratio of the cost incurred by RR and cost incurred by OPT-OFF in the frame. The final result then follows from stitching together the results obtained for individual frames.

VI. Conclusions

In this work, we focus on designing online strategies for service caching on edge computing platforms. We show that the widely used and studied TTL policies do not perform well in this setting. In addition, we propose an online caching policy named RR and its variants. Via analysis for adversarial and i.i.d. stochastic arrivals and simulations for synthetic and trace-based arrival processes, we show that RR and its variants perform well for a wide array of request arrival processes.

VII. Acknowledgements

This work was supported in part by a SERB grant on “Content Caching and Delivery over Wireless Networks” and seed grants from IIT Bombay.

References

[1] N. Fernando, S. W. Loke, and W. Rahayu, “Mobile cloud computing: A survey,” *Future generation computer systems*, vol. 29, no. 1, pp. 84–106, 2013.

[2] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[3] 2019, aWS: `https://aws.amazon.com`.

[4] 2019, azure: `https://azure.microsoft.com`.

[5] L. Chen and J. Xu, “Collaborative service caching for edge computing in dense small cell networks,” *arXiv preprint arXiv:1709.08662*, 2017.

[6] D. Willis, A. Dasgupta, and S. Banerjee, “Paradrop: a multi-tenant platform to dynamically install third party services on wireless gateways,” in *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*. ACM, 2014, pp. 43–48.

[7] T. X. Tran, K. Chan, and D. Pompili, “COSTA: cost-aware service caching and task offloading assignment in mobile-edge computing,” in *16th Annual IEEE International Conference on Sensing, Communication, and Networking, SECON 2019, Boston, MA, USA, June 10-13, 2019*, 2019, pp. 1–9. [Online]. Available: https://doi.org/10.1109/SAHCN.2019.8824854

[8] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, “A comprehensive survey on fog computing: State-of-the-art and research challenges,” *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 416–464, 2017.

[9] S. Borst, V. Gupta, and A. Walid, “Distributed caching algorithms for content distribution networks,” Mar. 2010, pp. 1478–1486.

[10] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, “You can teach elephants to dance: agile vm handoff for edge computing,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 12.

[11] 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 Journal on Selected Areas in Communications*, vol. 36, no. 8, pp. 1857–1870, 2018.

[12] D. Carra, G. Neglia, and P. Michiardi, “Ttl-based cloud caches,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 685–693.

[13] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, “Fog computing for the internet of things: A survey,” *ACM Trans. Internet Technol.*, vol. 19, no. 2, pp. 18:1–18:41, Apr. 2019. [Online]. Available: http://doi.acm.org/10.1145/3301443

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

[15] P. Mach and Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[16] S. Pasteris, S. Wang, M. Herbster, and T. He, “Service placement with provable guarantees in heterogeneous edge computing systems,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 514–522.

[17] S. Bi, L. Huang, and Y.-J. A. Zhang, “Joint optimization of service caching placement and computation offloading in mobile edge computing system,” *arXiv preprint arXiv:1906.00711*, 2019.

[18] L. Yang, J. Cao, G. Liang, and X. Han, “Cost aware service placement and load dispatching in mobile cloud systems,” *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1440–1452, 2015.

[19] J. Xu, L. Chen, and P. Zhou, “Joint service caching and task offloading for mobile edge computing in dense networks,” in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 207–215.

[20] L. Chen and J. Xu, “Budget-constrained edge service provisioning with demand estimation via bandit learning,” *arXiv preprint arXiv:1903.09080*, 2019.

[21] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, “Dynamic service migration in mobile edge-clouds,” in *2015 IFIP Networking Conference (IFIP Networking)*. IEEE, 2015, pp. 1–9.

[22] B. Tan and L. Massoulié, “Optimal content placement for peer-to-peer video-on-demand systems,” vol. 21, no. 2, pp. 566–579, Apr. 2013. [Online]. Available: http://dx.doi.org/10.1109/TNET.2012.2208199

[23] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy, “On the scale and performance of cooperative web proxy caching,” in *Proc. ACM SOSP*, 1999, pp. 16–31.

[24] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, “Web caching and Zipf-like distributions: Evidence and implications,” Mar. 1999, pp. 126–134.

[25] D. D. Sleator and R. E. Tarjan, “Amortized efficiency of list update and paging rules,” *Communications of the ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[26] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[27] L. Narayana, S. Moharir, and N. Karamachandani, “Retrorenting: An online policy for service caching at the edge,” *arXiv preprint arXiv:1912.11300*, 2019.

[28] J. L. Hellerstein, “Google cluster data: Google ai blog,” 2010.