Replica Placement in Edge Computing

Leonardo Marques Epifânio
leonardo.epifanio@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Edge computing is defined as a paradigm in which servers are placed close to the edge of the network, in order to assist applications that running in resource-constrained devices. There are two main advantages of edge computing: firstly, edge nodes can provide assistance with much lower latency than the cloud, because servers are physically closer to the devices; and secondly, edge nodes can shield the cloud from most requests, by serving the requests locally. Edge nodes can help in the collection of information from the devices, by aggregating information from multiple devices before sending it to the cloud. Edge nodes can also make information available to local devices, that can then access this information with low latency. In this work we are mainly interested in the latter, and we study techniques that can be used to place replicas of relevant data on edge devices. We propose a system that can collect estimates of future data demand from different sources (such as historical data or current observed access patterns) and that can make data placement decision based on these demand estimates, taking also into consideration the costs of maintaining data replicas and the benefits that can be achieved by maintaining those replicas. We will illustrate the operation of our system using examples from the area of vehicular networks, where edge nodes can help vehicles to select the best traffic routes based on up-to-date information regarding road congestion, accidents, or other hazards.

1 Introduction

Edge computing is defined as a paradigm in which servers are placed close to the edge of the network, in order to assist applications that running in resource-constrained devices [1]. There are two main advantages of edge computing [2]: firstly, edge nodes can provide assistance with much lower latency than the cloud, because servers are physically closer to the devices; and secondly, edge nodes can shield the cloud from most requests, by serving the requests locally.

An application area where edge computing may prove helpful is the area of vehicular networks. Current vehicles have multiple sensors that can capture information not only about the operation of the vehicle itself (such as fuel consumption), but also about the environment (temperature, road condition, etc). If this data can be effectively collected, it can prove itself useful to many parties: the information can be used by makers to schedule vehicle maintenance and to improve vehicle design, city planners can use the information to design better road systems, etc. Furthermore, vehicles can also be consumers of this information: for instance, a vehicle can leverage the data collected by other vehicles regarding road congestion, accidents, or other hazards, to plan alternative routes [3]. In this setting, edge nodes can help in collecting information from the vehicles, by aggregating it from multiple devices in an area, before sending it to the cloud. Edge nodes can also make this information available to local vehicles, that may then access it with low latency.

In this work we address the problem of data placement for edge computing, studying several techniques that can be used to place replicas of relevant data on edge devices. We propose a system that can collect estimates of future data demand from different sources (such as historical data or current observed access patterns) and that can make replica placement decisions based on these demand estimates, taking also into consideration the costs of maintaining data replicas and the benefits that can be achieved by maintaining those replicas.

The developed system has to be scalable enough to support a high number of participants, and adaptable to dynamic changes in the data accesses. We will also account for different types of operations, and how they affect the communication between nodes, according to the consistency models they are set to support.

We will illustrate the operation of our system using examples from the area of vehicular networks, as we've described. In this case, historical information regarding the requests of vehicles on their routes, may allow to predict which information is relevant to maintain, at a given edge node, at a given time of the day. The decision of placing a replica of that information, on a given node, then needs to take into consideration its potential benefits (how likely it is that the information is going to be used, and what savings can be achieved by serving requests closer to the requesting nodes).

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. We provide a brief introduction to key concepts in Section 3, and review existing solutions in Section 4. Section 5 describes the proposed architecture to be implemented and Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of data placement in edge networks. More precisely:

> *Goals:* We plan to design, implement, and evaluate techniques to perform automatic data placement decisions in edge computing scenarios. Each data placement decision should be made based on a utility function that considers the expected benefits of placing a replica on given edge nodes, and the costs of maintaining that replica. For estimating the benefits of each decision, we plan to use both historical and current information regarding data usage.

We plan to maintain, for each data item, information regarding both maintenance costs and access patterns. This metadata should reflect not only the current usage of the item but also historical records. Then we plan to design policies to make decisions regarding which sites should replicate which data items at a given point in time. Ideally, the policies would maximize the utility of the entire system but, given the scale of the system, that is expected to be composed of a high number of edge nodes and data items, to compute an optimal assignment would be infeasible, in practice. Thus, we will also study heuristics that can be executed in a decentralized manner. The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of the metadata that needs to be maintained to drive the data placement policies; ii) a specification of the optimization problem, regarding replica placement; iii) an algorithm to execute these policies in runtime, in a decentralized and scalable manner; iv) an implementation of the resulting system; v) an extensive experimental evaluation using simulation software.

3 Background

3.1 Edge Computing and Related Paradigms

The emergence of technologies and applications for mobile computing and the Internet of Things (IoT), has driven computation towards dispersion, as opposed to centralization. With the growing quantity of data generated at the edge of the network, in order to assess the issues of high-bandwidth, geographically-dispersed, low-latency applications, several paradigms have been proposed [1], in which the data is stored, processed, and acted upon close to these devices:

Edge Computing is one of such paradigms, where computation is located as close as one hop from IoT devices (data sources), extending their capabilities, in a ubiquitous manner.

Fog Computing also decentralizes computation and storage, but its model is hierarchical, providing progressively increasing resources along a multi-tier succession of nodes, located between the IoT devices and the cloud data centers.

Multi-access Edge Computing, known also as Mobile Edge Computing, provides resources through the Radio Access Network (RAN) infrastructure. This model is expected to benefit significantly from the, now developing, 5G platform, which will allow for edge computing to be accessible to a wide range of mobile devices, with reduced latency.

Cloudlet Computing uses computers or clusters of computers, known as cloudlets, with virtualization capabilities. Tasks are taken from mobile clients, and divided among the cloudlet nodes in their proximity, which in themselves, might also be composed of clustered mobile devices.

Despite presenting some differences, these concepts have been used interchangeably in previous literature, and demonstrate how the system models vary within the edge computing paradigm. Despite the fact that each system is, in its majority, composed of nodes that are less resourceful than the cloud datacenters, there can still be found a large variance in the storage capabilities and network reach of each node, and how these communicate with one another, according to the topology and routing algorithms. Along with these factors, there can also be seen differing characteristics from application to application, such as the amount of data involved, the frequency and locality of requests, and the consistency requirements of each operation.

Since our aim is to define a general replica placement scheme for the edge, we must take these differences into account, by using strategies that are flexible and scalable, while maintaining effectiveness.

3.2 Example of an Edge Computing Application

Vehicular Ad-hoc Networks [4] are defined as decentralized wireless networks of devices, with vehicle mobility patterns. With the increase in the number of vehicles on the road, and the advancement of vehicle and communication technologies, VANET technologies have been developed with the goal of increasing road safety and transportation efficiency. Two types of nodes are distinguished in this model: On-Board Units (OBU) and Road-Side-Units (RSU). Usually, the RSUs, being more resourceful nodes placed strategically along the vehicles' paths, represent higher tier nodes, that connect the OBUs, which are placed inside the vehicles, to the upper level of the network.

In systems where these nodes are responsible for producing data, we can very intuitively observe the applicability of an edge computing model. Such is the case in [5], where the authors propose LiveMap, an automated system for acquiring, curating, and disseminating detailed road conditions, with low latency. A simple approach would be to ship the data from the vehicles to the cloud for its analysis and storage, however, as the authors note, this would hinder the scalability of the system, as there are specific areas in cities with a large amount of traffic, in which multiple vehicles could be transmitting high quality media at the same time. Because of this, vehicle edge devices are used, along with zone cloudlets, each operating over a county-sized coverage area.

Specifically, there are several *vehicle cloudlets*, each operating in a vehicle, which collect sensor data from the road, and transform it into semantic updates of lower bandwidth, which are then transmitted to the zone cloudlet. This unit is responsible for fusing the inputs from all vehicle cloudlets, storing the data, and selectively disseminating it to the participants, which may then cache it in the vehicle cloudlet. Several data acquisition/dissemination approaches are considered, with varying data upload rates. The most notable technique is *throttle-by-cache*, in which vehicles maintain a cache of the surrounding area in the LiveMap, and when a hazard is detected, they consult their caches and only upload new observations.

The system achieves lower latency accesses and updates, in comparison with a centralized model, but since each zone cloudlet covers a full county-sized area, in cities with very high, concentrated traffic, bandwidth usage for hazard reporting may still overwhelm the zone cloudlet, negatively affecting the request latencies.

By using less resourceful and smaller coverage-area cloudlets, and allowing vehicles to query these cloudlets before reaching their respective zones, we can alleviate the aforementioned bottlenecks. To further improve performance, and to allow the effective operation of real-world applications such as trajectory planning and traffic rerouting, we may also replicate hazard information according to their requests' popularity and geographical locality, also leveraging daily traffic patterns that occur due to daily commutes or weekly events. This way vehicles in a zone z_1, going to z_2, can have the hazards of z_2 cached in z_1, even prior to fetching them.

Several replica placement techniques that have been applied to this paradigm account for the mobility of clients, and how to replicate the chunks of an item, in a way that a requesting vehicle is able to collect the full item by accessing the RSU nodes along its path [6–8]. However, these type of placement techniques are, in a way, more related to the problem of how to transmit data to mobile users, and are orthogonal to our work, since we only consider mobility as a factor that may influence the locality and dynamism of requests.

3.3 Replica Placement

Although there are several applications for replica placement, such as fault tolerance and load balancing, in this work, we investigate when and where to create item replicas, with the objective of improving user-perceived latencies, in an efficient way.

A replica placement scheme aims at achieving a set of goals, that may be described by specifying a cost function, whose minimization achieves optimal placement. These cost functions can account for several factors such as request popularity and geographical locality, distances between nodes, connection latencies, link bandwidth, storage costs, etc.

In [9], an earlier work on replica placement, the authors specify a simple cost model that accounts for access costs from nodes to replicas, but assumes homogeneous request patterns across all nodes. In this model, the goal is to find the placement strategy which minimizes the average number of hops that requests must perform, in order to reach the desired replica. The cost function of placement strategy x is defined by:

$$C(x) = \frac{1}{\Lambda} \sum_{i=1}^{I} \sum_{j=1}^{J} \lambda_i p_j d_{ij}(x)$$

in which Λ is the total request rate of all nodes, I is the set of nodes, J the set of objects, λ_i the request rate of node i, p_j the probability that object j will be requested in any node, and $d_{ij}(x)$ the shortest distance from i to a node that contains j, under placement x. This placement strategy is also subjected to a storage constraint, regarding the maximum capacity at each node.

The authors prove the NP-Completeness of deciding if there is a placement x that achieves cost value lower than a target T, by mapping the problem to the multiple knapsack problem. However, if we consider the optimization problem, that finds the solution which minimizes cost, we may conclude that, like the multiple knapsack optimization problem, solving it is NP-Hard.

Because of this property, inherent to any type of Generalized Assignment Problem, finding the optimal solution for the cost function of a placement scheme is not feasible. Due to this, each scheme defines instead an heuristic approach, that aims at approximating the solution. We will classify each approach according to:

Optimality, which defines how well the scheme approximates to the optimal solution.

Centrality, which characterizes where the decisions and computations take place. On one end, we have systems that perform these actions on a central node, which takes into account the whole of the network, on the other end, a system may have each node executing the placement algorithm in a parallel and autonomous way, with a limited view of the network, which normally results in rougher approximations to the optimal solution. Since we want the system to be able to handle a large number of nodes and data items, we aim for decentralization, in order to avoid bottlenecks, and reduce execution time via parallelism, all the while preserving optimality.

Replication degree, which can be fixed, restraining the number of replicas for each item to a predefined value, or adaptive, changing the number of replicas according to item demand. Performing placement with adaptive degrees is more complex than with fixed degrees, but achieves more optimal results, particularly in skewed workloads.

Routing, which may be classified as replica-aware or replica-blind. Replica-aware routing schemes have knowledge on the locations of replicas, directing requests according to these locations, and adapting to changes in their placement. These schemes have the advantage of reducing request latency, but may require the nodes to synchronize with each other and update their routing tables, whenever there are changes in the replica locations, also achieving lower efficiency when performing replica lookup.

Replica-blind schemes, on the other hand, route requests regardless of replication. If a request, when being propagated through its path to the original file location, reaches a node with a replica, then the request is satisfied by this node, reducing latency. Therefore, these schemes are limited to replicating items in the request paths, but don't require any synchronization, achieving high efficiency.

Efficiency, which is split into the computation and communication overheads. The computation overhead represents the complexity of computation necessary to perform a decision. These computations can be performed in parallel across multiple nodes, so the considered overhead is that of a single node.

The communication overhead represents the amount of communication between nodes that is needed not only during the execution of the placement algorithm, to send the information needed for performing the placement decisions, but also after the placement algorithm, for keeping the system up-to-date with the new replica locations.

Adaptability to dynamic workloads, which is a requirement in systems where the requested data varies in time, with regards to its locality and popularity. Since we are making a general placement scheme for edge computing applications, we aim for strong adaptability.

Convergence, which defines how long, in terms of optimization rounds, the scheme takes to converge to a stable placement solution. A stable solution is reached when the system stops optimizing, by reaching the best possible solution for the current workload, within the heuristic's capability. Slow convergence manifests itself more negatively in dynamic workloads, since a stable solution may never be reached before the workload changes.

Decision factors, which are the factors considered when performing the placement decisions. These may include, among others, the popularity of requests for each item at each node, the latency between nodes and item replicas and the storage costs of each node.

3.4 Context-Sensitive Differential Consistency

In [10] a distributed data store system for the fog paradigm is proposed, with a differential consistency mechanism, where devices have distinct consistency guarantees based on their context. The authors demonstrate the applicability of context-sensitive differential consistency, by exemplifying with a vehicular network scenario, where autonomous cars may read the state of a traffic light on a road junction, from multiple replicas, each placed in nearby cloudlets, some of which may have outdated information. Cars located farther from the road junction, might only use the traffic light status to update their estimated arrival time, and as so, have weaker consistency guarantees on the read operation, only querying one replica; while cars located closer to the road junction have stronger consistency guarantees, in order to avoid a possible accident, and must thus obtain the responses from a quorum of replicas.

This concept can be applied to any scenario where, for the same item, there can be requests with differing consistency requirements, affecting their latency and routing, and consequently, affecting the optimal strategy for replica placement. To understand this, consider an item i whose read operations may only consult one replica, but whose write operations must access all replicas. If there is a high read frequency for i coming from a set of nodes N, it may be beneficial to create more replicas close to N, reducing the perceived latencies. However, the latency of write requests will increase with the addition of each replica, due to the fact that all of them must be contacted in a single request. Therefore, if the write frequency is also high, this strategy possibly renders itself disadvantageous.

In this project, we consider the problem of placing replicas in a way that minimizes the average latency (or any other type of access cost), in a system model where requests may have different behaviours in the way their are routed through the system, not only due to the type of operation being performed, but also due to the supported consistency model in their current context.

4 Related Work

In this section we review the existing literature on replica placement, focusing on schemes that, despite considering varying system models, present viable applications to the edge paradigm. We first describe each solution according to their goals and how they operate, and then we perform a comparison of these solutions, highlighting each of their advantages and shortcomings.

4.1 Replica Placement Schemes

4.1.1 EAD File Replication [11] The Efficient and Adaptive Decentralized (EAD) file replication uses an algorithm that optimizes replica placement in a dynamic workload environment. It does so by placing replicas in the nodes that are subject to more frequent queries. The algorithm is fully distributed.

The algorithm works by running consecutive optimization rounds. In round t, each node periodically computes q_{f_t}, the query rate of each file f, whose lookups are initiated or forwarded by that node. If q_f is bigger than a threshold (that depends on the average query rate of the system), the node is considered a traffic hub for f, and a *replication request* is submitted to one of its file servers.

A file server node, which may hold the original file or a replica, aside from computing q_{f_t}, also periodically computes the visit rate of its files (query load l_t). If the load surpasses a threshold above the node's capacity, the list of replication requests received in t is ordered according to the q_f of each request, and requests are accepted until the predicted load is bellow the threshold. If this list is empty, the files are instead replicated to the neighbour nodes that most frequently query for file f. In case the file server is not overloaded, it will accept replication requests only if the benefit of file replication (considering the query rate, and the resource consumption of forwarding the query from the requesting node to the server) outweighs the storage cost. Each replica node, excluding the original file server, may also remove f, if q_f falls under a minimum load threshold during a certain number of consecutive rounds.

In order to avoid replica fluctuation and the replication overhead that may result from sudden rises in file popularity which only last for a round, an exponential moving average (EMA) technique is employed to calculate q_f, applying weighting factors to older q_f, such that their weight decreases exponentially at each round, culminating in more reasonable measurements of the query rate.

By placing replicas in the query paths from requesters to file servers, the scheme allows the usage of a DHT lookup strategy, and replica-blind deterministic routing, keeping this process highly efficient. However, since this approach restricts replication to a specific set of nodes, it results in sub-optimal placement when looking at cases where the replica could be placed closer to its requesters.

When there is a node join/leave, since the network topology changes, and traffic may follow different paths, replica nodes might, in such events, witness lower visit rates for some files. To deal with this *churn* issue, the system may transfer replicas between the neighbours of the node which has joined/left, in order to place these files in the predicted new traffic hubs of the network.

The scheme is able to dynamically adapt to nonuniform and time-varying file popularity and node interest, adaptively adjusting the replica degrees. Furthermore, it only requires synchronization in the case of node churn, leading to a low communication overhead. High replica utilization is ensured, but the optimality factor is hindered by using replica-blind routing, restricting placement to query hubs. Convergence is fast, since replicas are immediately placed in their destinations, in a parallel fashion, and also quickly eliminated, if their access rate drops.

4.1.2 Congestion-Aware Caching (CAC) [12] This work investigates cache management and routing policies that account for link capacity constraints and network congestion. The work is based on the observation that popularity-based caching may not provide the best average latency. This is due to the fact that less popular items, if not cached, may also incur a large overhead on their retrieval, in particular of retrieval requests are routed through highly congested links.

Based on this observation, the proposed scheme aims at caching content that is both popular and whose retrieval is costly, bandwidth-wise. And although this was developed as a caching solution, the system model assumes cooperation between caches, and we may interpret cached data as replicas of the original items, intuitively translating the scheme into a replica placement algorithm, with an adaptive replication degree.

It is assumed that each content delivery response follows the opposite path of the corresponding request. Each node keeps track of the number of flows (active requests) currently passing over its links, which is used for estimating the available link bandwidth for new requests. Content popularity is also estimated in each node, for every item whose requests pass through the node. These values are used by the node to calculate the utility of replicating each item, which is directly proportional its popularity, and inversely proportional to the minimum available link bandwidth values along the upstream path (from content source to current node). These last values are sent between nodes, by attaching them to requests, resulting in very little communication overhead.

Technically, no synchronization is required, since items are replicated on the request path, allowing for replica-blind routing. However a congestion-aware search protocol is proposed by the authors, in which, starting from the requester, nodes flood their neighbours with an interest request, until reaching a boundary, determined by the number of hops and available link bandwidth, to prevent further congestion. Nodes containing a replica of the item respond back to the requester, which then routes its requests to the node whose path is the least congested.

By accounting for link congestion, this scheme is able to reduce the average latency in a fully distributed way, despite reducing the cache hit rate and possibly increasing the number of routing hops. By employing the search protocol, traffic is displaced from the possibly more congested default paths, to alternative, less congested routes, also contributing to the avoidance of unnecessary replicas in those same paths. This way, even though replication is performed on the request paths, routing is aware of nearby replicas, contributing to a greater approximation to the optimal solution.

Convergence is fast, since replicas are placed immediately on their optimal destinations. However, when a replica is created in a node due to congestion, the node doesn't keep track of the available link bandwidth to the original object. In a dynamic environment, this value could eventually improve, but because it isn't tracked, a replica may be kept indefinitely in a node whether or not it is beneficial to the system, which contributes to a sub-optimal performance. This could be improved by using an age-based tactic, such as the one presented in the scheme we will discuss next.

4.1.3 Age-Based Cooperative Caching (ABC) [13] In this work the authors leverage the coupling between routing and caching, by proposing an adaptive replication mechanism that pushes popular contents to the network edge, in a fully distributed way, without requiring extensive computations, nor message exchanges between nodes. The scheme distributes the contents hierarchically, placing the most popular contents at the network edge, while the least popular are placed closer to the sources, focusing on the improvement of the average cache hit rate and load reduction at central storage units, while handling dynamic workloads.

When the original item is requested from its source server, its initial age value is calculated in the neighbour node through which it is routed back to the requester. This value reflects the lifetime of the replica, and is proportional to its associated popularity, which is the observed access frequency in the source server. If there is space left for the item, it is cached in the aforementioned neighbour node. In the opposite case, if there's an expired item, it is replaced by the arriving one.

An item is considered expired when it reached the end of its age-defined lifetime. As the item is forwarded to the requester with its age piggybacked, every node doubles the value for its age, unless it surpasses a predefined *max_age* parameter. This way, the scheme prioritizes item popularity and distance from the source nodes, with items being replicated along the path from the requester to the source, with decreasing importance, and also according to their popularity.

The clear advantage of using such a placement tactic is the efficiency, since there is no need for synchronization, just like in the previously described schemes, where replicas are also placed on the routing paths of their correspondent requests. The scheme also takes into account the distance from nodes to the original items, but the issue relies on the fact that the popularity factor in the age values is calculated and accounted only in the source nodes, which may result in inaccuracies, where items, having a global high popularity, are replicated, with an high age value, in edge nodes where they might only be requested a couple of times. This curbs optimality, and since nodes suffer from the departure of popular contents, when they reach their expiration date, a stable placement solution is never reached.

4.1.4 Associated Data Placement (ADP) [14] In this data placement scheme, the focus is on improving the co-location of associated data, while still ensuring localized data serving and balance. Two schemes are modelled, the first not allowing replication, and the second, allowing replication with fixed degrees for each data item. We focus on the study of the second scheme, being more relevant to our context.

As a precursor of this scheme, the authors in [15] presented a data placement system that iteratively moves an item closer to both its clients and other items it communicates with, but in contrast with [14], the group association problem is relaxed to pair-wise relations, and aside from this, the system uses physical distance between nodes as a decision factor, focusing on the actual geo-distribution of cloud datacenters. Thus, its application to our context is limited.

In [14], the system model assumes that geographically distributed nodes may request sets of specific replicas, of a data item, from each other. These sets are called *request patterns*. The placement decision needs global knowledge of the request patterns, and it's run in a centralized unit. In a time period, the request rate of each pattern and single replica are measured and/or predicted. These are then used to calculate 2 metrics:

Co-location of associated data, whose value is proportional to the number of replicas in each request pattern not handled by the same node, without accounting for relaying distance.

Localized Data Serving, whose value is proportional to the number of requests where the requesting node is not the one holding the data. The authors also mention an extension of this metric, by considering the logical distance from the nodes serving the requests.

The objective is to minimize a weighted sum of both of these metrics, whose weights are defined by a trade-off parameter. A constraint for balance is also applied, by setting upper and lower limits in the number of replicas stored in each node, according to average of all nodes. The placement is then formulated as a hypergraph partition problem.

For the initial replica placement, a simple greedy method is applied, in which, for each data item, the k nodes with the highest request rate are used to store the replicas of the item. After this step, the scheme runs a loop, performing, in each round, a placement decision, but also a routing decision, since the system adopts a deterministic routing function, that maps to an adequate destination, each request for an item in a pattern, from each node. This decision is made considering the request patterns, and their coverage by each node, aiming to minimize the number of nodes contacted. When a replica is migrated, the routing decisions concerning it have to be calculated again, and routing is performed by iteratively choosing the nodes that cover the highest number of items in the request pattern. This enables fetching several items from a single node, and reducing the request paths, but it is still less efficient than using a replica-blind routing scheme.

Placement is performed by building a hypergraph in which the vertex set is composed by the nodes and the replicas, and the hyperedge set contains all the request patterns, each assigned a weight proportional to its frequency, and all the pairings between each node and each replica, also weighted according to frequency. This hypergraph is then partitioned, minimizing the cut weight.

A scheme for replica migration, specific for dealing with dynamic workloads, is also proposed, in which only a maximum number K of replicas may be migrated in each round, performing an incremental adjustment. The set of replicas chosen is the set whose migration provides the highest gain to the system performance. This incurs that several rounds must go by, before reaching the stable placement solution.

However, although the scheme is resilient to dynamism, and achieves a great approximation to the optimal solution by having a global view of the network, its centralization aspect hurts scalability when we combine dynamic workloads with a high number of nodes. The computation overhead to perform the placement decision is much higher than in the previously analyzed schemes, and there is a high communication overhead due to the necessary data collection to build the hypergraph, and due to the synchronization phase after each placement decision.

4.1.5 AutoPlacer [16]

AutoPlacer is a self-tuning replica placement system, developed for a cluster environment. The system operates on the migration of replicas, considering a fixed replication degree. It assumes consistent hashing is used as the initial routing strategy, but allows replicas to be migrated to any node, placing their new locations in a relocation map. Since the map can grow too large to be efficiently distributed, the authors propose a structure to ensure fast lookups, called PAA (Probabilistic Associative Array). The scheme executes in rounds, and each optimization round consists in the following sequence:

Task 1: The top-k most-accessed data items (hotspots), at each node, are calculated. Once some hotspots have been identified (and relocated) in a given round, new hotspots are sought in the next round, since in the presence of static workloads, the top-k lists at each node may stagnate.

Task 2: Each node gathers access statistics on any hotspot items it supervises and finds the optimal placement for those items, solving a relaxed version of the cost function minimization problem.

Task 3: Each node computes the PAA for the relocated items it supervises.

Task 4: Each node disseminates its PAA among all nodes, which then assemble the received PAAs, locally building an object lookup table.

Task 5: The relocated items are transfered.

To find the location of an item's replicas, each node first queries the PAA, whose response, despite being deterministic, may contain innaccurate information but guarantees that no false negatives are provided. If the node does not find any location for an item in the PAA, this means that the replicas have the default placement, and the hash-based routing scheme can be used to find a replica. The PAA is able to reduce space by classifying keys according to available values within them (for example, a key may contain the user that created the item), and using rules that take these values from a key, and output its correspondent replica set. The accuracy of this structure is controlled by a few adjustable parameters.

To deal with dynamic workloads, the system starts a new epoch each time there is an abrupt change of the access patterns, restarting the execution of the optimization algorithm with a new lookup table. This is not optimal for a workload with a progressively changing distribution, undetected by the simple threshold verification. Likewise, choosing to only optimize k items for each node in each round impairs the celerity at which the system converges to a stable placement solution, but despite this, the system, eventually, effectively approximates to the optimal solution.

While optimization is done in a distributed fashion, the fact that every node has to synchronize with each other in each round, represents an overhead in communication. The replication degree is fixed, and hence, the replica number is not adjusted, hindering better tuning for improving request latency and replica usage. However, the system is successful at maintaining efficient lookups for a replica-aware routing scheme.

4.1.6 Distributed Cache Management in ICNs [17] This work explores several on-line distributed and autonomic cache management algorithms directed at Information-Centric Networks (ICNs), and compares them in terms of performance and complexity. The aim is to minimize the overall network traffic cost, by adaptively assigning (and re-assigning) item replicas to caches, based on their time-varying request popularity and locality.

The network is modelled as a graph of caches/nodes, connected by their communication links. It is assumed that a replica-aware routing mechanism is in place, routing requests to the nearest replica. Considering V as the set of nodes, M as the set of available information items, H as the current configuration, which maps each item to a set of nodes, with h_v^m being a variable that indicates whether item m is replicated in v, and N_v^m as the set of nodes accessing m through its replica at v, the total network traffic cost is given by:

$$T(H) = \sum_{m=1}^{M} \sum_{\substack{v \in V: \\ h_v^m = 1}} \sum_{u \in N_v^m} r_u^m d_{vu}$$

where r_u^m is the request rate for m, generated by u, and d_{vu} the communication cost from v to u. The vector r_u is an estimation based on the observed request patterns within a given time window, used to forecast future rates. The objective is to minimize this cost function, without violating each node's storage constraints, whilst maintaining, at least, one replica of each item, in the system. To achieve this, the authors propose four heuristic solutions which vary in the amount of knowledge each node needs about the network, and the required level of coordination between nodes. These algorithms are executed in rounds, and maintain execution as long as they detect possible performance gains.

The first algorithm, called *cooperative*, requires each node to maintain a global view of request patterns, and the current configuration. At each iteration, each node v calculates, for each replica m that can be removed, the global performance loss l_v^m, which is computed as the cost difference between a new configuration, in which m is removed from v, and the current configuration. For each item m that is not replicated in v, the expected global performance gain g_v^m, achieved by caching m in v, is also calculated. The node then considers the item with maximum performance gain, i, as the candidate for insertion, and the item with minimum performance loss, k, as the candidate for replacement. The local maximum relative gain, b_v, is calculated and a report message $Rep(b, v, i, k)$ is sent to the other nodes. After receiving these messages, each node obtains the network-wide lowest cost configuration, updating H accordingly, and applying its changes to the system.

The *holistic* algorithm requires no coordination of the actions of each node, which perform their decisions autonomously. It differs from the cooperative algorithm, by assuming that only a single node may modify the global configuration at a given time. This way, in each iteration, a single node v computes the maximum relative gain b, and broadcasts the report message. After receiving this message, each node immediately updates H, and the changes are applied. By noticing that more than one replacement per node, in each iteration, can be beneficial, speeding up convergence to a stable solution, the authors propose the *holistic-all* algorithm, which behaves in a similar way to the previous one, but allows the node v to perform all the beneficial replacements, by selecting, from all possible items in M, the set of candidates that minimize the total traffic cost.

Finally, the *myopic* algorithm assumes that each v has no information about the demand patterns at the other nodes, performing decisions based solely on local information. Considering u_m as the nearest node from v that caches m, the objective function for this algorithm becomes:

$$T_v(H) = \sum_{m=1}^{M} r_v^m d_{u_m v},$$

The candidates are then selected only according to $d_{u_m v}$, which represents a very limited view of the network, leading to more sub-optimal results than the other approaches.

All of the algorithms adapt to dynamic workloads, however, myopic, due to its limited view, has nodes competing for replicas, leading to slow convergence. The cooperative and holistic-all algorithms don't have this problem, but only perform placement on a limited number of objects, which also hinders convergence. The advantage of the myopic approach is clearly the lower computation and communication overhead, with no information collection, while all the other algorithms operate on a global view of the system's item demands. The cooperative algorithm also specifically requires the communication between all nodes after computing the candidates in each of them, resulting in lower efficiency.

4.1.7 Decentralized Replica Placement (D-ReP) [18] This work proposes a fully distributed dynamic algorithm for replica placement in the paradigm of edge computing, taking into account the latency between nodes and their storage costs, considering these as IaaS providers. It aims at improving the proximity of requesters to replicas, in a bandwidth- and cost-effective way.

The cost function is defined as the number of requests for each replica, multiplied by the latencies of these requests (much like in the previously analyzed system [17]), summed up with the storage costs of the placed replicas. The authors developed a very lightweight heuristic approach for minimizing this function, by delegating to each node the task of performing the placement decisions relative to their currently stored replicas in an autonomous way.

Each node executes an instance of the placement algorithm, and is aware not only of itself but also of its immediate neighbours. As so, the inputs used by the algorithm include the number of requests for each replica stored in the node, the neighbour node that each request is received through, the perceived latency to each neighbour node, and the unit storage price of each neighbour node. The authors make a distinction between source nodes, which can only duplicate items, and edge nodes, which may also migrate and remove replicas.

The algorithm is executed in epochs, where an item is duplicated from node h to neighbour n if the total latency cost, according to the number of requests in the last epoch, is larger than the storage cost for that replica in n. If the duplication condition does not hold, the migration condition is tested, accounting for the reduction in storage cost, from the removal of the replica on the h, and the augment in the latency cost, by assuming that the latency of requests of each other neighbour $i \neq n$ for the replica, will increase, since it is assumed that they will be forwarded through h. This assumption may not hold true for some cases, where i has a lower latency connection to n, bypassing h, resulting in a rougher approximation to the optimal solution. The condition to remove a replica evaluates the expected replica utilization, calculated at creation, against the current number requests. If the value reaches a lower-bound, the replica is removed.

For replica discovery the authors also propose a lightweight solution. When a replica is created in a node n by h, node n sends a notification to each node that requested the replica from h in the last epoch, in a distributed attempt at informing nodes of a lower distanced replica. This strategy is sub-optimal, since new requesters may not have knowledge about the replica, but incurs a low communication overhead, while maintaining some replica-awareness.

Placement is completely distributed and executed with very reduced information, requiring little computation per node, but may lead to the creation of unnecessary replicas and high fluctuation. This, combined with the fact that only one optimization per replica is performed in each round, results in a slow-paced convergence towards the stable solution.

4.1.8 QoS-Aware Replica Placement [19, 20] We now discuss two systems that make QoS-Aware replica placement.

In [19] the authors define a QoS-Aware replica placement problem, with a cost model that accounts for replica storage costs and update costs. This model regards the placement of the replicas of a single object. The network is presented as graph of the servers and their connections, in which each server has an associated weight, representing its unit storage cost. Moreover, a distance $d(u, v)$ is associated with each edge $(u, v) \in E$, representing the communication cost between the nodes. This distance definition is also extended to every pair of nodes, representing the shortest path between them.

The object is associated with an origin server, where the content provider may perform updates. If this object is replicated at a server that receives requests for it, the response is generated locally, otherwise, the server contacts another server in the network and relays the response.

Every server in the network has a QoS requirement $q(v)$ which specifies an upper bound on retrieval cost, which is the distance $d(v, u)$ to the closest server u that holds a replica. The objective of the QoS-aware placement problem is to find a replication strategy (defined as the set of nodes R in which the object is placed), that satisfies the QoS requirements of all servers, while minimizing the replication cost. This cost is a weighted sum of the storage cost of all placed replicas, and the update cost, which refers to the necessary communication to update the replicas. For updates, the model assumes that the network is organized into an update distribution tree structure, rooted at the origin server, which propagates the updates in an efficient way. The update cost is computed by multiplying the distances $d(u, v)$ needed to propagate the updates, with the update rate value of the origin server. Therefore, this cost rises with the number of replicas and their latency to the origin.

Building on the previous model, [20] also considers the storage cost of each node and the cost to propagate updates from an item's source to its replicas, through the distribution tree. Furthermore, this work also takes into account the access cost, which is the distance from a consumer node v to the replica node u that is assigned to serve v, and other consumers. The set of consumers served by u is denoted $SS(u)$, and the sum of the workloads $W(v)$ from each consumer v, must not surpass u's capacity constraint, $C(u)$. The replication strategy is considered feasible if no replica servers are overloaded and the QoS requirements are satisfied. The objective of the scheme is to find a feasible strategy that minimizes the storage, update and access costs.

The first algorithm presented is named Greedy-Remove, starting with a replica in every server, and subsequently removing each replica, and adjusting workload attribution, according to the largest possible cost reduction. In the first iteration, the service set $SS(v)$ of each replica server v contains only itself. These sets are then adjusted for every pair of servers, while maintaining feasibility, considering two separate cases:

Case 1: The replica is removed from server v by shifting all servers in $SS(v)$ to $SS(u)$, which is only possible when u is within the QoS range of every server in $SS(v)$, and the workload addition doesn't exceed $C(u)$.

Case 2: A portion of the workload is shifted from v to u, by moving only some servers in $SS(v)$ to $SS(u)$, obeying to the same conditions as the first case, plus a rule which dictates that there must be a distance reduction, $d(w, u) < d(w, v)$, for any server w that shifts to $SS(u)$. While in case 1, replicas are removed according to the reduction in storage costs, update costs, and distance between nodes and replicas, in this case the only affected factor is distance, since we are only dealing with workloads.

A second algorithm, Greedy-Add, was also proposed, starting with an empty replication set, and adding replicas greedily, not only until the strategy is feasible, but, until the total cost is impossible to reduce, since, after achieving feasibility, the access costs can be reduced further by adding more replicas to the system. This algorithm is more computationally efficient than Greedy-Remove, but achieves slightly less optimal results. Nevertheless, both algorithms are computationally heavier than most of the schemes we've analyzed.

These strategies, when applied to a global system, with several objects, make for a somewhat decentralized scheme, where the placement decisions can be performed at the origin servers for each item. The goal of the algorithm is to satisfy latency constraints, and reduce costs without considering request patterns, therefore it doesn't need to adapt to dynamic workloads. And since the algorithm calculates the near-optimal solution in a single round, convergence is immediate.

There is awareness of the underlying network topology, and an assumption that updates are propagated through a tree structure. Routing is replica aware, due to the fact that requests are routed to the nearest replica, but since we have no knowledge on how synchronization and replica discovery would be implemented, we make no assumptions on the communication overhead.

4.2 Discussion

The aforementioned schemes were each developed for different systems, and considering different goals. Each of them succeeds in some factors and flounder in others. In Table 1 we can see a comparative overview of these schemes.

Placement Schemes	Optimality	Distribution	Replication degree	Routing	Comp. overhead	Comm. overhead	Adaptability to dynamic workloads	Convergence	Decision factors
EAD [11]	Medium	Distributed	Adaptive	Replica-blind	Low	Low	Strong	Fast	Popularity
CAC [12]	Medium	Distributed	Adaptive	Mixed	Low	Low	Weak	Fast	Link congestion, Popularity
ABC [13]	Low	Distributed	Adaptive	Replica-blind	Low	Very low	Strong	Slow	Node-server distance, Popularity
ADP [14]	High	Centralized	Fixed	Replica-aware	High	High	Strong	Slow	Co-location of associated data, Popularity
AutoPlacer [16]	High	Distributed	Fixed	Replica-aware (efficient lookups)	Medium	High	Weak	Medium	Popularity
Cooperative Caching [17]	High	Distributed	Adaptive	Replica-aware	Medium	Very high	Strong	Medium	Access costs, Popularity
Holistic-All [17]	High	Distributed	Adaptive	Replica-aware	Medium	High	Strong	Slow	Access costs, Popularity
D-Rep [18]	Medium	Distributed	Adaptive	Mixed	Low	Low	Strong	Slow	Storage costs, Access costs, Popularity
QoS-Aware Replica Placement [20]	High	Distributed	Adaptive	Replica-aware	High	N/A	N/A	Fast	Storage costs, Update costs, Load balance, Access costs, QoS constraints

Table 1: Comparative overview of the existing replica placement schemes.

Placement schemes that use replica-blind routing, such as [11, 13] are very efficient, but are restricted to placing replicas along the request paths, hindering optimality. [12, 18] attempt at maintaining efficiency, while improving optimality, mixing the two routing strategies, however, [12] still places replicas along request paths, while [18] performs other approximations in decision-making. These strategies prevent the schemes from achieving a significantly more optimal solution. Along with that, they also suffer from, respectively, weak adaptability and slow convergence.

To help with the inefficiencies of replica-aware routing, [16] uses a probabilistic structure, but this scheme has weak adaptability and uses fixed replication degrees. [14] solves the adaptability problem, and computes near-optimal solutions, but it is centralized, requiring the collection of data from all nodes, which is infeasible in the edge computing paradigm, since there can be a very high number of nodes and items. By considering the co-location of associated data as a decision factor, in order to perform the placement of a single item, this scheme requires information on the access patterns of its associated items, and, thus, a more global view of the system. Because of this, it is hard to develop an efficient solution for this cost model, while maintaining high optimality.

The Cooperative Caching scheme [17] is distributed, has strong adaptability and high optimality, but incurs a very high communication overhead in each placement decision. Holistic-All [17] lowers this overhead, trading-off with a slower-paced convergence, but still needs each node to gather information from all the other nodes.

As we have mentioned, one of the main goals of the edge paradigm is to reduce the delay from clients to content. As so, the aim of a placement scheme for the edge is to place data closer to the nodes that request it the most, which depends on factors such as request popularity, and proximity to replicas.

Proximity can be defined as the latency or cost for accessing a replica, hence, we can affirm that both of the aforementioned factors are accounted for in [17, 18, 20]. However, none of these strategies consider the possibility of differential consistency in requests. [20] considers update costs, associated with the write operation, but it assumes a specific topology and routing algorithm, not allowing for other ways to propagate the requests, or for differing consistency guarantees in each read/write operation, such as what we exemplified in Section 3.4.

Our objective is to develop a decentralized and adaptive replication scheme for dynamic workloads, that accounts for the different costs of each request, depending on the type of operation being performed and its supported consistency model, all the while striking a balance between optimality, efficiency and convergence speed.

5 Architecture

In this section we describe our solution for replica placement in the edge paradigm. We start by stating our assumptions about the system's components, in particular about the edge nodes and data objects. We then describe the optimization problem we aim at solving, by defining the cost function and the constraints. Finally, we discuss an heuristic approach to approximate the optimal solution in an efficient and distributed way.

5.1 Edge Nodes

We assume that the system is composed of a set of N edge nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_n\}$. Each node n_i has a known capacity in terms of storage, denoted $\mathit{capacity}(n_i)$. We assume the availability of a monitor system that can report the latency of the network path connecting any two nodes n_i and n_j, denoted $\delta(n_i, n_j)$.

5.2 Data Objects

We assume that the system must handle the deployment of a set of O objects $\mathcal{O} = \{o_1, o_2, \ldots, o_o\}$. Each object o_i has a known volume, denoted $\mathit{volume}(o_i)$, which represents the amount of storage capacity it consumes when it is stored in a given edge node. Each data object o_i supports a set of operations op_i^1, op_i^2, ..., op_i^k; as it will be clear later in the text, each operation may have a different cost, depending on its semantics (whether it is a read or a write, which consistency model it supports, etc).

5.3 Object Deployment

Each object may be replicated in a different set of edge nodes, and the nodes that maintain replicas of an object may change in time. It is the role of our placement algorithm to define which edge nodes replicate each object. We assume that for each object o_i there is a single edge node that serves as a master replica for the object, denoted $\mathit{source}(o_i)$. At this stage we assume that the source for each object is fixed (i.e., it does not change over time). The set of edge nodes that keep a replica of a given object o_i are denoted $\mathit{replicas}(o_i)$. The remaining edge nodes are denoted the $\mathit{consumers}(o_i)$. The deployment of an object is a tuple defined by its source, replicas and consumers, i.e.:

$$\mathit{deployment}(o_i) = (\mathit{source}(o_i), \mathit{replicas}(o_i), \mathit{consumers}(o_i))$$

with

$$(\{\mathit{source}(o_i)\} \cup \mathit{replicas}(o_i) \cup \mathit{consumers}(o_i)) = \mathcal{N}$$

5.4 Cost of an Operation

We assume that the cost of a given operation op_i^k, when executed at a given edge node n, can be expressed as a function of n and of the object deployment $\mathit{deployment}(o_i)$:

$$\mathit{cost}(\mathit{op}_i^k, n) = \mathit{costfunction}_i^k(n, \mathit{deployment}(o_i))$$

The $\mathit{costfunction}_i^k$ for each operation allowed for a given object must be provided by the user of our system. To ease the task of defining the appropriate cost functions, we plan to offer a library of cost functions that can be used when configuring the system.

5.5 Example Cost Functions

We illustrate the use of cost functions with a concrete example. Consider that the user wants to optimize the system for latency, and that, therefore, the cost of each operation is the latency required to execute that operation. Consider an object that supports two operations, namely a write operation op^w and a read operation op^r, using a primary backup replication scheme where the source of the object plays the role of the primary.

A write operation is executed by sending the update to the primary, then having the primary send the update, in parallel, to all replicas, waiting for all the acknowledgements from these replicas and, finally, sending back an acknowledgement message to the edge node that executed the operation. The latency of the write operation can be captured by the following cost function:

$$\mathit{wcost}(n, \mathit{deployment}(o)) = 2\delta(n, \mathit{source}(o)) + 2 \max_{\forall j \in \mathit{replicas}(o)} \delta(\mathit{source}(o), j)$$

The read operation is executed by performing the read locally, if the node n maintains a replica of the object, or by sending the read request to the nearest replica. The latency of the read operation can be captured by the following cost function:

$$\mathit{rcost}(n, \mathit{deployment}(o)) = 2 \min_{\forall j \in \mathit{replicas}(o)} \delta((n, j)$$

To allow for context-sensitive differential consistency, several types of read/write operations can be defined for the same object, along with their adequate cost functions.

5.6 Access Frequencies, Deployment Cost, and Total Cost

We assume that each node n can keep a record of the frequency of each operation op_i^k it requests, denoted $f_n(op_i^k)$. This value can be estimated using an exponential moving average (EMA) technique, such as the one employed in [11].

The cost of maintaining a given deployment configuration for an object is defined as the sum of the costs for all operations on all nodes, times the corresponding frequency rates, i.e.:

$$\mathit{cost}(\mathit{deployment}(o_i)) = \sum_{\forall_n} \sum_{\forall_k} \mathit{cost}(op_i^k, n) \cdot f_n(op_i^k)$$

And the total cost of the system is the sum of the costs of maintaining the deployments of all objects:

$$\mathit{totalcost} = \sum_{\forall_i} \mathit{cost}(\mathit{deployment}(o_i))$$

Note that this cost model can be easily extended to consider other types of costs (such as storage costs), by computing, instead, a weighted sum of these costs, as was seen in [14, 19].

5.7 Optimal Placement

Using the definitions above, the optimal placement would be a set of deployment configurations that would minimize the *totalcost*, while respecting the capacity constraints at each mode. The capacity constraint can be expressed as:

$$\forall_n : \mathit{capacity}(n) \geq \sum_{\forall_i : n \in \mathit{replicas}(o_i)} \mathit{volume}(o_i)$$

This optimization problem is analogous to the one analyzed in [9], which is NP-Hard, as we've mentioned in Section 3.3. We, therefore, resort to an heuristic strategy, in order to approximate the solution.

5.8 Heuristic Solution

The heuristic solution decentralizes the placement algorithm, by letting the source of each object be in charge of the decisions regarding the deployment of that object. For the sake of simplicity, this solution assumes that the system is only optimizing for latency, however, it can be intuitively adapted to include other types of operation-related costs, such as the amount of information transferred in each request or the bandwidth usage in each connection.

The scheme operates in rounds, in each of which, the nodes estimate $f_n(op_i^k)$ for each object they have requested. In each round, a *shrinking* phase and an *expansion* phase occur sequentially.

5.8.1 Reducing the Replica Candidates While the shrinking phase is concerned with removing replicas, the expansion phase is concerned with creating replicas. In this last phase, the system may consider any node for replication. With a large number of nodes, calculating the best deployment may involve a lot of information and a large computation cost, even if done for a single item. Because of this, we will only consider certain nodes for replication, reducing the set of deployment options to be evaluated.

Considering the deployment for a single object o_i, we allow each replica node n to collect the access frequencies, $a_n(op_i^k)$, for each operation it served in the current round, the set consumers from which it received these requests, C_n, and the average latency $\overline{\delta}_n$. These values are sent to the source, which orders the C_n sets by estimated weight:

$$\mathit{weight}(C_n) = \sum_{\forall_k} a_n(op^k) \cdot \overline{\delta}_n$$

and picks these sets in sequence, computing their union, $P = (C_j \cup \ldots \cup C_l)$, until the following condition is verified:

$$\mathit{weight}(P) > \lambda \cdot \sum_{\forall_n} \mathit{weight}(C_n)$$

where λ is an adjustable parameter. The nodes from the computed set P will be the candidates to become replica nodes in the expansion phase.

5.8.2 Shrinking Phase Each node n, will calculate its part of the total $\mathit{cost}(\mathit{deployment}(o_i))$, for each deployment where the replica number is reduced by 1. This is done by moving each single node $j \in \mathit{replicas}(o)$ to $\mathit{consumers}(o)$. The nodes, after calculating these values, transfer them to the source node, which aggregates them and computes the total cost for each deployment. If the total cost is reduced by any of these changes, the lowest cost deployment is picked and applied to the system, sending the replica to the correspondent node. The process repeats until it verifies that the lowest cost deployment isn't lower than the previous total cost.

When the cost can't be reduced, the cost difference rd_i^n of removing the replica of item o_i from node n, is computed for each replica node, by the source, and sent to them for future use in the next phase. This way, each node contains, for each of its replicas, the potential cost difference obtained by removing them.

5.8.3 Expansion Phase Each node, after obtaining the set of candidates P, from the source, will compute its part of the total cost for the deployment options where the number of replicas is increased by one of these candidates. The rest of the phase executes in an analogous way to the shrinking phase, with the source node picking the best deployment options, until there is no further reduction in cost.

When a replica of o_i is created in a node n, besides receiving the replica, n also receives the total cost difference obtained by adding that replica, ad_i^n. If, in the current phase, the addition of o_i to node n exceeds its storage capacity, its ad_i^n value is summed up with the rd_j^n value of each resident replica, replacing the one with which it obtains the lowest value, if, and only if, this value is negative.

5.8.4 Reducing the Participants To further decrease the communication and computation costs in the deployment decisions, we can reduce the number of participants, in each round of the algorithm. For each set C_n of nodes that weren't picked as candidates for replication, we take its correspondent operation access frequencies $(a_n(op_i^k))$, and average latency $(\overline{\delta}_n)$ from the nodes in the set to the replica n, and we agglomerate the data into a virtual node v_n. This node represents the whole set C_n, and its latency to any other node m is calculated by:

$$\delta(v_n, m) = \overline{\delta}_n + \delta(n, m)$$

This latency value is used with the access frequencies to estimate the deployment costs from the requests of the consumer nodes that contacted with n in the current round, instead of calculating the exact cost for each of these nodes. The costs can be computed either at the source or replicas, and this method can be used in both phases.

5.9 Using Workload Predictions

Instead of only using the current access frequencies to drive the deployment of objects, we plan to also use predicted access frequencies whenever available. In this case, the monitoring system would provide the predicted values for the next round, which may then be fed into the decision algorithm of the current round. Deployment actions could be driven by a weighted sum of the costs computed from current observations and the costs computed from the predicted values.

6 Evaluation

To evaluate the proposed solution we will use the simulation system developed for testing LiveMap [5], which was built by extending the SUMO framework [21]. This system is able to simulate vehicle movement, with support for real maps and realistic traffic patterns. It is also able to simulate in-vehicle applications that communicate with fixed infrastructure, and allows interfacing with real implementations of system components in runtime. This way we can simulate data accesses from these vehicles to edge cloudlets, and execute the replica placement scheme on the cloudlet infrastructure, to improve request delay. To demonstrate the support for differential consistency, we will define different operations that can be applied to each item, and generate workloads that use these operations.

We will evaluate the performance of the heuristic solution, by comparing its results according to several metrics, with the results obtained with our implementation of Holistic-All [17], which has high optimality, but has slow convergence and a high communication overhead, affecting the throughput of the system; and our implementation of D-Rep [18], which has a low communication overhead, but also slow convergence, and lower optimality, affecting its general effectiveness.

The complete set metrics we account for span across four topics: 1) Effectiveness; 2) Convergence; 3) Communication overhead; 4) Optimality.

Effectiveness: To measure the effectiveness of the scheme we will use, as performance metrics, the average delay of requests, the throughput, and the number of created replicas. These metrics will also be used to measure how well the system adapts to changes in the access patterns, by testing with dynamic workloads.

Convergence: We will measure convergence as the average necessary time until the system reaches a stationary point in which no more placements are performed. In this evaluation, we plan to use static workloads, since this metric is not related with how the access patterns evolve through time.

Communication overhead: To measure the communication overhead, we will use the total number of messages sent between nodes to perform the placement algorithm, including any data collection or synchronization prior to each deployment, over a time period, whose duration is yet to be defined.

Optimality: We will measure optimality using the total deployment cost of the computed placement in each single round. We will compare the results obtained with the heuristic against the exact solution of the optimization problem. Since computing the exact solution is NP-Hard, we will only evaluate optimality for small networks and workloads.

7 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

8 Conclusions

In this report we have addressed the problem of replica placement in edge computing scenarios. We did a survey of a number of replica placement strategies that can be used in this context. Based on these previous work we propose a replica placement strategy that can be configured to support different data replication schemes (associated to different data consistency models). The algorithm is designed to operate in a decentralized way, where the replica placement of different items is computed by different nodes, more specifically, each data item may be replicated in multiple replicas, but there is a single replica, which we name the source replica, that is responsible for computing the placement for that item. This strategy, avoids a single point of control for the entire system. Replication among source replicas is performed indirectly, using the feedback provided by the non-source replicas. As future work we plan to implement our proposal and compare its performance against previous approaches.

Acknowledgments We are grateful to Nivia Quental for the fruitful discussions and comments during the preparation of this report.

References

1. Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., Kong, J., Jue, J.P.: All one needs to know about fog computing and related edge computing paradigms: A complete survey. Journal of Systems Architecture (2019)

2. Networking, C.V.: Cisco global cloud index: Forecast and methodology, 2016–2021. White paper. Cisco Public, San Jose (2016)
3. Kasprzok, A., Ayalew, B., Lau, C.: Decentralized traffic rerouting using minimalist communications. In: 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC). (2017) 1–7
4. Zeadally, S., Hunt, R., Chen, Y.S., Irwin, A., Hassan, A.: Vehicular ad hoc networks (vanets): status, results, and challenges. Telecommunication Systems **50**(4) (2012) 217–241
5. Hu, W., Feng, Z., Chen, Z., Harkes, J., Pillai, P., Satyanarayanan, M.: Live synthesis of vehicle-sourced data over 4g lte. In: Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems. (2017) 161–170
6. Chen, F., Zhang, D., Zhang, J., Wang, X., Chen, L., Liu, Y., Liu, J.: Distribution-aware cache replication for cooperative road side units in vanets. Peer-to-Peer Networking and Applications **11**(5) (2018) 1075–1084
7. Mahmood, A., Casetti, C., Chiasserini, C.F., Giaccone, P., Harri, J.: Mobility-aware edge caching for connected cars. In: 2016 12th Annual Conference on Wireless On-demand Network Systems and Services (WONS), IEEE (2016) 1–8
8. Zhang, F., Xu, C., Zhang, Y., Ramakrishnan, K., Mukherjee, S., Yates, R., Nguyen, T.: Edgebuffer: Caching and prefetching content at the edge in the mobilityfirst future internet architecture. In: 2015 IEEE 16th International Symposium on a World of wireless, mobile and multimedia networks (WoWMoM). (2015) 1–9
9. Kangasharju, J., Roberts, J., Ross, K.W.: Object replication strategies in content distribution networks. Computer Communications **25**(4) (2002) 376–383
10. Mayer, R., Gupta, H., Saurez, E., Ramachandran, U.: Fogstore: Toward a distributed data store for fog computing. In: 2017 IEEE Fog World Congress (FWC). 1–6
11. Shen, H.: An efficient and adaptive decentralized file replication algorithm in p2p file sharing systems. IEEE Transactions on Parallel and Distributed Systems **21**(6) (2009) 827–840
12. Badov, M., Seetharam, A., Kurose, J., Firoiu, V., Nanda, S.: Congestion-aware caching and search in information-centric networks. In: Proceedings of the 1st ACM Conference on Information-Centric Networking. (2014) 37–46
13. Ming, Z., Xu, M., Wang, D.: Age-based cooperative caching in information-centric networking. In: 2014 23rd International Conference on Computer Communication and Networks (ICCCN), IEEE (2014) 1–8
14. Yu, B., Pan, J.: Location-aware associated data placement for geo-distributed data-intensive applications. In: 2015 IEEE Conference on Computer Communications (INFOCOM). (2015) 603–611
15. Agarwal, S., Dunagan, J., Jain, N., Saroiu, S., Wolman, A., Bhogan, H.: Volley: Automated data placement for geo-distributed cloud services. (2010)
16. Paiva, J., Ruivo, P., Romano, P., Rodrigues, L.: Autoplacer: Scalable self-tuning data placement in distributed key-value stores. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **9**(4) (2015) 19
17. Sourlas, V., Gkatzikis, L., Flegkas, P., Tassiulas, L.: Distributed cache management in information-centric networks. IEEE Transactions on Network and Service Management **10**(3) (2013) 286–299
18. Aral, A., Ovatman, T.: A decentralized replica placement algorithm for edge computing. IEEE Transactions on Network and Service Management **15**(2) (2018) 516–529

19. Tang, X., Xu, J.: Qos-aware replica placement for content distribution. IEEE Transactions on parallel and distributed systems **16**(10) (2005) 921–932
20. Cheng, C.W., Wu, J.J., Liu, P.: Qos-aware, access-efficient, and storage-efficient replica placement in grid environments. The Journal of Supercomputing (2009)
21. Krajzewicz, D., Erdmann, J., Behrisch, M., Bieker, L.: Recent development and applications of SUMO - Simulation of Urban MObility. International Journal On Advances in Systems and Measurements **5**(3&4) (December 2012)