Adaptive Replica Selection in Mobile Edge Environments

João Dias, João A. Silva, and Hervé Paulino(✉)

NOVA Laboratory for Computer Science and Informatics (NOVA LINCS), Departament of Computer Science, NOVA School of Science and Technology, NOVA University Lisbon, Caparica, Portugal
{jpm.dias,jaa.silva}@campus.fct.unl.pt, herve.paulino@fct.unl.pt

Abstract. Mobile Edge Computing (MEC) is a paradigm that aims to bring cloud services closer to mobile clients, effectively reducing latency and saving backbone bandwidth. As in cloud environments, many applications make use of replication to enhance their quality of service. However, here, data generated by the mobile devices is usually kept near its source, and can have multiple replicas scattered through the network (e.g., on the mobile devices or on edge servers). When requesting data, replica selection can have a significant impact in multiple aspects of a system, e.g., load balancing, throughput, or energy efficiency. Thus, the possible herd behavior combined with the unreliable wireless communication channels can cause systems to under-perform. In this paper, we propose Mecerra, a replica ranking algorithm tailored for the characteristics of MEC environments. Additionally, we detail Wasabi, a flexible replica ranking framework that also handles the management of system metrics. We implement Mecerra in Wasabi, and integrate it into a data storage system for edge networks, building an adaptive replica selection scheme. We use the resulting system to evaluate our proposal and compare it against related work. Results show that Mecerra is able to greatly increase the probability of finding the best replica, and Wasabi provides low overhead.

Keywords: Replica selection · Replica ranking · Mobile edge computing · Replication

1 Introduction

Mobile Edge Computing (MEC) [1] brings cloud services closer to the mobile clients, i.e., to the *edge* of the network, by leveraging on the storage and computing resources of small servers deployed in base stations. By being closer to

This work was partially supported by *Fundação para a Ciência e a Tecnologia* through project *DeDuCe* (PTDC/CCI-COM/32166/2017) and the *NOVA LINCS* research center (UIDB/04516/2020).

T. Hara and H. Yamaguchi (Eds.): MobiQuitous 2021, LNICST 419, pp. 243–263, 2022.
https://doi.org/10.1007/978-3-030-94822-1_14

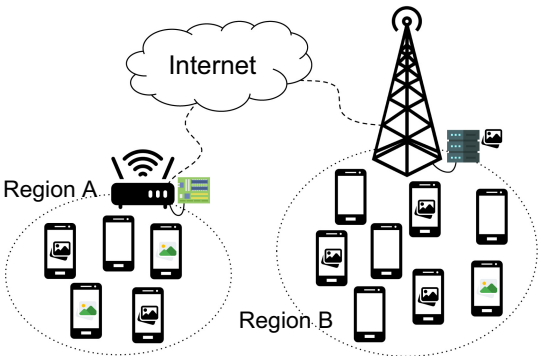

Fig. 1. Example MEC scenario. The two different pictures depicted in the phones represent replicas of two data items. Hence, both items have two replicas on region A, and one of them has four replicas on region B (including one on the edge server) while the other item has only one.

the data sources, MEC servers effectively reduce end-to-end delays and save backbone bandwidth for those cases that strictly require remote cloud infrastructures. This paradigm promotes the reduction of network bottlenecks, and allows the support of new applications with strict latency requirements. Additionally, by residing at the network edge, MEC servers are capable of collecting real-time network data, like congestion rates or subscriber mobility [2]. Furthermore, there is also the proposal of using the mobile devices themselves as actual edge nodes, harnessing their resources [13].

Both in MEC and cloud environments, many applications and services use data replication as a way to improve performance and overall quality of service. With replication, data items can have multiple replicas scattered among several nodes of the system, depending on the used replica management strategy. Consider the scenario of Fig. 1, on which a data sharing MEC application leverages on the publish-subscribe interaction pattern to enable data sharing (such as photos) among co-located users. As times evolves, the shared data items are downloaded by the users, spawning new replicas (on the mobile devices), and may also be cached by the MEC servers. In this context, when a node is notified that a new data item (matching one of its subscriptions) has been published, it is faced with the decision of selecting from which replica to download the item from. This decision can have a significant impact in multiple aspects of a system—such as load balancing, throughput, or energy efficiency—both on the server and client-sides. Aspects that can be even more exacerbated when referring to mobile devices and MEC scenarios. Thus, the question arises as how to decide which replica to contact for each data item request.

Replica ranking and selection has been addressed in cloud environments [3, 15,16], but these solutions do not take into account the idiosyncrasies of MEC systems, such as the high churn or energy constraints of mobile devices. In turn, data storage systems for MEC environments do not employ adaptive replica selection mechanisms. Also, most of the solutions do not take into account the

evolution of the system, using static strategies [10,13,14]. To the best of our knowledge, the only exception is MobiTribe [17], which centralizes that decision on a proxy server that has the knowledge of the entire system (and of the ongoing requests)—a solution that is not very resilient and difficult to apply in the decentralized MEC scenarios that we target.

In this paper, we propose Mecerra, a replica ranking algorithm specifically tailored for MEC environments, that addresses the challenges raised in these environments, namely churn, dynamic replica set, energy constraints, and metric freshness. Additionally, we detail Wasabi, a flexible replica ranking framework, that handles both the management of system metrics and replica ranking according to those metrics. Wasabi forms a continuous feedback loop between clients and servers in order to grant clients with a fresh (albeit usually partial) view of the system. Using server-emitted, as well as client-observed metrics as parameters to the underlying replica ranking algorithm, clients are able to sort the available replicas for a given request. We implement Mecerra in Wasabi, and build an adaptive replica selection scheme, by integrating this into a data storage and dissemination system for edge networks [13]. We use the resulting system to evaluate our proposal and compare it against other algorithms in related work. Experimental results through simulation show that Mecerra finds the best replica much often than the alternatives, and Wasabi provides low overhead (that can be further configured).

In sum, the contributions of this paper are the following: 1) Mecerra, a replica ranking algorithm tailored for MEC environments (Sect. 3); 2) Wasabi, a flexible replica ranking framework (Sect. 4); and 3) the evaluation of our prototype in simulation, comparing it with other replica ranking algorithms (Sect. 5).

2 Related Work

Nowadays, data is usually replicated and distributed across servers for availability, performance and scalability. Consequently, several replica servers might be available to answer a given request. In this context, the overall performance of a system that employs replication techniques is greatly dependant of two challenges: replica *placement* and *selection*. Replica placement is the problem of placing duplicate copies of a data item in the most appropriate nodes. In turn, replica selection is the problem of selecting the best replica of a required data item, at a given instant in time. In this paper our focus is directed to the replica selection problem in the context of MEC systems.

2.1 Replica Management in MEC Systems

Replicated storage is one of the building blocks of edge computing systems. Replication in this context might happen at the level of the edge server, at the mobile client level, or even at both. However, replica selection research in the field [5–7,11] has focused almost exclusively on choosing the most appropriate edge server, among several that cache a given replicated data item, primarily stored in some cloud storage system.

We are addressing a different problem. We are considering systems that preferably keep the data on the edge, allowing for replicas to exist both on edge servers and on the mobile devices. In this context, some proposals employ *full replication* (e.g., TOTA [8]), but the majority use *partial replication*. TOTA and GHT [10] rely solely on their *active replication* policy to get data across the system. However, systems like Ephesus [14] or Thyme GardenBed [13] have an *active replication* policy for new data, but then use other nodes requesting the content as *passive* replicas. Then, in order to select a replica for data retrieval, systems like GHT and Ephesus delegate to the Distributed Hash Table (DHT) logic. In turn, Thyme GardenBed always prefers the edge server when available, otherwise falls back to the DHT.

As far as we know, MobiTribe [17] is the only system with a dynamic strategy for this, but is achieved by centralizing all the decision power onto the (edge) server which acts as a proxy to redirect data requests.

In the end, none of these systems employ a sophisticated replica selection mechanism that takes into account network, load, or device-specific metrics to make an isolated informed decision of which replica should be contacted.

2.2 Replica Selection in Cloud Environments

Replica ranking algorithms for cloud environments can be classified into three categories [3]. **Information-agnostic** algorithms pick a replica in an uninformed way, not taking into account any extra information or external metrics (e.g., random or round-robin strategies). Next, **client-independent** algorithms take into account metrics independently measured by the client, without any aid from the servers. Lastly, **feedback** algorithms build on top of *client-independent* algorithms by adding *piggybacked* information with the returned values from the server, which means that both clients and servers form a feedback system.

C3 [16] is a *feedback* algorithm that combines two mechanisms in order to carefully manage tail latencies in a distributed system: (i) a load-balancing, replica ranking scheme that is informed by a continuous stream of in-band feedback about a server's load, and (ii) a distributed rate control and back-pressure mechanism. With replica ranking, clients individually rank servers according to a scoring function, with the scores serving as a proxy for the latency to expect from the corresponding server. Servers piggyback information about their queue size and approximate service time on each response to a client, and clients maintain a weighted moving average of these metrics. There is also a *concurrency compensation* that is calculated to account for both the existence of other clients in the system and the number of requests that are potentially in flight. If the *concurrency compensation* is not taken into account for the estimation of each server's queue-size, replica selection gets prone to herd behaviors. The number of requests that a client has pending over a given server also weights on the server's score. It was also decided to penalize scores over queue sizes using a non-linear function. This is because for a given server A with a service time n times faster than server B, such server would be able to get the same score as server B while holding a queue n times longer. If the service time of A then increases due to an

unpredictable event such as a garbage collection pause, all requests in its queue would incur higher waiting times.

The purpose of the rate control and back-pressure module is to ensure that the combined demand of all clients on a single server remains within that server's capacity. Although such component might have a positive impact on data-center environments [16], where all nodes are continuously engaged in high throughput/high bandwidth operations, for systems composed of energy-constrained nodes that will not be the case. Here, we argue that it might even hurt latencies if a low *sending rate* threshold was to be configured where we could easily hit the quota on all replicas and incur in unnecessary waiting times imposed by the client itself. For these reasons, we decided to discard the rate control and back-pressure component.

L2 [3] is a *client-independent* algorithm which is simpler than *C3* but can achieve a similar performance in terms of tail latency. It gives consideration to both the selection of the fastest replica server and the load balance among replica servers. Thus, the authors conclude that the intricate rate control mechanism of C3 itself is not helpful to reduce the tail latency. Even though it can achieve a similar performance comparatively to C3 in cloud environments, MEC environments are in contrast more volatile, and present additional challenges. Here, communication channels are less reliable and the available replicas are constantly varying. Therefore, we argue that it is best to use a *feedback* algorithm which provides extra contextual information on the replicas to support their decisions.

NetRS [15] tries to overcome the pitfalls of other replica ranking algorithms, where each client is responsible for picking a replica on its own, by centralizing the selection power on programmable network devices. By doing this, it can achieve a more accurate load distribution of requests and completely avoid herd behaviour. Although this might make sense in a cloud-based environment, it is not suitable for MEC environments.

Other works have built upon C3. Of these we highlight On-Off [4] and TAP [19] that address replica selection under poor timeliness conditions. On-Off improves C3's rate control by replacing the latter's cubic-function-based approach by one that associates two states to replicas: ON, by default, and OFF, for a short period of time whenever the replica is considered bad. In turn, TAP replaces C3's use of Exponentially Weighted Moving Averages (EWMAs) in queue-size estimation by a prediction algorithm that takes into consideration the trend of the servers' queue-size variation over time.

3 Replica Ranking in MEC Environments

From our analysis of existing replica ranking algorithms, C3 [16] presents itself as the most well-suited solution for MEC environments. It provides enough information for understanding the servers' resource occupation, and the network conditions, while also providing intrinsic load-balancing and the ability to avoid herd behavior. However, there are critical MEC-related challenges that are not addressed, namely churn, energy efficiency, metric freshness and the volatility of

the replica set (i.e., the number of replicas is not known a priori and may vary in time, due to churn). In this section, we elaborate on how these challenges may be addressed in the MEC context, and present the foundations of an algorithm that handles them: MECERRA.

Churn. Contrarily to cloud systems, MEC systems are deployed in highly volatile environments, where nodes often experience movement, effectively shifting their physical location, and use a wireless communication medium that might degrade with physical distance and obstacles (and other external interferences). With mobility, two outcomes are possible: i) the nodes move within range of the beacon for the wireless medium (e.g. the Access Point (AP)), or ii) they cross the range boundary, effectively leaving or rejoining the system. Regarding i), C3 provides the means to detect node movement since a server's network latency should be directly influenced by their distance to the connected network device. This means that the clients' perceived response time will increase or decrease when the server gets closer or further to the beacon, respectively and directly impacting the computed score.

However, this might not be enough to cope with ii). When a node leaves the system, the outgoing requests targeting it will be left hanging. To react to such situation, we can leverage the error conditions, and make use of predefined reply timeout values. On a timeout, we can simply stop tracking the request and use the timeout's value as the perceived response time, which will penalize the replica's score as intended. An alternative is to use the *outgoing requests count* to that node, which will potentially never decrease, and hence reduce the chances of the node being picked again as a designated replica. Without extra bookkeeping and logic, this penalty would, however, be perpetual and keep penalizing the node if it ever rejoins the system. Given this, we advocate for the timeout alternative (maybe with some dynamic setting policy). It is thus important for MEC systems to have a request timeout policy in place to react and adapt to node churn.

Dynamic Replica Set. Due to churn and node movement, more often than usual, a client might be faced with the situation where it has unknown nodes within a list of replicas to rank. In such case, several approaches are possible: i) be pessimistic and always consider these replicas last; ii) be optimistic and favour them over the remainder; iii) or try to be impartial and prefer them over *bad* replicas, but only after the *good* ones.

To define the quality of a replica, we require scores to be values in a closed interval, and classify as *bad* and *good* the replicas whose score is, respectively, lower or higher than the interval's medium value, referred to as the *neutral score*.

Energy Awareness. Mobile devices are limited by their battery capacity, which may differ considerable among devices. It is also common for nodes to join the system with only part of their total battery capacity and the remaining

capacity be shared by several applications. Being of the system's interest to keep the maximum number of nodes online for as long as possible, the replica ranking algorithm should take this metric into account and score replicas according to their current battery capacity. Furthermore, it is important to observe the battery's evolution over time [9,18]. Batteries drop at different rates and some might even be increasing (e.g., when connected to some power source, like a powerbank).

Metric Freshness. Freshness (or timeliness) relates to the time elapsed from the instant a value is last registered (for a given metric) and its use in the ranking algorithm. It is directly bound to the ability of a node to perceive the system's actual state. Hence, the older the value, the less trustworthy it likely is.

The impact of freshness is higher on the server-side metrics, as these report a server's state at a given instant in time. Although some server metrics, such a battery state, may be accurately modelled from the client-side, others, such as *pending request queue size*, cannot. The same holds for the client-side metrics, e.g. a client always knows the exact value of the *outstanding requests* on a given server, but the perceived end-to-end latency (i.e., *response time*) depends of external conditions that cannot be accurately estimated beforehand. Thus, we see freshness as a concern that must be individually evaluated for each metric. This allows for a better fine-tuning of the replica ranking process and accommodates the possible asynchronous reading or computing of the metrics (resulting in some metrics being older than others).

To better represent the values of discrete metrics, a simple but effective approach (also used by C3) are EWMAs. Concerning continuous metrics that can be estimated on the client-side, we make use of decay functions.

3.1 The MECERRA Ranking Algorithm

Having discussed the challenges raised by MEC, we now move to the presentation of our proposal: the MECERRA algorithm.

Building on C3, we evaluate a replica considering its internal state (*queue size* and *service time*), on how much demand we already have over that server (*outstanding requests*), and on the *response time*. We additionally consider the replicas' *battery capacity*. The outstanding requests and last reported battery value are registered as absolute values, while the remainder metrics are stored as averages (i.e., EWMAs).

To rank a set of replicas, we need to compute their individual score. For that purpose, the set of metrics stored about a node is retrieved and passed to function REPLICASCORE of Algorithm 1. If any metric essential to compute a score is missing, the replica cannot be (yet) classified and, hence, the function falls back to the neutral score (line 24). Otherwise, the expected latency is computed in a way similar to C3 (lines 13 to 15) by computing the value for the following formula:

$$lat = (rt - st) + \hat{qs} \cdot st$$

Algorithm 1. MEC Enhanced Replica Ranking Algorithm.

```
1: MAX_LATENCY                ▷ Maximum latency assumed for communication
2: BATTERY_WEIGHT             ▷ Weight of the battery metric wrt the remainder
3: PENALTY_PERCENTAGE         ▷ Penalty assigned to nodes when communication fails
4: numberNodes                ▷ Estimate of the number of nodes currently in the system

5: function ReplicaScore(metrics, penalty)
6:     if hasEssentialMetrics(metrics) then
7:         qs ← metrics.get('queueSize')
8:         st ← metrics.get('serviceTime')
9:         or ← metrics.get('outstandingRequests')
10:        rt ← metrics.get('responseTime')
11:        (bat^rec, ts^rec) ← metrics.get('battery')
12:        b̂at ← batest(bat^rec, ts^rec)          ▷ battery capacity estimate
13:        cc ← or × numberNodes                  ▷ concurrency compensation
14:        q̂s ← (1 + cc + qs)^3                   ▷ queue size estimate
15:        lat ← (rt − st) + (q̂s × st)            ▷ expected latency
16:        ls ← (MAX_LATENCY − lat) × 100/MAX_LATENCY   ▷ latency score
17:        wls ← (1 − BATTERY_WEIGHT) × ls        ▷ weighted latency score
18:        s ← BATTERY_WEIGHT × bat^rec + wls     ▷ score with recorded data
19:        ŝ ← BATTERY_WEIGHT × b̂at + wls         ▷ score with battery estimate
20:        if ŝ > NEUTRAL_SCORE ∧ s < NEUTRAL_SCORE then
21:            return NEUTRAL_SCORE × penalty
22:        else
23:            return ŝ × penalty
24:    return NEUTRAL_SCORE × penalty

25: procedure OnTimeout(node, requestId, timeout_value)
26:    requests ← requestMap[node]
27:    requests ← requests \ {requestId}
28:    recordResponseTime(node, timeout_value)
29:    penaltyMap[node] ← PENALTY_PERCENTAGE
```

where rt, st and $\hat{qs}$ are, respectively, the replica's observed response time, service time and queue-size estimation.

Due to the volatility of the environment, the concurrency compensation factor (cc in line 13) takes into consideration an estimate of the number of nodes currently in the system. This value may change in-between but never during a replica raking process, otherwise it would render scores non-comparable.

The ReplicaScore function also takes into consideration the last recorded battery value (bat^{rec}) and an estimate of its current value ($\hat{bat}$), obtained in line 12 by calling function batest (defined ahead). From these two values, two scores are computed, s and $\hat{s}$, each calculated from the expected latency and correspondent battery value, converted to the same scale and weighted according to a previously defined relative importance (lines 16 to 19). The score computed from the estimated value is considered whenever both scores are above or below

the neutral score. Otherwise, if the prediction causes the replica to cross the *good*-to-*bad* replica boundary, the neutral score is used instead (lines 20 to 22). This approach guarantees that replicas from which nothing is known won't be chosen over replicas previously considered *good*, but now estimated to have become *bad*.

The final score may be subjected to a *penalty* factor that penalizes replicas to which the latest communication attempt failed. This penalty is set on timeouts (procedure OnTimeout of Algorithm 1), along with the removal of the request from the outstanding request list and the record of the response time, which will trigger the computing of the associated EWMA (lines 28 and 29). Note that the penalty is also applied in the neutral score branch to avoid repeatedly choosing yet unknown replicas to which communication is failing.

Battery Capacity Estimation. Being the battery a resource that is continuously being consumed, a battery value bat^{rec} recorded at instant ts^{rec} may not represent battery's value at the current instant ts^{now}. To estimate the current capacity of a battery when it is draining (the last recorded value is lower than the previous) we use the following decay function:

$$\text{BATEST}(bat^{rec}, ts^{rec}) = bat^{rec} - (ts^{now} - ts^{rec}) \cdot \frac{\theta}{\epsilon} \cdot \nu \cdot \beta$$

where ϵ denotes the time elapsed since the first message exchanged between the client and the target node, θ denotes how many bytes have been transferred between the target node and the client (upstream and downstream) since they first connected, ν represents an estimate of how many nodes are currently in the system, and β is a pre-defined value for the cost of battery consumption for every byte sent or received.

To the measured value, the function subtracts an estimate of the energy spent from ts^{rec} until now. To that end, we multiply the measurement's age by the *byte per millisecond* rate (θ/ϵ), to represent a rough estimation of how many bytes have been transferred, in average, between the target nodes and the remainder. The result is then multiplied by the number of nodes to account for the remainder nodes in the system and obtain some sort of *concurrency compensation*. The final factor (β) accounts for the average energy needed to communicate a byte in the used communication medium.

3.2 Handling Popularity Bursts on Under-Replicated Data

In some situations, there may be the case of a sudden demand for content that still has not been replicated across the system. An example of such scenario is when someone publishes something to an over-subscribed topic on a publish/subscribe system (i.e., a very popular topic). Because that item has just been published, there are no replicas. However, as the system sends out notifications for all subscribers, the publishing party will be flooded with all the data requests.

Unfortunately, replica ranking algorithms are not the solution here since there are no replicas to select from. To solve this problem, we came up with a redirection mechanism to leverage the emerging replicas from this initial request.

The problem with having a single entity serving all data requests is that: i) some of these requests will incur in long waiting queues, potentially resulting on a timeout on the client side; and ii) if we are serving the contents from mobile nodes, then these will have their battery drained faster.

Our solution consists of having server nodes register which data items they have served to whom recently, allowing for them to present a list of alternative replicas to any request that would otherwise have to wait for others to finish. By taking this *fail fast* approach, we avoid making the client wait for something that might not arrive. On its end, the client can decide whether to take one of the suggested alternatives or pick some other replica it already knows. If there are no alternatives, however, we still put the request on a queue. With this, servers are capable of better distributing their load with other emerging replicas, potentially preserving both their battery life and the system's liveness.

4 The WASABI Framework

In order for a client to make an informed decision on which is the best server (or replica) to contact when requesting a data item, it needs to collect information about the existing servers in the system. Thus, we present WASABI, a modular and extensible replica ranking framework. It handles the collection and management of system metrics, as well as replica ranking according to the collected metrics.

To allow the implementation of a wide range of replica ranking algorithms, WASABI enables the collection of metrics from both the client and server sides. It is also parametric, allowing the configuration and implementation of the majority of its components.

4.1 Overview

Figure 2 depicts a general overview of the WASABI framework. To be flexible, and cover scenarios both with functionally symmetric or asymmetric systems, the framework is divided into independent server and client modules, each of which can coexist in any node of the system (i.e., any node can be a server, a client, or both at the same time).

WASABI can be seen as a middleware that is embedded in an application and sits above its network communication layer. Overall, the server module collects system metrics, and aggregates them into Metrics bundles on demand. Thus, when a message is about to be sent on the server side, the application queries the server module for a Metrics bundle and piggybacks it in the message, before sending it. When a message is received on the client side, the application extracts the piggybacked metrics from the message, before sending it up the stack. At this moment, the client can also take the opportunity to measure some pertinent

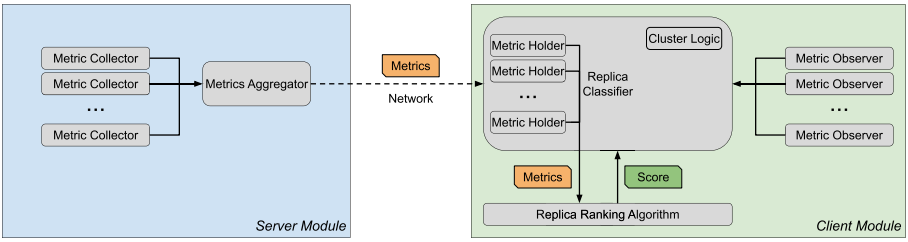

Fig. 2. WASABI architecture overview.

metric. Then, both the extracted and the measured metrics are recorded by the client module, associating them with the corresponding server. In turn, when a client is about to send a request for a data item, it first contacts the client module asking for a list of replicas ordered from best to worst, and then decides which to contact. The client module makes use of the previously collected metrics to sort a known set of replicas according to a predefined ranking algorithm.

In the framework, a Metric is represented by a double-precision floating point value, identified by a system-wide generic label (e.g., a string). Meanwhile, a Metrics bundle is simply a set of metrics concerning a server, represented as a map of labels to metric values.

4.2 Server Module

The framework's server-side module consists of only two different components: one *Metric Collector* for each metric the application wants to keep track, which in turn are registered in the *Metrics Aggregator*.

Metric Collector. This component contains the necessary logic to read and/or compute a system metric. It employs a *pull-based* strategy, whereby it needs to be registered in the framework, which in turn polls this component for metric values. Typically, it is a *functional* component, i.e., it performs stateless computations to produce a value. Thus, it suffices to implement its interface, which contains a single method: `collect(): double`. For other scenarios where stateful computations might be required, we provide a *Sample Metric Collector*, that stores metric samples and returns a value based on those samples when queried. As an example of a typical use case, we provide the *Average Metric Collector* that computes the average of the stored metric samples.

Metrics Aggregator. To be able to aggregate their values, collectors have to register in the *Metrics Aggregator*, mapping collectors to their labels. Thus, the main function of this component is to aggregate the registered *Metric Collectors*, for producing Metrics bundle on demand. When queried, through method `getMetrics(): Metrics`, this component polls all the registered collectors for their values, and returns a bundle to be sent to the client.

4.3 Client Module

The framework's client-side module consists of several components. The Replica Classifier receives metrics from servers, and keeps each one in a Metric Holder. When queried, the Replica Classifier passes the stored metrics to the Replica Ranking Algorithm which returns a list with the replicas sorted by descending score. Metric Observers allow the collection of metrics from the client-side.

Metric Observer. It works similarly to a Metric Collector on the server-side, by encapsulating the logic for reading and/or computing a system metric. However, while collectors are pull-based, observers employ a *push-based* strategy, not requiring to register within the framework. Instead, they push their readings into the Replica Classifier. This push model is required to update the stored information regarding a given replica on system events, such as network activity. A typical example is the measurement of the server response time, for which we record the request identifier with a timestamp, and when the response is received, measure the elapsed time. Additionally to this Response Time Observer, we also provide the Outstanding Requests Observer, to track how many outstanding requests there are to a server.

Replica Ranking Algorithm. This component encapsulates the ranking algorithm, i.e., it computes a score given a set of metrics over a replica. It requires the implementation of the method `score(m: Metrics): double`, which is also responsible for handling missing metrics.

Metric Holder. This is simply a container for a metric's current value, allowing to set a different retention policy for each metric registered on the Replica Classifier, i.e., we can keep just a discrete value, or something more complex such as a moving average. We offer a variety of Metric Holders, namely a record, a counter, and an exponential moving average.

Replica Classifier. This is the central component of the client module, responsible for classifying the available replicas for a given request. It does so by taking a set of replicas and sorting them by descending order of the score given by the Replica Ranking Algorithm. Naturally, the accuracy of that ordering is proportional to the amount of collected metrics over that set of replicas and of its freshness. All metrics required by the Replica Ranking Algorithm should be registered in the classifier at bootstrap time. However, the Replica Classifier does not hold metrics directly, but rather stores them within Metric Holders, indexed by server. As metrics are received and stored in their corresponding containers, unrecognized metrics are ignored. Metrics can also be registered together with a decay function. When scoring a replica, this decay function considers the elapsed time since the last recorded value for the given metric and updates the value accordingly.

Cluster Logic. Because some systems might have the need to tell apart clusters of nodes from individual nodes (e.g., cluster-based DHTs), we introduce the (optional) Cluster Logic component. It requires the implementation of two methods: a predicate which determines whether a given replica represents a cluster or not; and a method which retrieves the cluster replicas from the known replica set on the Replica Classifier. When classifying, a cluster gets the average score of all its composing (known) replicas.

5 Experimental Results

In order to evaluate our solution we integrated it with Thyme GardenBed [12, 13], a time-aware reactive data storage and dissemination system for MEC environments that offers a time-aware topic-based publish/subscribe interaction model. Mobile devices may publish content and subscribe to their topics of interest. Subscriptions are bound to time intervals that may encompass the past, the present, and the future, matching with all publications (to the same topic) performed in the subscribed time interval. A successful match triggers notifications to the interested devices (also referred to as *nodes*). The notification does not carry the data item itself, but a rather small description (such as a photo thumbnail) and the list of known replicas to where download the item from, among other information. If a node chooses to download the item, it then becomes itself a new replica, in a *passive* replication mechanism. Moreover, edge servers (one per region) periodically inspect the region and retrieve the region's most popular data items, according to a pre-defined popularity-based ranking algorithm. Thus, among other things, the edge servers also cache replicas of some of the items published in their region (see Fig. 1). Lastly, the system may also be configured to actively replicate data items. This configuration may be set on a per-item granularity and the replicas are stored in the mobile devices organized in a cluster-based DHT. In sum, when a node receives a list of replicas, this list may contain passive and active replicas hosted by mobile devices, as well as a replica cached by the edge server.

In this context, our evaluation seeks to answer the following questions: 1) how good is our replica ranking algorithm? 2) how much overhead does the underlying feedback system introduces and does it pose as a decent trade-off? and 3) does the redirection mechanism presented in Sect. 3.2 helps in improving the load balance in some extreme cases?

5.1 Experimental Setup

To evaluate our solution we resort to the emulation of the mobile devices. This emulation environment allows us to perform experiments with a large number of devices, as well as better control the operations performed by each device, without having to develop a new implementation of Wasabi. The code running on the experiments is exactly the same that runs of the Android devices.

The emulator is built on a trace-based framework that accepts a trace file containing the operations we want each node to perform and when. The framework runs on a single process, emulating each mobile device in a separate thread (and from here, each device can use as many threads as necessary). Furthermore, the emulator replaces the Network Layer to support logical dissemination of messages between any number of virtual nodes. This allows us to simulate network conditions and control communication in a predictive manner. Each operation within the trace maps to an *action*: the logical representation of the operation. Each kind of action has a mapped *behavior* that represents the effect the action should cause in the system.

To support our simulated scenario, we made use of a computational cluster. We ran the mobile device emulator in one node and a (non-simulated) edge server in a second one. The communication between emulator and edge server was made through the network connection link, while the communication between nodes was made through the logical layer provided by the emulator.

5.2 Replica Selection

To assess whether or not we consistently pick the best replica, we need to know two things for each download (i.e., data request): the chosen replica and the actual best replica. To know which replica was chosen is just a matter of recording it. However, to know which would have otherwise been the best choice requires a bit more effort. For this, we use an *oracle*.

The *oracle* is an external component that needs to be fed with the whole system information to be able to answer any question with the highest degree of certainty. It is composed of two parts: i) an extra persistent logging component that is enabled within the system nodes to record their state and downloads information; and ii) a post-processing script which computes metrics (e.g., how good was the replica selection on each specific download) from the data previously collected at runtime.

Using the recorded system snapshots, which contain the most up-to-date metrics of each node and their stored contents at each moment, we can use the *oracle* to compute the optimal replica system-wide for each node's decision. With this, we can now answer our first question: *how good is our replica ranking algorithm?*

For this test, we defined the following baselines: **Random Selection** - the client sorts the available replicas in an arbitrary fashion; **Edge First** - the client always picks the edge server whenever it is available; **C3** - the cloud envisioned implementation of C3; and **Mecerra** - our proposed replica ranking algorithm.

To properly evaluate the effects of our solution, we decided to create a scenario with lots of subscriptions and publications. Our trace is divided in four parts: 1) spawn 64 nodes and let them join the system (only after all the nodes are online and ready do we proceed); 2) all nodes have a 50% chance of subscribing to each of the available topics. This will cause some topics to be more subscribed than others, which will cause some published items to be popular later on; 3) after all the subscriptions comes a barrage of publications. As before, each node

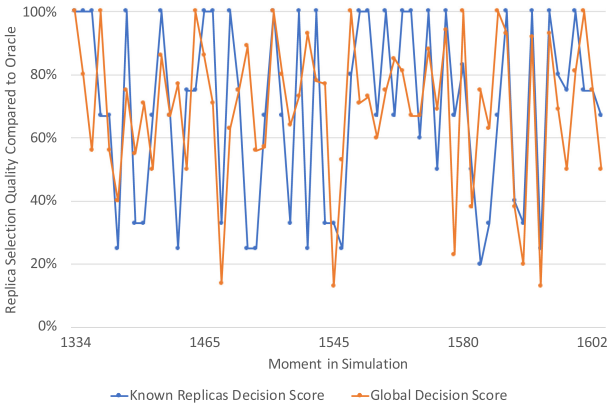

Fig. 3. Replica selection benchmark (Random Selection).

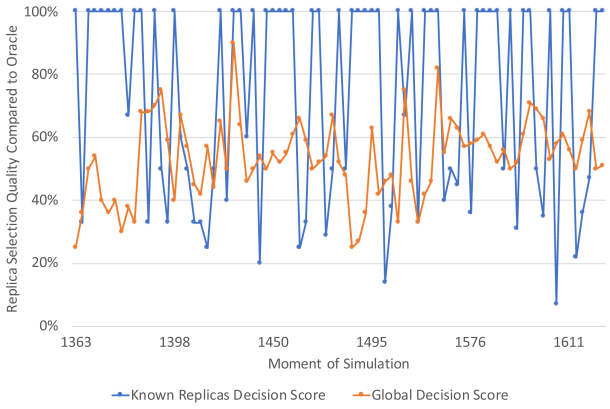

Fig. 4. Replica selection benchmark (Edge First).

has a 50% chance of making a publication on each topic. This will trigger several notifications on each publication, which in turn will trigger the same number of downloads; 4) all nodes have a 70% chance to subscribe to the topics they have not subscribed before. These subscriptions, however, are spanning to the past, meaning that each of them will pick all the items published to the same topic on the previous step. In this phase we will continuously have a high volume of downloads in the system which means that smart replica ranking and selection might play a big part in load balancing and resource management. We use the same trace to run simulations for each of the previously outlined baselines.

Random Selection is the most inconsistent baseline. Figure 3 shows its replica selection quality, having the blue plot represent how good the selection was for a given download considering the available metrics on the client at that moment (this can also be seen as a direct comparison to Mecerra); and the orange plot represents how good the selection was considering the actual state of the

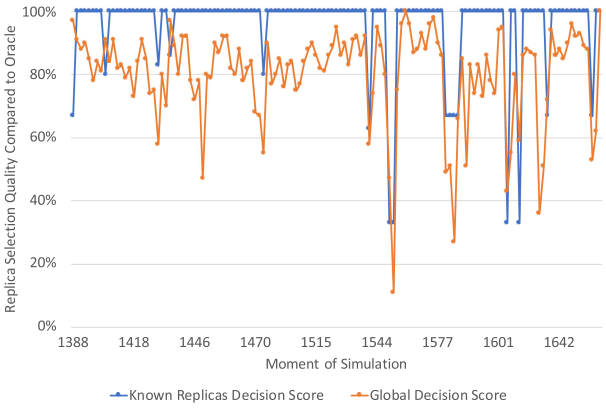

Fig. 5. Replica selection benchmark (C3).

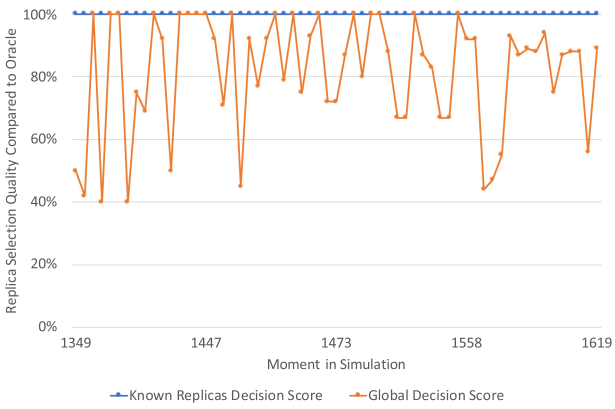

Fig. 6. Replica selection benchmark (MECERRA).

whole system at the same moment, as provided by the *oracle*. Here, the average selection quality is 69% and 68%, respectively. This is only because this has no logical selection criteria and is thus not affected by the asynchrony between stored metrics and actual system state. We can argue that it is not a bad result and that it even takes the best available replica for some of the downloads. But the fact is that these were all scenarios where the edge server was an option and was picked by chance. In fact, most downloads only consider a tiny part of all the existing replicas–40% on average–which is about 3.6 known replicas for each download. Given that we only considered downloads with 3 or more available replicas at the client, there was a high number of downloads where the client only had 3 options, amongst them being the edge server which was the actual best option most of the times. Other than that, we rarely see any download with a selection score over 80% and there are more below 40% than on any other baseline. It is also worth noting that the trace executions for the

Random Selection always yielded less total downloads than any of the others, meaning that some download requests incurred in big waiting queues.

Edge First was the previous strategy of Thyme GardenBed, and consists of always preferring the edge server when it's available, otherwise falling back to *Random Selection*. Just like *Random Selection*, it is not making use of any metrics available on the client, but as we can see from Fig. 4 the selected replica frequently matches the best option the client could compute using the available information (blue plot). This is because even though our underlying framework does not distinguish between edge and mobile servers, the edge server is always reporting 100% battery and the expected latencies are very reasonable. However, considering the actual state of the system on each download, we can see that from our sample, the edge server was **never** the best option. This is because most nodes were concurrently downloading files from it, creating a hot-spot which, in the face of edge server delays or network congestion, can compromise system liveness. The average selection quality is 73% and 53%, respectively. The 20% gap between how good the selected replica was according to the available metrics of the client versus how good it was considering the actual snapshot of the system at that given moment is explained by the fact that each client only had a partial view of the system (on average only 40% of the existing replicas were known) and some of the available metrics were not the freshest.

C3 was our highlighted cloud algorithm. From Fig. 5 we see a positive increase in the replica selection quality compared to the previous baselines, scoring an average of around 90% when considering the clients' local metrics and 80% considering the global system state. Although it is a cloud algorithm, we can see the benefit of an adaptive ranking scheme fed by a feedback loop between servers and clients. In spite of the improvement, there were still some less optimal selections, namely the ones below 40% quality. These are explained by the fact that *C3* does not cover the several challenges that arise from transitioning from the cloud to the MEC environment, such as considering servers' battery levels or ranking replicas it has not seen before, even preferring replicas it knows perform badly over new ones.

Mecerra was made specifically for MEC environments, building on top of our findings from the cloud and covering all the new concerns. Looking at Fig. 6 we can see clear improvements from any other baseline. First, we look at the selection quality from the clients' local metrics which is consistently at 100%. This simply means that the ranking algorithm is correct since it is always producing the same result as the oracle, considering the exact same set of replica metrics. The most interesting part is indeed the selection quality considering the global system state. Here, we had many more 100% selections than any other baseline. Moreover, most selections stayed around 80% to 100% while the remaining choices did not drop below 40%. On average, the replica selection quality was 82%. The average increase is not as steep as it was from *Edge First* to *C3*, which can be attributed to the few selections that still scored below 60%. This, however, can only be increased if we can improve the feedback system since the worst selections were caused by the fact that, like on the other benchmarks,

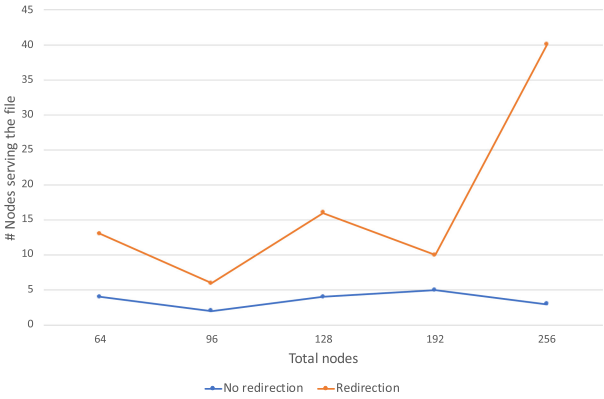

Fig. 7. How many nodes served the file.

we only knew around 40% of the available replicas on each download and on some cases some of the metrics stored on the client side did not accurately reflect the target replica state anymore.

The bottom-line is that with a feedback system the quality of the selected replicas is directly proportional to the amount and freshness of the available information.

5.3 Framework Overhead

We now evaluate the increase in byte count introduced by adding metrics the each system message. To do this, we ran a similar trace to the previous and registered all the bytes sent by each node. After, we aggregate all these values to have a total amount of bytes transferred during the simulation. We ran the simulation for our version of Thyme GardenBed and then repeated the process with a version without Wasabi. In the end, we compared the two values.

With this comparison, we observed an increase of about $10\% \times w_r$, where w_r denotes the ratio of messages carrying Wasabi information. On average, each metric payload increased the size of each message by about 50 bytes. At this size, the metrics payload was smaller than any system message. So, in the case of Thyme GardenBed, adding Wasabi with Mecerra yields much greater improvements (around 29% increase) than the maximum overhead (10% more bytes transferred for $w_r = 1$). This is a considerable improvement which will reflect in the system's resource management and liveness.

5.4 Redirection Mechanism

To evaluate the redirection mechanism presented in Sect. 3.2, we designed a trace where all nodes subscribe to a topic and then one of them publishes a (big) file to that topic, causing all others to download the file. Each node sends its download request with 1 s distance from the previous one and each download operation

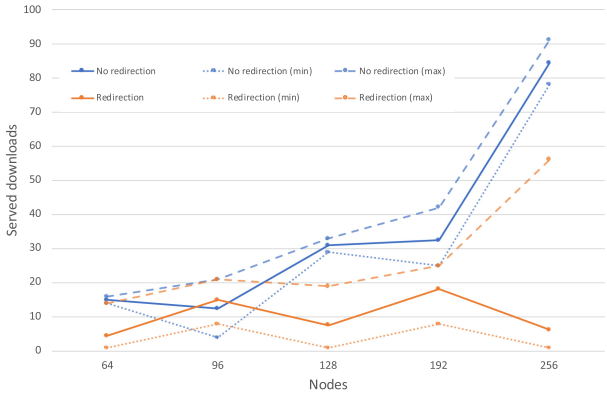

Fig. 8. Downloads served by serving nodes.

takes 5 s to finish. We also configured 8 s timeouts. We have then created 5 variants of this trace with 64, 96, 128, 192 and 256 nodes and ran each variant with and without the redirection mechanism, to understand the difference in load distribution. Also, because Thyme GardenBed groups the mobile nodes on clusters, we tried to guarantee a 1:4 ratio of clusters to nodes which means that, on average, a cluster should be comprised of 4 nodes.

Indeed, from Fig. 7 we can see that the redirection mechanism effectively enables more nodes to serve the file that is on demand, decentralizing the load from the publishing party. When no redirection was in place, the maximum number of nodes participating in distributing the requested file were at most 5, which is the approximate number of nodes in a cluster. If redirection was enabled though, this number increased on every experiment, showing a considerable growth on the 256 nodes mark.

Additionally, Fig. 8 shows the average downloads each serving node served on each simulation. Here we confirm that each serving node serves a lot more requests when there is no redirection than with the alternative. With redirection, the average of downloads served by a single node never crossed 20 whereas without redirection it went over 80. Both the maximum and minimum plots followed the average plot on both scenarios, with the exception of the maximum number of downloads registered for the simulation with 256 nodes and redirection enabled, where a single node served 56 downloads. This, however, is still considerably less than the 91 registered for its counterpart.

Regarding timeouts, there was only a negligible amount in the simulations with no redirections and higher node counts.

6 Conclusion

In this paper, we presented Mecerra, a replica ranking algorithm that directly addresses the inherent challenges of MEC environments, which, to our knowledge, no other replica ranking or replica selection algorithm does. We also

detailed Wasabi, a flexible and lightweight replica ranking framework on top of which we implement Mecerra. Finally, we integrate this ensemble into a data storage and dissemination system for edge networks, building an adaptive replica selection scheme.

To validate our solution we resorted to simulation. From our experimental results, we can see that Mecerra is able to outperform *C3* in MEC environments, ranking the best available replica as #1 eight times more often. It also provides a less volatile replica selection quality, not dropping below 40%. Still, the less satisfying selections were only due to lack of available information on the client, which, on average, only knew 40% of the existing replicas. We have also measured the increase in network traffic caused by the feedback system, reaching a small overhead of roughly 10%. Finally, the redirection mechanism allows the system to take earlier advantage of newly emerging replicas, effectively sharing the load with those. With 256 nodes, there were eight times more nodes answering requests, when comparing with the system without redirection mechanism. In turn, each serving node ended up serving eighth times less requests with the redirection mechanism.

As future work, we would consider alternative communication channels. There is a wide range of wireless mediums we can leverage to circumvent problems that arise on the current one (e.g., congestion in the AP when using Wi-Fi) and ultimately provide a better quality of service.

References

1. Abbas, N., Zhang, Y., Taherkordi, A., Skeie, T.: Mobile edge computing: a survey. IEEE Internet Things J. **5**(1), 450–465 (2018). https://doi.org/10.1109/JIOT.2017.2750180
2. Beck, M.T., Werner, M., Feld, S., Schimper, T.: Mobile edge computing: a taxonomy. In: AFIN 2014: The Sixth International Conference on Advances in Future Internet, pp. 48–54. IARIA (2014)
3. Jiang, W., Xie, H., Zhou, X., Fang, L., Wang, J.: Performance analysis and improvement of replica selection algorithms for key-value stores. In: Fox, G.C. (ed.) 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, 25–30 June 2017, pp. 786–789. IEEE Computer Society (2017). https://doi.org/10.1109/CLOUD.2017.115
4. Jiang, W., Xie, H., Zhou, X., Fang, L., Wang, J.: Haste makes waste: the on-off algorithm for replica selection in key-value stores. J. Parallel Distrib. Comput. **130**, 80–90 (2019). https://doi.org/10.1016/j.jpdc.2019.03.017
5. Li, C., Tang, J., Luo, Y.: Scalable replica selection based on node service capability for improving data access performance in edge computing environment. J. Supercomput. **75**(11), 7209–7243 (2019). https://doi.org/10.1007/s11227-019-02930-6
6. Li, C., Song, M., Zhang, M., Luo, Y.: Effective replica management for improving reliability and availability in edge-cloud computing environment. J. Parallel Distrib. Comput. **143**, 107–128 (2020). https://doi.org/10.1016/j.jpdc.2020.04.012
7. Li, C., Wang, Y., Tang, H., Luo, Y.: Dynamic multi-objective optimized replica placement and migration strategies for SaaS applications in edge cloud. Future Gener. Comput. Syst. **100**, 921–937 (2019). https://doi.org/10.1016/j.future.2019.05.003

8. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), 14–17 March 2004, Orlando, FL, USA, pp. 263–276. IEEE Computer Society (2004). https://doi.org/10.1109/PERCOM.2004.1276864
9. Metri, G., Agrawal, A., Peri, R., Shi, W.: What is eating up battery life on my smartphone: a case study. In: International Conference on Energy Aware Computing, ICEAC 2012, Guzelyurt, Cyprus, 3–5 December 2012, pp. 1–6. IEEE (2012). https://doi.org/10.1109/ICEAC.2012.6471003
10. Ratnasamy, S., et al.: GHT: a geographic hash table for data-centric storage. In: Raghavendra, C.S., Sivalingam, K.M. (eds.) Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications, WSNA 2002, Atlanta, Georgia, USA, 28 September 2002, pp. 78–87. ACM (2002). https://doi.org/10.1145/570738.570750
11. Shao, Z., Huang, C., Li, H.: Replica selection and placement techniques on the IoT and edge computing: a deep study. Wirel. Networks **27**(7), 5039–5055 (2021). https://doi.org/10.1007/s11276-021-02793-x
12. Silva, J.A., Cerqueira, F., Paulino, H., Lourenço, J.M., Leitão, J., Preguiça, N.M.: It's about thyme: on the design and implementation of a time-aware reactive storage system for pervasive edge computing environments. Future Gener. Comput. Syst. **118**, 14–36 (2021). https://doi.org/10.1016/j.future.2020.12.008
13. Silva, J.A., Vieira, P., Paulino, H.: Data storage and sharing for mobile devices in multi-region edge networks. In: 21st IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks", WoWMoM 2020, Cork, Ireland, 31 August–3 September 2020, pp. 40–49. IEEE (2020). https://doi.org/10.1109/WoWMoM49955.2020.00021
14. Silva, J.A., Monteiro, R., Paulino, H., Lourenço, J.M.: Ephemeral data storage for networks of hand-held devices. In: IEEE Trustcom/BigDataSE/ISPA, pp. 1106–1113. IEEE (2016). https://doi.org/10.1109/TrustCom.2016.0182
15. Su, Y., Feng, D., Hua, Y., Shi, Z., Zhu, T.: NetRS: cutting response latency in distributed key-value stores with in-network replica selection. In: 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, 2–6 July 2018, pp. 143–153. IEEE Computer Society (2018). https://doi.org/10.1109/ICDCS.2018.00024
16. Suresh, P.L., Canini, M., Schmid, S., Feldmann, A.: C3: cutting tail latency in cloud data stores via adaptive replica selection. In: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015, Oakland, CA, USA, 4–6 May 2015, pp. 513–527. USENIX Association (2015). https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh
17. Thilakarathna, K., Petander, H., Mestre, J., Seneviratne, A.: MobiTribe: cost efficient distributed user generated content sharing on smartphones. IEEE Trans. Mob. Comput. **13**(9), 2058–2070 (2014). https://doi.org/10.1109/TMC.2013.89
18. Vallina-Rodriguez, N., Hui, P., Crowcroft, J., Rice, A.C.: Exhausting battery statistics: understanding the energy demands on mobile handsets. In: Cox, L.P., Wolman, A. (eds.) Proceedings of the 2ndt ACM SIGCOMM Workshop on Networking, Systems, and Applications for Mobile Handhelds, MobiHeld 2010, New Delhi, India, 30 August 2010, pp. 9–14. ACM (2010). https://doi.org/10.1145/1851322.1851327
19. Zhou, X., Fang, L., Xie, H., Jiang, W.: TAP: timeliness-aware predication-based replica selection algorithm for key-value stores. Concurr. Comput. Pract. Exp. **31**(17) (2019). https://doi.org/10.1002/cpe.5171