Cathode: A Consistency-Aware Data Placement Algorithm for the Edge

Leonardo Epifâneo, Cláudio Correia, Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{leonardo.epifanio,claudio.correia,ler}@tecnico.ulisboa.pt

2021 IEEE 20th International Symposium on Network Computing and Applications (NCA) | 978-1-6654-9550-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/NCA53618.2021.9685123

***Abstract*—Data storage has been recognized as one of the key tasks for edge/fog computing infrastructures. Keeping replicas of data near the edge has many advantages, including allowing client to be served directly from the fog layer with lower latency and avoiding short-lived data to be shipped in its entirely to the cloud servers. In both cases, edge storage can offer significant bandwidth savings in traffic to and from the cloud datacenters. However, replica placement on the edge is challenging for multiple reasons. First, objects can be updated by many sources, unlike in classic CDN networks where most updates are centralized. Second, different objects may have different consistency requirements. Third, the number of nodes and objects is very large, which precludes the use of centralized solutions. In this paper, we propose a replica placement algorithm for the edge, named Cathode, that addresses the challenges above. Cathode is decentralized and scalable, providing fast convergence, but also achieving high quality deployments. Furthermore, when making placement decision, it takes into account the data consistency protocol, considering both the cost of update and read operations, leading to different placements for different replica-consistency algorithms. The paper offers an extensive evaluation of Cathode and show that it outperforms previous state-of-the-art replica placement algorithms.**

***Index Terms*—Edge Storage, Data Placement, Consistency**

I. 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 run in resource-constrained devices [1]. In this context, data storage has been recognized as one of the key tasks for edge/fog computing infrastructures [2]. There are two important advantages of storing data at the edge [3]: firstly, edge nodes can provide assistance with much lower latency than the cloud, because servers are physically closer to the devices; and secondly, edge storage can prevent short-lived data from being shipped in its entirely to the cloud servers; instead, only the relevant aggregated information is shipped for deferred processing.

In this work we address the problem of data placement for edge computing. In short, the data placement problem consists in finding a suitable allocation of data objects to edge nodes, subject to a number of constraints, such that one can maximize the utility of these placements for the system. Replica placement on the edge is challenging for multiple reasons. First, objects can be updated by many sources, unlike in classic CDN networks where most updates are centralized. Second, different objects may have different consistency requirements. Third, the number of nodes and objects is very large, which precludes the use of centralized solutions.

978-1-6654-9550-9/21/$31.00 ©2021 IEEE

Although the topic of data placement for edge computing has been addressed before in the literature [4]–[6], most systems assume that all objects offer the same consistency model, enforced by some static, pre-defined, replica consistency algorithm. This is unfortunate, as there is a growing interest in designing systems for edge computing that can support multiple consistency criteria [7]–[10].

This paper presents Cathode, a replica placement algorithm that is consistency-aware, and that makes data placement decisions based on client demand, storage-constraints, and the costs of keeping replicas consistent. Different objects can use different consistency models and Cathode makes placement decisions accordingly. Because optimal replica placement is known to be an NP-hard problem [11], Cathode resorts to an heuristic that is highly decentralized, avoiding the bottlenecks associated to solutions that have a single point of control. Since nodes that replicate the same objects are allowed to share their local views of where replicas are most required, Cathode is able to yield a high utility. This allows Cathode to approximate the utility provided by algorithms such as Holistic-All [12], but with better efficiency, by parallelizing computations among nodes. Furthermore, while other algorithms for the edge are unable to provide satisfactory results when objects use different consistency models, Cathode provides a scalable structure to support placement optimizations for multiple consistency protocols. Cathode mainly targets applications such as smart cities, where the data access patterns are likely to exhibit strong geographical locality. Despite this, Cathode is versatile and can be applied on various levels along the path from the cloud to the edge, by providing several configuration parameters that can be tuned by the system administrator to fit the goals of the underlying system.

II. Background

Data replication is widely used in distributed systems as it can bring many advantages, such as fault tolerance, load balancing, and lower data access latency. In this paper we consider the use of data replication in the context of edge computing with the primary goal of reducing the latency clients experience when accessing data.

A. Data Placement as an Optimization Problem

We can abstract the problem of deciding which replicas are placed in which edge nodes as an optimization problem that aims at minimizing a system wide *cost function*. In this case, the cost is correlated with the access latency for edge clients. If the data is placed in a remote location, and the access latency is large, the cost is high; conversely, if the replica is placed in the edge node used by the client, the latency is small and the cost is low. The cost of a given data placement x can be formulated as follows [11]:

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

where 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 $d_{ij}(x)$ is a cost value represented by the shortest distance from i to a node that contains j, under placement x.

If latency is the sole factor to be considered when computing these costs, the best strategy would be to place a replica of every object at every edge node. Naturally, in a real scenario, this is neither feasible nor desirable. First, edge nodes have limited resources, thus data placement must be performed under the constraint that the assignment needs to respect the capacity of individual edge nodes. Also, there are costs involved in maintaining data replicas. First, when a replica is deployed, data needs to be transferred from another replica (most likely, from a datacenter in the cloud) to the target edge node, a task that consumes network resources. Then, the replica must be kept up to date in face of updates, which, in some systems, involves propagating any changes among all the existing replicas.

The cost of keeping a replica up to date therefore depends on several factors, such as the frequency of updates, and the data consistency criteria that needs to be enforced.

B. Solving the Optimization Problem

It has been shown that an optimization problem expressed above can be mapped to the multiple knapsack problem and is, therefore, NP-Complete [11]. Thus, practical solutions of the data placement problem are solved by heuristics that approximate the optimal solution. The most straightforward way of applying a heuristic is to centralize all the information required in a single node that can run the replica placement algorithm locally, and then, according to its decisions, instruct other edge nodes to fetch or discard replicas.

Unfortunately, the centralized solution has a number of drawbacks. In particular, in most cases, the access patterns (i.e. how often each object is accessed in each edge node, and the corresponding read-write ratio) are not static and known *a priori*. Instead, access patterns are dynamic and need to be estimated in run time. As a result, the placement of replicas needs to be recomputed frequently. While access patterns can be captured on-line by the edge nodes, a centralized solution requires this information to be shipped on a regular basis to a single node, which can easily create a bottleneck in the system. Therefore, there is an interest in studying an heuristic that can be implemented by distributed algorithms, where the load can be distributed among multiple nodes.

When using heuristics that can be executed in a distributed manner, the following criteria should be considered:

- *Approximation Quality*, which defines how well the algorithm approximates to the optimal solution.
- *Efficiency*, which is split into the computation and communication overheads.
- *Convergence speed*, which defines how fast and effective the scheme is at converging 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.
- *Elasticity*, which defines how well the placement adapts to workloads in which the request patterns change over time.

Given the highly dynamic nature of edge environments [13], data placement should be elastic. This means that the placement algorithm must run frequently and thus, it should be efficient and converge quickly. In previous literature, it is possible to find algorithms that provide high approximation quality, but that are not efficient and do not converge quickly, or, on the opposite side, elastic algorithms that achieve efficiency at the cost of sacrificing quality. In the next paragraphs, we explore two examples that detail this trade-off.

Holistic-All [12] is a replica placement algorithm that aims at achieving an high approximation quality. It does so by having each node trade information with every other node at the start of an optimization round. This information is then used by each node to perform placement decisions regarding its own storage space. However, only one node may perform a placement decision at a time, so the execution of the algorithm requires a long serial number of optimization rounds to converge. This effect is amplified in systems with a high number of nodes and with a large diameter network.

D-Rep [4] is an example of an algorithm that trades the approximation quality for efficiency. D-Rep has been designed specifically for the edge and requires very little communication among nodes to perform placement decisions. This is achieved by requiring each node to coordinate only with its direct neighbours. The algorithm is very fast at performing optimization rounds, but the approximation quality is hindered by the fact that each node works with partial information. Placement decisions are performed in parallel, with each node, in each round, being capable of duplicating or migrating a replica of a data object to/from one of its neighbours. Because, in each round, optimizations are local, D-Rep also requires many optimization rounds to approximate the global optimum.

C. Taking Data Consistency in to Account

In [14] the authors elaborate on how *distributed database systems* (DDBSs) have to perform decisions regarding the trade-off between latency and consistency, and exhibits the differences on how several of these systems handle this trade-off. For example, the Apache Cassandra [15] distributed storage system allows for the specification of the consistency level on every single operation. This system has been used as the base for the storage module of the Cloudpath [8] platform, which specifies a multi-tier architecture, with nodes spanning from the cloud to the edge of the network. In this system, updates are propagated following an *eventual consistency* model, but stronger consistency levels may be achieved by the clients when they perform requests. Further development of this storage system added the option to use *session consistency* in the operations performed by the clients, providing yet another form of handling requests [9]. The relevance of supporting different consistency models in the paradigm of edge computing has also been recognized in [7], which proposes a data storage model that allows for different consistency levels in requests, depending on the context in which a client is inserted.

III. CATHODE

In this section we describe Cathode, a placement algorithm that aims at attaining a good balance between the quality of the solution, and the speed at which this solution is achieved. This is accomplished by having a near global, but approximate, view of the problem, whilst minimizing the steps necessary to perform placement decisions. Furthermore, Cathode aims at supporting the use of different replica consistency protocols, such that the programmer can select the consistency model that better matches the application semantics. We will start by describing the system model and then we formalize the optimization problem which the algorithm is set to solve. Finally, we address the Cathode operation and describe how it implements an heuristic that approximates the optimal solution in an efficient and distributed manner.

A. System Model

1) Node Network: We assume that the system is composed of a set of N edge nodes $\mathcal{N} = \{n_1, n_2, \dots, n_n\}$. Each node n_i has a known capacity in terms of storage, denoted $\mathit{capacity}(n_i)$, and a unit storage cost $\mathit{storageCost}(n_i)$, which defines the cost of storing a single data unit in that node.

We assume that edge nodes are connected by a multi-hop network, such that any edge node can communicate with any other edge node. Nodes can communicate to serve client requests, and to coordinate placement decisions. The communication between any two edge nodes n_i and n_j is subject to some (average) delay denoted $\delta(n_i, n_j)$. We assume that these delays are known by Cathode. The manner in which these values are measured is orthogonal to our contribution: E.g. nodes can ping other nodes to estimate the delays between each other, or rely on some external monitoring infrastructure to obtain this information.

2) Data Objects: We assume that the system must handle the deployment of a set of O objects $\mathcal{O} = \{o_1, o_2, \dots, o_o\}$. Each object o_i has a known volume, denoted $\mathit{vol}(o_i)$, representing 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^v; and each operation may have a different cost, according to its semantics (whether it is a read or a write operation, which consistency model it supports, etc) and to the current data placement. The granularity of these objects will depend on the application that is using the algorithm, and is defined by the system administrator.

3) Clients and Requests: We assume clients are not aware of the data placement. Clients are attached to a given edge node and send requests to that node. The system model abstracts from these clients and merely consider these requests as "emerging" in nodes. Each request is characterized by a target object o_i and the specific operation op^v performed on that object. The semantics of each operation then depend on the consistency model being used.

4) Object Deployment: Each object may be replicated in multiple edge nodes. The set of nodes that store the replicas of a given object may change in time, according to how the replica placement algorithm adapts to the changes in the workload (number of clients, frequency of request, etc). It will then be the role of Cathode to define which edge nodes replicate each object. For each object o_i there is a single edge node that serves as a *source* replica for the object, denoted $\mathit{src}(o_i)$. The full set of edge nodes that keep a replica of a given object o_i, including its source, is denoted $\mathit{replicas}(o_i)$. The replica of object o_i in node n is denoted $\mathit{replica}(o_i, n)$. The edge nodes that perform requests for given object, are denoted the $\mathit{consumers}(o_i)$. A object deployment is a tuple defined by the identifier *id* of the source and the set of replicas:

$$\mathit{deployment}(o_i) = (\mathit{id}(\mathit{src}(o_i)), \mathit{replicas}(o_i)) \quad (2)$$

B. Operation and Consistency Models

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

$$\mathit{cost}(\mathit{op}_i^v, n) = \mathit{costfunction}_i^v(n, \mathit{deployment}(o_i)) \quad (3)$$

The $\mathit{costfunction}_i^v$ may be different for each operation and allows to model different replica consistency protocols. Cathode users may provide their own cost functions to accommodate novel replication strategies, making the system extensible. To ease the task of defining the appropriate cost functions, Cathode includes a library of predefined cost functions, for several widely used consistency protocols, that can be used when configuring the system.

1) 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. We call this operation WRITESOURCE. The latency of the WRITESOURCE operation can be captured by the following cost function:

$$\begin{aligned} &\textit{wcost}(n, \textit{deployment}(o)) = \\ &2[\delta(n, \textit{src}(o)) + \max_{\forall j \in \textit{replicas}(o)} \delta(\textit{src}(o), j)] \end{aligned} \tag{4}$$

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. We call this operation READCLOSEST. The latency of the READCLOSEST operation can be captured by the following cost function:

$$\textit{rcost}(n, \textit{deployment}(o)) = 2 \min_{\forall j \in \textit{replicas}(o)} \delta((n, j) \tag{5}$$

Several types of operations, each with their own consistency requirements and differing semantics, can be allowed for the same object, and it is the role of Cathode to optimize the placement of that object, accounting for the observed costs of each operation, according to their cost functions. Section VII provides more examples of cost functions.

IV. CATHODE DATA PLACEMENT

A. Optimal Placement

The main objective of the replica placement will be to reduce the value of the total cost. We assume that each node n keeps track of the frequency $f_n(op_i^v)$ of each operation op_i^v that it handles (i.e., requests received from attached clients). The cost of maintaining a given deployment configuration for an object o_i is defined as:

$$\begin{aligned} \textit{cost}(\textit{deployment}(o_i)) = \textit{oCost}(\textit{deployment}(o_i)) + \\ \textit{sCost}(\textit{deployment}(o_i)) \end{aligned} \tag{6}$$

with *oCost* representing the operations' cost:

$$\textit{oCost}(\textit{deployment}(o_i)) = \sum_{\forall n} \sum_{\forall v} \textit{cost}(op_i^v, n) \cdot f_n(op_i^v) \tag{7}$$

and *sCost* representing the storage cost:

$$\textit{sCost}(\textit{deployment}(o_i)) = \sum_{\forall n} h_i^n \cdot \textit{storageCost}(n) \cdot \textit{vol}(o_i) \tag{8}$$

where h_i^n is a binary which takes the value of 1 when a replica of o_i is stored in n, and 0 otherwise.

The total cost of the system is the cost of maintaining the deployments of all objects:

$$\textit{totalcost} = \sum_{\forall i} \textit{cost}(\textit{deployment}(o_i)) \tag{9}$$

Building up from the definitions above, the optimal placement is 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:

$$\forall_n : \textit{capacity}(n) \geq \sum_{\forall_i : n \in \textit{replicas}(o_i)} \textit{vol}(o_i) \tag{10}$$

Since this optimization problem is NP-Hard, as we have mentioned in Section II, we resort to an heuristic strategy, in order to approximate the optimal solution.

B. Data Placement Metadata

We decentralize the placement algorithm, by letting the source of each object be in charge of the decisions regarding the deployment of that object based on information it receives from the nodes that replicate the object.

The algorithm operates in epochs. In each epoch, every node n estimates $f_n(op_j^v)$ for each object o_j for which $n \in \textit{consumers}(o_j)$. Alongside, for each replicated object ($\textit{replica}(o_i, n)$), that node n stores ($n \in \textit{replicas}(o_i)$), two structures are computed:

candidateSet: a set of tuples $(m, w_n(o_i, m))$, in which m is a consumer of $\textit{replica}(o_i, n)$, and $w_n(o_i, m)$ is a weight that represents the benefit for creating a replica in m, estimated by n. The way this weight is estimated is by:

$$w_n(o_i, m) = a_n(o_i, m) \cdot \delta(m, n) \tag{11}$$

in which $a_n(o_i, m)$ is the frequency of requests from m to $\textit{replica}(o_i, n)$. The *candidateSet* is then composed of the k_{cand} tuples with the largest weights. Thus, k_{cand} determines the number of candidates chosen by each replica node n.

summaryTuple: a tuple that is a summarized representation of the consumers of $\textit{replica}(o_i, n)$. This tuple is defined as $(\textit{consumers}(o_i, n), w_n(o_i))$, where $\textit{consumers}(o_i, n)$ are the consumers of the replica, and $w_n(o_i)$ is the average of the weights $w_n(o_i, m)$ of each consumer.

These two structures, calculated by the replicas, are sent at the end of the epoch, from the replicas to the source node of the object. The source then selects the *candidates* to replication and the *participants* in the process. To select the *candidates*, it uses the sets $\textit{candidateSet}(o_i, n)$ from each replica n, to compute a global set, $\textit{candidateSet}(o_i)$ of the k_{cand} consumers with the largest weights. To select the *participants*, the source takes each $\textit{summaryTuple}(o_i, n)$, and picks the consumer sets with the largest weights, until the summed weight of the sets surpasses a threshold $T_{part} \in [0, 1]$. The union of these "large weight" sets is denoted $\textit{participants}(o_i)$.

C. Data Placement Algorithm

Each optimization round is composed of two phases, executed sequentially, the *shrinking* phase, used to reduce the number of replicas, and the *expansion* phase, used to increase the number of replicas. The shrinking phase is executed before the expansion phase. We describe these phases next.

1) Shrinking Phase: The goal of the shrinking phase is to check if it is possible to reduce the cost of the current configuration by eliminating one or more replicas. The shrinking phase starts by estimating, for every combination of k_{comb} replicas, the cost of an alternative configuration where that combination of replicas is removed. The *gain* that can be achieved by selecting a alternative configuration, is the difference of the cost of the current configuration and the estimated cost of the alternative configuration. Cathode selects the alternative configuration with the largest estimated gain and, if the estimated gain is above some minimal threshold T_{gain}, that configuration is selected.

To avoid overloading the source node when estimating the gains above, we parallelize the computation of these gains as follows. The source node contacts the nodes in *participants*(o_i). Since the estimated cost of a alternative configuration is the sum of the costs incurred by the operations requested by each consumer, we have these consumers locally compute their own contribution for the cost of the current configuration, and for the cost of each of the r alternative configurations that will be considered by the source. The consumers will then send those partial costs to the source, which can quickly sum these contributions to estimate total cost of each alternative configuration, and its associated gains.

2) Expansion Phase: The goal of the expansion phase is to check if is possible to reduce the cost of the current configuration by adding one or more replicas. As in the shrinking phase, this is performed by estimating the possible gains that can be achieved by selecting alternative configurations that, in this case, include 1 to k_{comb} more replicas than the current configuration. There is however a significant difference between the shrinking phase and the expansion phase. While we assume that the average number of replicas of any given object is relatively small and, therefore, the number of target configurations for the shrinking phase is also small, the number of potential target configurations for expansion is very large because, in theory, any edge node could be considered as a candidate to host a potential new replica. As so, to avoid estimating gains for an extremely high number of configurations, the only nodes considered by the participants in the expansion phase, are the nodes in *candidateSet*(o_i, n), which are essentially, some of the (estimated) ”best” nodes in which to create a replica. The rest of the expansion phase is executed in a way that is analogous to the shrinking phase: The *src* contacts the participants, the costs are computed in a distributed way, sent to the *src*, which estimates the largest gain alternative deployment, and, if the estimated gain is above some minimal threshold T_{gain}, that deployment is selected, and a new replica is created on the corresponding candidate(s).

V. Analysis of Cathode

A. Communication Overhead

Regarding the communication overhead, iteration of the algorithm requires 6 steps of communication; this accounts for sending of information from replicas to source, for the shrinking/expansion phases, and for creating the new replicas. The latency of each step is mainly tied to the distance between consumers and replicas, which relates to the network diameter. The computation overheads fully depends on the choice of parameters k_{cand} and k_{comb}, which determine the number of alternative deployments whose costs must be estimated by each consumer in the shrinking and expansion phases. The system administrator is responsible for tuning these parameters to the needs of the system. We discuss them further in Section VII-C.

B. Space Overhead

We now analyze the space overhead regarding the space s_{node} required for bookkeeping in each node, that is captured by the following equations:

$$s_{node}(n) = s_{consumer}(n) + s_{replica}(n) \tag{12}$$

$$s_{consumer}(n) = \left(\sum_{\forall_i : n \in consumers(o_i)} reqOpTypes(n, i) \right) \times (2 \times size(integer)) \tag{13}$$

$$s_{replica}(n) = \sum_{\forall_i : n \in replicas(o_i)} \left(\sum_{m \in consumers(o_i, n)} (2 \times size(integer) + size(double)) \right) \tag{14}$$

where *reqOpTypes*(n, i) is the number of types of operations that n requests for given object i. To illustrate with an example, consider a node n that requests 1000 objects, that performs 2 types of operations on each object, and that stores the replicas of 1000 objects, which are each periodically requested by 100 consumer nodes. Considering the size of an integer as 4 bytes, and a double as 8 bytes, the space necessary for bookkeeping consumer metadata ($s_{consumer}(n)$) in this node would be of 16KB, while the space necessary for bookkeeping replica metadata ($s_{replica}(n)$) would be of, approximately, 1.6MB.

The granularity of each object is defined by the system administrator, and should follow the logic of the application. We expect the application to cluster fine grain objects into reasonable sized “data partitions” that are treated as a single object by the algorithm. This allows the admin to keep the bookkeeping costs of Cathode within some target limits.

C. Replica Discovery and Fault Tolerance

The mechanisms used for replica discovery, are orthogonal to the main contributions of this document, since Cathode's operation is somewhat independent of the operation of these underlying mechanisms. Cathode may use a simple system such as the one proposed in D-Rep [4], or a more complex and precise system, such as the one proposed in AutoPlacer [16].

As seen, Cathode elects one of the replicas (the source replica) to play a special role in the algorithm. If this replica fails, a leader election algorithm should elect another replica as the new source. The implementation of these mechanisms can be achieved by a simple extension of Cathode. In face

Operation	Cost Function
WRITESOURCE	$2[\delta(n, src(o)) + \max_{\forall j \in replicas(o)} \delta(src(o), j)]$
READCLOSEST	$2 \min_{\forall j \in replicas(o)} \delta((n, j)$
WRITECLOSEST	$2 \min_{\forall j \in replicas(o)} \delta((n, j)$
READMAJORITY	$2 \max_{\forall j \in closestQuorum(n,o)} \delta((n, j)$
WRITEMAJORITY	$2 \max_{\forall j \in closestQuorum(n,o)} \delta((n, j)$
READWRITEMAJORITY	$4 \max_{\forall j \in closestQuorum(n,o)} \delta((n, j)$

TABLE I: Cost functions used in the evaluation

of a network partition, the algorithm should elect a source in each partition and let deployment decisions to be executed in parallel in both partitions, without blocking the placement algorithm. When the network partition heals, the leader election mechanism ensures that only one source remains active.

Note that some data consistency models, namely strong consistency models, may block operations in the presence of network partitions. This is unavoidable, and has been captured by the now well known CAP theorem [17]. Still, Cathode will continue to adjust placement for all objects that use weak consistency and that remain live during the partition.

VI. IMPLEMENTATION

A. Time Measurement

To perform a pure and adequate comparison of Cathode to the other replica placement algorithms, we use an abstract time unit named epoch, to represent time, whose value is not absolute, but relative. Therefore, Cathode algorithm is implemented as "round-based". These rounds occur at the end of a time window, an *epoch*, during which statistics, regarding, for example, the access patterns and latencies between nodes, are registered by the nodes. An optimization round then executes placement decisions based on the information collected during the last epoch. Because of this, we use epochs as a measure of time. The (real) time interval that corresponds to an epoch would depend on factors such as the request rate of the system. Our implementation focus is to perform a comparison between the quality of deployments, the request rate is not a factor that exerts a large influence on the performance metrics. This is because it merely affects the values that go into the placement problem's cost function, and not the way the cost function itself is computed. Because of this, we have setup the duration of the epoch to be long enough to collect a mere average of 30 (new) client requests at each edge node. This way, the epoch is merely an abstract time window with the same duration for each algorithm, independent of its operation and speed.[1]

B. Operations and Consistency Models

In our framework, we include just two major classes of operations: READ and WRITE operations, the former does not change the state of the object while the latter does. Each operation has a different implementation depending on the selected consistency protocol, we considered four protocols:

- *linerizability:* We consider the non-blocking implementation of atomic registers proposed in [18]. In this implementation, writes are performed in a majority of replicas, and reads are implemented by, first, reading a majority of replicas, and second, writing back the value (that was read) to a majority of replicas. This write-back phase is required to ensure consistency among multiple reads that are executed in concurrence with a write operation. This consistency model uses the following operations: READWRITEMAJORITY and WRITEMAJORITY.
- *strong consistency primary-backup:* In this implementation all writes are performed directly at the source replica, which then propagates the update to all the other replicas, waits for acknowledgements, and returns to the client. Reads are performed on the nearest replica. This consistency model uses the following operations: READCLOSEST and WRITESOURCE; details in Section III-B1.
- *strong consistency quorum:* In this implementation both reads and writes are performed on a quorum of replicas. We use a majority quorum for both operations, i.e., each quorum includes the $(r/2+1)$ replicas that are closer to the client. This consistency model uses the following operations: READMAJORITY and WRITEMAJORITY.
- *weak consistency:* In this implementation, the read and write operations are always performed on the nearest replica. Updates are propagated to other replicas in the background. This consistency model uses the following operations: READCLOSEST and WRITECLOSEST.

The cost functions for each of these operations are depicted in Table I (see Section III-B1). When comparing Cathode with Holistic-All and D-Rep we use the weak consistency model, as this model is directly supported by all systems.

C. Network Topology

We consider two network topologies to deploy our nodes:

- *Scale-free networks:* In a scale-free network model the number of edges k originating from a given node exhibits a power-law distribution. This model is widely used, due to it capturing the properties of many human-made networks, such as the Internet.
- *Grid networks:* In the grid (lattice) network model, nodes are deployed in a two dimensional grid, with each node having, at most, 4 neighbours. This network model can capture some edge networks, in which edge nodes may be placed along the roads, or areas, of a city.

VII. EVALUATION

We now present our evaluation of Cathode's performance and compare it with Holistic-All [12] and of D-Rep [4]. We aim to show that Cathode strikes a balance between the advantages and disadvantages of these two systems. Additionally, we assess the gains that can be achieved by Cathode, derived from the fact that it is aware of the different operations and consistency models used in the system. Our experimental evaluation aims an addressing the following questions:

- What is the deployment quality provided by Cathode and how fast is it at converging to a stable solution?
- How well does Cathode adapt to dynamic workloads?

[1]As a reference, epochs of 4-7min provide satisfactory results in D-Rep [4].

- Do the semantic-aware mechanisms of Cathode bring advantages over simpler oblivious approaches?

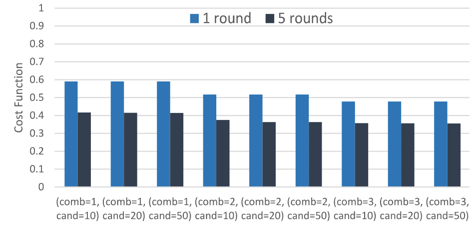
A. Experimental Setup

Our evaluation is performed by simulating the execution of the algorithms, using the PureEdgeSim framework [19]. As previously stated we evaluate the algorithms based on epochs, since all the algorithms we have evaluated are "round-based". In our experiments we considered a system of 1000 objects. To simplify the experiments we assume that all objects have roughly the same size and therefore we assign a unitary storage cost to each replica. We also do not impose any hard limit to the maximum number of objects that each edge node can store (even though our algorithm is able to take these limits into account). Our option to not impose such a limit, is in order to perform a fair comparison with D-Rep, since the latter does not specify behaviour on how to deal with hard limits in the storage capacity of nodes. Note that there is always a storage cost associated with each replica, which acts as a discouraging factor for creating replicas, and an encouraging factor for removing replicas. This way, even when operating in workloads where the *oCost* values always encourage replication, the algorithm will still refrain from creating replicas in every single node.

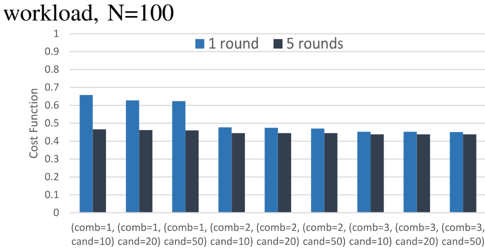
B. Workloads

We consider two different types of workloads:

- *Client-driven workload:* In this workload clients access objects according to their own interests, regardless of their network position. This workload is highly skewed [20] capturing applications such as social networks, news websites, etc. In this workload, every time a node makes a request, it selects which object it is going to access, by sampling a Zipf-like distribution that is attributed to that node with an exponent $\alpha \in [0.7, 0.8]$ this interval represents several types of environments [20]. We divide this workload type in two subtypes, *homogeneous* and *heterogeneous*. In the *homogeneous client-driven workload*, the global distribution of requests also follows a Zipf law. In the *heterogeneous client-driven workload*, clients have independent distributions from one another, making it less skewed.
- *Locality-driven workload:* In this workload objects are geo-referenced and clients access objects based on their own geographic positions. This workload captures, for example, recommendation applications, such as Tripadvisor, where users search for nearby restaurants, museums, etc., or applications in which vehicles search for road conditions in their proximity. In this workload, each object is assigned to a given location and the edge node closest to that location is designated to be the source node for that object. Every time a node makes a request, it selects which object it is going to access following a skewed distribution in which the skew is based on the distance between the client and the object.



(a) Scale-free network, homogeneous client-driven workload, N=100



(b) Grid network, locality-driven workload, N=100

Fig. 1: Cost function after 1 and 5 epochs, with a static workload, for several configurations.

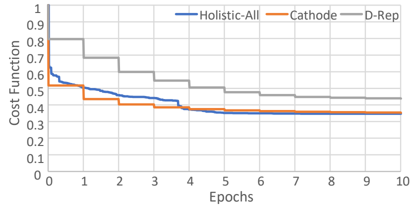
C. Configuring the Parameters of Cathode

As we have seen, the operation of Cathode can be tuned by four configuration parameters:

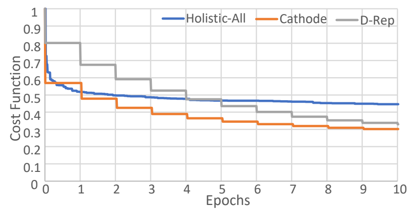
- T_{gain}, the minimum gain (cost reduction) that justifies adding or deleting a replica;
- k_{cand}, that controls how many candidates are considered;
- k_{comb} that controls how many combinations (of creations/removals) are considered;
- T_{part}, that indirectly controls the amount of participants in the optimization process.

The value of T_{gain} is application dependent, it defines the minimum gain that can bring business benefits. To observe the full optimization potential of the algorithms, we set $T_{gain} = 0$ in the evaluation. The other threshold, T_{part}, affects the balance between approximation quality, and speed. We have set this parameter as its maximum value, $T_{part} = 1$, to allow for maximum approximation quality, and to prove that Cathode is able to surpass the converge speed of the other algorithms, even when all consumers participate in the computations. The parameters k_{cand} and k_{comb} affect the several factors, with higher values leading to higher computational overheads, but more precise deployments. To make Cathode as lightweight as possible, the point is to pick low values for these parameters, while still allowing the algorithm to make fast progress.

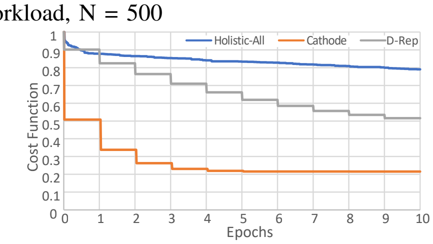
Instead of evaluating every possible combination, we picked several plausible configurations and measured the deployment cost function computed by Cathode after 1 and 5 rounds, depicted in Figure 1. In general, the parameter that appears to exert more influence is k_{comb}. In both networks, the the most noticeable performance improvement, is when k_{comb} is switched from 1 to 2. Despite this, the result after 5 epochs is fairly similar for all configurations. In our experiments we have used $(k_{comb} = 2, k_{cand} = 10)$.



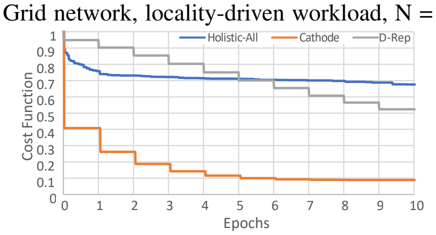
(a) Scale-free network, homogeneous client-driven workload, N = 100



(b) Scale-free network, homogeneous client-driven workload, N = 500

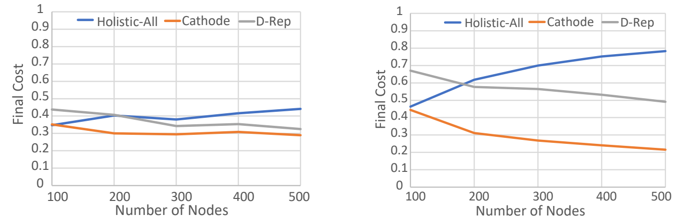


(c) Grid network, locality-driven workload, N = 500



(d) Grid network, heterogeneous client-driven workload, N = 500

Fig. 2: Evolution of the cost function during 10 consecutive epochs with a static workload.



(a) Scale-free network, homogeneous client-driven WL

(b) Grid network, locality-driven WL

Fig. 3: Final cost after 10 epochs with static workloads as a function of system size (N), WL is workload.

D. Convergence Speed

To evaluate the convergence speed of Cathode we set-up an initial deployment in which there is a single replica of each data object (the source). Clients access objects using one of the workloads described in Section VII-B, while the system runs for several epochs. We then plot the evolution of the cost function as the experiment progresses.

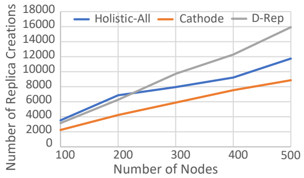
Figure 2 depicts the evolution of the cost function in time, for systems of 1000 objects and varying number of nodes, using different combinations of network topologies and workloads. As shown in Figures 2a and 2b, Holistic-All and D-Rep, work best in scale-free networks. This happens because these networks have a low diameter and good results can be achieved quickly by placing data in well connected hubs. Conversely, on grid networks, D-Rep performs poorly because the placement decisions performed in each node consider a very limited horizon of information (observing only the state of its neighbours), and hence, the algorithm fails to capture the global system behaviour, slowing down convergence.

Figures 2c and 2d show that Holistic-All has a much slower convergence on grid networks. This can be explained by the fact that Holistic-All requires nodes to wait, for several serial rounds, for the placement decisions performed in other nodes. This process turns out to be inefficient in systems with large diameter and a large number of nodes. Cathode outperforms both systems on all large networks. This happens because, on one hand, the information and communication needed to run the algorithm is less than in Holistic-All, making the process faster. On the other hand, replicas are placed exactly where they are needed using data horizon, instead of propagating these replicas from neighbour to neighbour, as D-Rep.

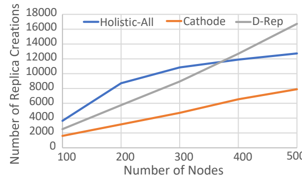
E. Approximation Quality and Scalability

Figure 3 shows the cost of the final deployment of each algorithm, after 10 epochs, for different systems sizes. It can be observed that on scale-free networks the quality of the solution is less affected by the system size in any of the algorithms. This is because the quality of the solution mainly depends on the placement of replicas on the few network hubs, whose number grows logarithmicaly. Conversely, on grid networks, a larger system size normally translates to a higher diameter, and thus a higher initial cost. The same factor that affects the convergence of Holistic-All and D-Rep in grid networks, also affects the quality provided by these systems, that perform considerably worse than Cathode in this setting. We also studied how the number of data objects affected the performance of the algorithms and observed that Cathode is much more conservative, creating less replicas than the other algorithms for higher N values, but achieving better quality. Due to lack of space we do not present those experiences here.

In Figure 4 we have the number of replica creations performed in the same network/workload pairs of Figure 3. As we can see, Cathode is the most conservative algorithm, creating the least number of replicas in each case, Cathode attempts at calculating the deployment that appears to be the most optimal in the present workload. Holistic-All, in the same epoch, may consider several replacements for the same object, depending on the node that is in charge at each moment. While D-Rep is the one that creates the most replicas in larger systems, because replicas need to hop through multiple nodes, in order

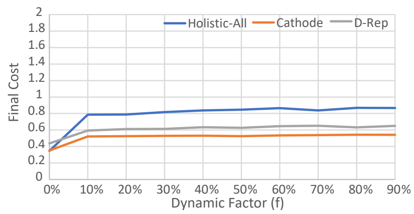


(a) Scale-free network, homogeneous client-driven WL

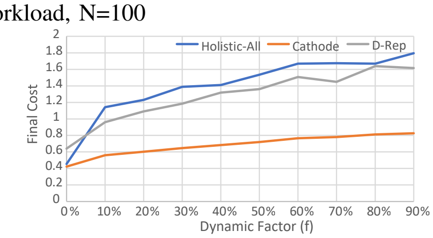


(b) Grid network, locality-driven WL

Fig. 4: Number of replica creations in 10 epochs of runtime, as a function of system size (static workloads), WL is workload.

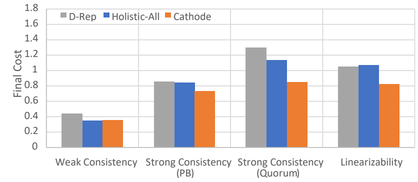


(a) Scale-free network, homogeneous client-driven workload, N=100

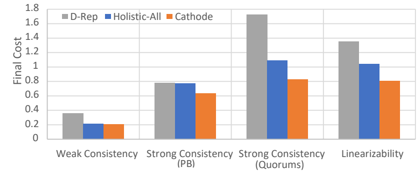


(b) Grid network, locality-driven workload, N=100

Fig. 5: Dynamic workloads



(a) Scale-free network, homogeneous client-driven workload, N=100



(b) Grid network, homogeneous client-driven workload, N=100

Fig. 6: Performance with different consistency protocols

to reach their locations. Algorithms that create a large amount for replicas have to transfer more data between nodes and take up more storage capacity in the nodes, these disadvantages are even aggravated if the object size increases.

F. Dynamic Workloads

In Figure 5 we show how the different algorithms perform when faced with dynamic workloads. For these experiments we let each algorithm run for 10 epochs, but, in each epoch, a percentage f of the sets of objects accessed by the clients, is changed. The changed sets in each epoch are generated from a Zipf distribution correspondent to that epoch.

As we have seen before, scale-free networks are ”easier” to optimize. Thus, in these networks, all algorithms can adapt the deployment in face of dynamic workloads, with reasonable performance. As f rises, we can see that Cathode is able to consistently achieve a better performance than the others. On grids, however, D-Rep and Holistic-All cannot adjust fast enough, and increase, instead of decreasing, the cost of the deployment. In sharp contrast, Cathode still offers considerable benefits even in highly dynamic workloads.

G. Performance with Different Consistency Protocols

We now present the advantages that a consistency-aware placement algorithm can obtain. Figure 6 shows the cost of the final deployment of each algorithm, for the 4 consistency protocols (see Section VI-B). To better showcase the performance of the algorithms using different consistency models we use in both networks a homogeneous client-driven workload.

When using weak consistency, Cathode offers approximately the same utility as Holistic-All (in fact, it performs slightly worse). However, when other consistency models are used Cathode is able to outperform both Holistic-All and D-Rep. One interesting result is the fact that, when quorum strong consistency or linearizability are used, both D-Rep and Holistic-All degrade the performance of the system. This happens because these algorithms have no way of correctly assessing the costs involved when a new replica is added to a quorum based algorithm. Instead, D-Rep and Holistic-All only account for the flow of information, assuming a weak consistency model, where a new replica always reduces the latency experienced in the system, which is not the case when a quorum based replication scheme is used. In contrast, when quorum strong consistency or linearizability are used, Cathode is able to increase the utility by $1.17\times$ on average, due to the fact that it is aware of the semantics of the operations, taking into account the cost function and frequency of each. This support for multiple consistency protocols is made possible because of the operational structure of Cathode, in which a single node (source) is responsible for the deployment decisions of an object, however, the computations necessary for these decisions are distributed among multiple other nodes.

VIII. Related Work

There is quite an extensive amount of literature on the data placement problem [4], [6], [12], [16], [21]–[28] but most of these system aim at different settings and cannot be easily adapted to edge computing. For instance, [23], [24] have been designed for cloud environments, in which the number of nodes is small: their placement algorithm is run in a centralized manager, an approach that is not scalable, nor feasible, in the edge scenario. Some systems consider edge scenarios but are tailored to specific applications. For instance, [27], [28] are designed for specific type of workflows. Some systems consider the costs of updates [25], or the costs of multiple consistency protocols [26], but none of them specify a scalable structure to compute this costs in scenarios like the edge. We now focus on the approaches that are closer to Cathode.

Like Cathode, [21] and [22] also aim at being efficient and decentralized. However, unlike Cathode, they do not account for latency, and also assume that the routing is blind in regards to replicas, only creating replicas in the path from clients to the source. Autoplacer [16] is a data placement algorithm for the cloud that distributes computations among nodes and allows for efficient replica-aware routing. However, Autoplacer requires an all-to-all communication phase that cannot be efficiently executed among edge nodes. In [6], the heuristic is based on geographically partitioning the data placement problem in several regions. However, the algorithm only considers the placement of a single copy of object, disregarding replication as a strategy to reduce latency.

The systems that are conceptually closer to ours are Holistic-All [12] and D-Rep [4], previously introduced in Section II and used in our evaluation. The former illustrates how to achieve a good approximation to the optimal cost and the latter how to achieve a fast optimization process. The objective with Cathode was not to compromise neither optimality nor speed, but to provide both in any system size or topology. Furthermore, unlike these systems, Cathode provides a structure that enables consistency-awareness.

IX. Conclusions and Future Work

Cathode achieves high quality solutions in a scalable way, providing a structure that enables optimizing placement in multiple consistency models, irregardless of system size or topology. Cathode surpasses the performance of other algorithms in most of the analysed cases. In the worst case, it achieves 5% of the utility achieved by quality-focused algorithms. However, in environments such as large grid networks, Cathode is able to surpass the quality of the other solutions by a factor of $1.7\times$. Furthermore, it is able to reduce costs for all the considered consistency protocols, unlike the other solutions. Currently, Cathode already supports 4 different consistency protocols, namely weak consistency, primary-backup based strong consistency, quorum-based strong consistency, and atomic registers. As future work we would like to enrich the library to include protocols that implement causal consistency and session guarantees.

Acknowledgments

This work was partially supported by the FCT under grant 2020.05270.BD, and project COSMOS (with ref. PTDC/EEI-COM/29271/2017), project NG-STORAGE (with ref. PTDC/CCI-INF/32038/2017) and project UIDB/ 50021/ 2020.

References

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

[2] J. Gedeon, F. Brandherm, R. Egert, T. Grube, and M. Mühlhäuser, "What the fog? edge computing revisited: Promises, applications and future challenges," *IEEE Access*, vol. 7, Oct. 2019.

[3] Cisco, "Cisco global cloud index: Forecast and methodology, 2016–2021," *White paper, Cisco Visual Networking*, 2016.

[4] A. Aral and T. Ovatman, "A decentralized replica placement algorithm for edge computing," *IEEE TNSM*, vol. 15, no. 2, pp. 516–529, 2018.

[5] C. Li, J. Bai, and J. Tang, "Joint optimization of data placement and scheduling for improving user experience in edge computing," *JPDC*, vol. 125, pp. 93–105, 2019.

[6] I. Naas, R. Parvedy, J. Boukhobza, and L. Lemarchand, "iFogStor: an IoT data placement strategy for fog infrastructure," in *ICFEC*, Madrid, Spain.

[7] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, "Fogstore: toward a distributed data store for fog computing," in *FWC*, Santa Clara (CA), USA, 2017.

[8] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *SEC*, San Jose (CA), USA, 2017.

[9] S. Mortazavi, B. Balasubramanian, E. de Lara, and S. Narayanan, "Toward session consistency for the edge," in *HotEdge*, Boston (MA), USA, 2018.

[10] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, "Dual-quorum: a highly available and consistent replication system for edge services," *IEEE TDSC*, vol. 7, no. 2, pp. 159–174, 2008.

[11] J. Kangasharju, J. Roberts, and K. Ross, "Object replication strategies in content distribution networks," *Computer Comm.*, vol. 25, no. 4, pp. 376–383, 2002.

[12] V. Sourlas, L. Gkatzikis, P. Flegkas, and L. Tassiulas, "Distributed cache management in information-centric networks," *IEEE TNSM*, vol. 10, no. 3, pp. 286–299, 2013.

[13] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[14] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[15] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *OSR*, vol. 44, no. 2, pp. 35–40, 2010.

[16] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "Autoplacer: Scalable self-tuning data placement in distributed key-value stores," *ACM TAAS*, vol. 9, no. 4, p. 19, 2015.

[17] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, p. 51–59, Jun. 2002.

[18] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *JACM*, vol. 42, no. 1, pp. 124–142, 1995.

[19] C. Mechalikh, H. Taktak, and F. Moussa, "Pureedgesim: A simulation toolkit for performance evaluation of cloud, fog, and pure edge computing environments."

[20] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM*, New York (NY), USA.

[21] H. Shen, "An efficient and adaptive decentralized file replication algorithm in p2p file sharing systems," *IEEE TPDS*, vol. 21, no. 6, pp. 827–840, 2009.

[22] Z. Ming, M. Xu, and D. Wang, "Age-based cooperative caching in information-centric networking," in *IEEE ICCCN*, Shanghai, China, 2014, pp. 1–8.

[23] B. Yu and J. Pan, "Location-aware associated data placement for geo-distributed data-intensive applications," in *INFOCOM*, Hong, Kong, China, pp. 603–611.

[24] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," in *NSDI*, San Jose (CA), USA.

[25] C.-W. Cheng, J.-J. Wu, and P. Liu, "Qos-aware, access-efficient, and storage-efficient replica placement in grid environments," *The Journal of Supercomputing*, pp. 42–63, 2009.

[26] W.-C. Chang and P.-C. Wang, "Write-aware replica placement for cloud computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 656–667, 2019.

[27] Y. Shao, C. Li, and H. Tang, "A data replica placement strategy for iot workflows in collaborative edge and cloud environments," *Computer Networks*, vol. 148, pp. 46–59, 2019.

[28] B. Lin, F. Zhu, J. Zhang, J. Chen, X. Chen *et al.*, "A time-driven data placement strategy for a scientific workflow combining edge computing and cloud computing," *IEEE TII*, vol. 15, no. 7, pp. 4254–4265, 2019.