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

Leonardo Marques Epifânio

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

January 2021

Acknowledgments

Firstly, I would like to thank my advisor, Professor Luís Rodrigues, for welcoming me as his student, and for the guidance and discussions that were crucial to the development of this thesis. I would also like to thank Nivia Quental for her insigths and suggestions.

I want to thank my mother, Cristina Marques, and my father, Paulo Epifânio, for their full support and encouragement throughout my college years. I also want to give a big thank you to the many great friends that I have made in the last few years, with which I was able to grow as a person.

To each and every one of you – Thank you.

Abstract

Placing data replicas in edge nodes is a key strategy to offer low latency to clients and to improve network utilization. In the case of immutable objects, data placement is only constrained by the limited capacity of edge nodes. However, for mutable objects, one also needs to consider the cost of keeping replicas consistent, which then varies with the data consistency model: a replica placement algorithm that performs well for weakly consistent replicas may perform poorly when strong consistency is required. In this thesis, we present Cathode, a replica placement algorithm that is tailored for the requirements of the edge environment, all the while being consistency-aware, making placement decisions based on client demand, storage costs, and the costs of keeping replicas consistent. In the underlying system, different objects may use different consistency models and Cathode makes placement decisions accordingly. Because optimal replica placement is known to be an NP-hard problem, Cathode resorts to an heuristic that is decentralized and scalable, providing fast convergence, but also achieving high quality deployments. The extensive performance evaluation reported in this thesis shows that it outperforms previous state-of-the-art replica placement algorithms.

Keywords

Edge Storage, Data Placement, Replication, Data Consistency

Resumo

A colocação de réplicas de dados em nós na periferia de rede é uma estratégia-chave para prover uma baixa latência aos clientes, e para tornar a utilização da rede mais eficaz. No caso de objectos imutáveis, a colocação de réplicas é apenas limitada pela capacidade de armazenamento dos nós. Contudo, quando se considera objectos mutáveis, também é necessário ter em conta o custo de manter as réplicas consistentes, custo este que varia com o modelo de consistência: um algoritmo de colocação de réplicas que seja eficaz com dados que usam consistência fraca, pode tornar-se ineficaz quando opera sobre um sistema em que é usado um modelo de consistência forte. Nesta dissertação, apresentamos o Cathode, um algoritmo de colocação de réplicas adaptado aos requisitos dos ambientes de execução e armazenamento na perifieria da rede, que efectua decisões que se baseiam em factores como a popularidade dos dados, os custos de armazenamento, e os custos que advêm de garantir a consistência das réplicas. Mesmo que no sistema subjacente sejam suportados múltiplos modelos de consistência, o Cathode é capaz de calcular configurações de réplicas que reduzem estes custos e, consequentemente, melhorem o desempenho do sistema. Sendo que o problema da colocação de réplicas é NP-Completo, Cathode recorre a uma heurística que opera numa estrutura descentralizada e permite uma convergência rápida, mas ao mesmo tempo, calcula soluções que melhoram, em grande escala, o desempenho do sistema. Apresenta-se uma avaliação do sistema, que mostra como o Cathode supera outros algoritmos de colocação de réplicas do estado da arte.

Palavras Chave

Armazenamento em Edge, Colocação de Dados, Replicação, Consistência de Dados

Contents

List of Figures

List of Tables

Acronyms

IoT	Internet of Things
DHT	Distributed Hash Table
PAA	Probabilistic Associative Array
ICN	Information-Centric Network
IaaS	Internet as a Service
QoS	Quality of Service
EMA	Exponential Moving Average
KB	Kilobyte
MB	Megabyte

1 Introduction

Contents

In this dissertation we address the problem of data placement in edge computing, and we study techniques that can be used to place replicas of relevant data on edge devices. We consider the case where objects are mutable, and the costs of maintaining replicas may vary according to the consistency criteria. To the best of our knowledge, previous systems haven't designed a scalable structure that is able to gather information relating to the costs of keeping replicas consistent, and able use this information to reduce such costs. Furthermore, previous replica placement systems for the edge, focus on providing efficiency, at the detriment of utility. This dissertation explores the qualities and limitations of various data placement algorithms from different contexts, and proposes a novel design for a scalable consistency-aware replica placement algorithm for the edge, that provides high utility without sacrificing efficiency.

1.1 Motivation

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 are 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 example of 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. 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. In the simplest case, data objects are immutable and, therefore, there is no significant drawback for creating a new replica of an object on a given edge node, as long as there is local demand and the edge node still has storage capacity. However, for mutable objects, one should also consider the costs associated with keeping the multiple replicas consistent: when a replica is updated, the changes need to be propagated to other replicas; and if the number of replicas is large, this can put a significant burden on the system. Thus, for mutable objects, factors such as the number of replicas and the read/write ratio, become important for optimizing the placement of those replicas.

Although the topic of data placement for edge computing has been addressed before in the literature [4–7], most systems assume that all objects offer the same consistency model, and that this consistency model is enforced by some static, pre-defined, replica consistency algorithm. It is therefore unclear how these algorithms perform when the edge storage uses different consistency criteria. This is unfortunate, as there is a growing interest in designing systems for edge computing that can support multiple consistency criteria [8–11].

This dissertation presents Cathode, a replica placement algorithm for the edge, 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, Cathode resorts to an heuristic that is decentralized and scalable, avoiding the bottlenecks associated to solutions that have a single point of control. The heuristic that was designed is inspired by the one used in D-Rep [4], a decentralized placement algorithm for the edge, however, ours presents substantial improvements. In D-Rep, nodes make decisions based on limited information regarding other nodes in the system; this makes the algorithm scalable but unable to effectively compute a placement solution that yields high utility. Cathode also allows for somewhat decentralized decision making but, since nodes that replicate the same objects are allowed to share their views of where replicas are most required, Cathode is able to aggregate the necessary information to consider the costs of keeping replicas consistent, and yield a higher utility. This way, Cathode is able to approximate (and even surpass) the utility provided by solutions such as Holistic-All [12], which is able to calculate high quality solutions but requires more time to converge in systems of larger size, due to the way it collects information and computes the placement solutions. To the best of our knowledge, Cathode is the only data placement algorithm that provides a scalable operational structure to support multiple replica consistency protocols, without any prior assumption on the underlying topology or on the semantics of the algorithms.

Cathode mainly targets applications such as smart cities, where the data access patterns are likely to exhibit strong geographical locality. For instance, vehicles in a city may access data regarding the traffic conditions in the areas around them or to which they are headed. Vehicles in the same areas will likely access the same data, and depending on their context, may access this data with varying consistency guarantees, as exemplified in [8].

We have extensively evaluated Cathode against the aforementioned solutions, D-Rep and Holistic-All. Our results show that Cathode is able to approximate the convergence speed of Holistic-All for smaller systems, and surpass the convergence speed of D-Rep for larger systems, achieving exceptionally better results in networks of larger diameter, which makes it an interesting approach, even when all objects use the same consistency criteria. Furthermore, while D-Rep and Holistic-All are unable to provide satisfactory results when objects operate under different consistency protocols, Cathode is able to leverage its consistency-aware features to provide high utility in all scenarios. As we have discovered in our experiments, when *strong consistency* models are used, such as *quorum consensus*, while Cathode is able to achieve utility gains of approximately $1.2\times$, the other aforementioned algorithms will, in some cases, even reduce the utility down to $0.6\times$. Aside from this improvement, in some environments, such as large grid networks, Cathode is able to greatly surpass the quality of this type of algorithms in real time, because of its fast convergence, improving the utility by a factor of $1.7\times$, even when using the same consistency model that is directly supported by Holistic-All and D-Rep.

1.2 Contributions

This dissertation analyzes, implements and evaluates strategies to design a consistency-aware replica placement service to operate on the edge of the network. The main contribution of this dissertation is a novel replica placement algorithm that is consistency-aware, and that makes data placement decisions in a decentralized and scalable way, while still achieving high utility and convergence speed.

1.3 Results

This work produced the following results:

- The specification of Cathode, a consistency-aware replica placement algorithm for the edge;
- An implementation of Cathode on the PureEdgeSim [13] simulator;
- An extensive experimental evaluation of Cathode and comparison against two other state-of-the-art replica placement algorithms.

1.4 Research History

This work was developed in the context of the Cosmos research project, that aims at finding techniques to offer causal consistent storage for edge computing scenarios. Techniques to manage the placement of data replicas in edge nodes are expected to be a key component in the final COSMOS architecture.

This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the “Programa Operacional Regional de Lisboa na sua componente FEDER” with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020.

1.5 Organization of the Document

The rest of the document is organized as follows: Chapter 2, firstly, provides a formulation of the problem, with an introduction to key concepts, and secondly, reviews several systems related to our work. Chapter 3 describes the design of the proposed solution, Cathode, specifying the system model, the algorithm and its implementation, also addressing the implementations of the other algorithms considered for evaluation. Chapter 4 details the evaluation of Cathode, presenting the results obtained in the experiments, along with a discussion of the results. Chapter 5 concludes the document, outlining the main findings, and unveiling a possible direction for future work.

2

Background and Related Work

Contents

This chapter delves into the background and definitions of the problem and analyses the related work. In Section 2.1 we explore the characteristics of edge computing environments. Section 2.2 defines the replica placement problem and in Section 2.3 we explore how data consistency relates to the placement problem. Finally, Section 2.4 explore various systems that are relevant to our work, and their limitations are discussed in Section 2.5.

2.1 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 in which the data is stored, processed, and acted upon close to these devices. These paradigms may define different architectures of nodes spanning from the cloud, where datacenters of high computational capacity and storage space are located, to the edge of the network, where weaker devices operate. In [1], several edge computing paradigms are highlighted, out of which, for exemplification, we selected the following:

- *Edge Computing*, in which computations are performed in nodes as close as one hop from IoT devices (data sources), extending their capabilities, in a ubiquitous manner.
- *Fog Computing*, which also decentralizes computation and storage, but presents a multi-tier model, providing progressively increasing resources along a hierarchical succession of nodes, located between the IoT devices and the cloud data centers.
- *Multi-access Edge Computing*, known also as Mobile Edge Computing, which 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*, which 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 the nodes, and how these communicate with one another, according to the architecture, network 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. Seeing that the aim of this work is to define a general replica placement scheme for the edge, we must take these differences into account, by exploring strategies that can be applied to various scenarios; and since these architectures may be composed of a large number of nodes, a replica placement system for the edge should also be scalable, but still provide utility.

2.2 The Replica Placement Problem

Data replication is widely used in distributed systems and it can bring many advantages, such as fault tolerance, load balancing, and lower data access latency. In this work 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.

2.2.1 Replica 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 [14]:

$$C(x) = \frac{1}{\Lambda} \sum_{i=1}^{I} \sum_{j=1}^{J} \lambda_i \boldsymbol{p}_j \boldsymbol{d}_{ij}(x) \tag{2.1}$$

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 item 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. Thus, the replica placement optimization problem should consider both the cost reductions that can be gained by creating a new replica (due to the lower access latency) and the cost increases that result from the additional data propagation that may be required to create and maintain the replica updated.

A more realistic formulation, which accounts for these updates and their costs, can then be expressed as:

$$C(x) = \frac{1}{\Lambda}\sum_{i=1}^{I}\sum_{j=1}^{J}\lambda_i \boldsymbol{p}_j\left((1-\alpha_j)\boldsymbol{d}_{ij}(x) + \alpha_j u_{ij}(x)\right) \tag{2.2}$$

subject to

$$\sum_{j=1}^{J} p_{ij}(x)s_j \leq S_i \quad \forall_i \tag{2.3}$$

where α_j is the write ratio for object j and $u_{ij}(x)$ is the cost associated with propagating the updates performed on object j, by node i, given the data placement assignment x, S_i is the storage capacity of node i, s_j is the volume of object j, and p_{ij} is a boolean that states if object j is replicated on node i under assignment x.

2.2.2 Solving the Optimization Problem

It has been shown that an optimization problem expressed by Equations 2.2 and 2.3 can be mapped to the multiple knapsack problem [14] and is, therefore, NP-Hard. Thus, practical solutions of the data placement problem are solved by heuristics that approximate the optimal solution. The most straightforward way of applying an heuristic is to centralize all the execution and information 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 associated with collecting information regarding the access patterns can be distributed among multiple nodes, along with the execution of the placement decisions.

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 (or equivalently, the utility that the algorithm can achieve within its capabilities).

- *Efficiency*, which is split into the computation and communication overheads. The computation overhead represents the complexity of computation necessary to perform a decision and apply it to the system. 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.

- *Convergence speed*, which defines how fast and effective the scheme is at converging to a stable 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 algorithm adapts the placement of replicas to workloads in which the request patterns change over time.

Each scenario and environment in which a replica placement algorithm may operate, will have priorities in terms of these criteria. For example, in some environments, efficiency is not a problem, and so an algorithm can perform more complex operations to achieve better approximation quality, with less regard for the computation and communication overheads that may arise. It is therefore important to define the criteria that is most important for an edge environment.

Addressing this, given the highly dynamic nature of edge environments [15], we can confidently determine that data placement should be elastic. This means that the placement algorithm must run frequently and thus, it should also be efficient and converge quickly. Alongside, edge systems essentially consist in using many nodes of small capacity, that are placed close to clients, instead of using only a few nodes of large capacity, that are placed far from clients (in large datacenters). Thus, a replica placement algorithm for the edge should be scalable and consider efficiency thoroughly, in order to be prepared to handle a large number of nodes. In previous literature, it is possible to find algorithms that provide high approximation quality, but that are not efficient and have difficulties in adapting to large systems. On the opposite side, there can be found highly elastic algorithms that achieve efficiency, but at the cost of sacrificing quality. We investigated several of these solutions, to find out the best way to design a placement algorithm for the edge. The most relevant ones to our work are explored in Section 2.4.

2.3 Taking Data Consistency in to Account

Ideally, one would like a replicated system to exhibit the same behaviours as a centralized, non-concurrent system, where each operation is executed in isolation, and all operations appear to execute in a serial order that respects external time. This intuitive behaviour is captured by *linearizability* [16]. The algorithm for atomic register emulation on message passing systems presented in [17] (ABD), has shown that it is possible to implement a linearizable register in a non-blocking and fault-tolerant manner. However, ABD requires both read and write operations to update a majority of replicas, which is prohibitively expensive in most settings. More practical implementations require all updates to be performed on a master replica and use locks to ensure atomic visibility [18]. However, even these implementations can be too expensive for most latency-constrained applications, that may be willing to trade consistency for faster data access.

To circumvent the costs of linearizability (and of other *strong consistency* approaches, such as serializability [19]), a wide range of weaker data consistency models have been proposed in the literature. These models are particularly relevant in geo-replicated scenarios such as the ones observed in the edge computing paradigm, because of their higher scalability. An example of such a model is the *causal consistency* model, which has already been efficiently implemented in edge networks by Gesto [20].

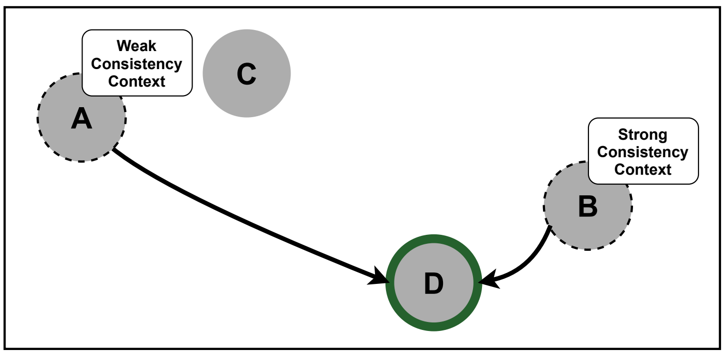
In [21] the authors elaborate on how modern *distributed database systems* (DDBSs) have to perform decisions regarding the trade-off between latency and consistency, and exhibits the differences on how several of these systems handle this trade-off. The authors describe three alternatives in which data replication may be implemented:

1. Updates are sent to all replicas at the same time;
2. Updates are sent to an agreed-upon location first (master replica), and then applied to the rest.
3. Updates are sent to an arbitrary location first, and then applied to the rest.

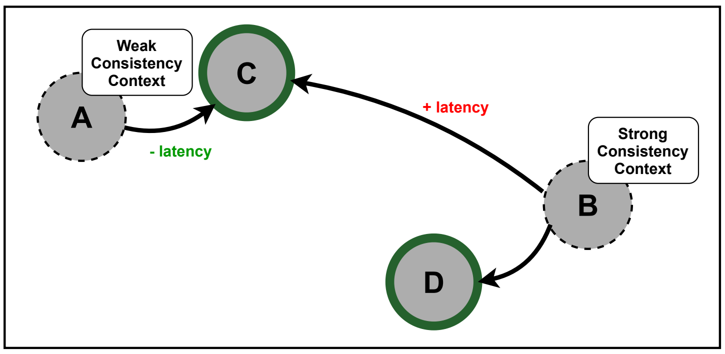
In the case of the second alternative, the replication may be synchronous, when the master replica waits until all replicas are updated, or asynchronous, when the updates are propagated in the background, and the master continues its operation. The system can then route all reads to the master node, which may increase latency, but prevents consistency issues; or the system can route reads to the nearest replica, which reduces latency but, in some cases, results in consistency issues. These approaches are also present in the third alternative, with similar problems.

A combination of synchronous and asynchronous operations can also be used. Consider R as the number of nodes that participate in a synchronous read, W as the number of nodes that participate in a synchronous write, and N as the number of replicas. Configuring R and W such that $R + W > N$, will therefore yield a quorum-like protocol. However, since in quorum protocol the latency of an operation is as high as the latency to communicate with the farthest of the R (or W) replicas, these parameters are often configured such that $R + W \leq N$, opting for a lesser consistency level.

Several modern distributed storage systems, such as Cassandra [22] and DynamoDB [23] use such a combination of synchronous and asynchronous operations. Furthermore, in various failure cases, or due to rerouting, updates of a data item are not always sent to the same node, resulting in a different way by which requests may be propagated, and consequently, also a different latency. Cassandra has been used as the base for the storage module of the Cloudpath [9] 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, in the application level, by having the requests sent to higher tier nodes, instead of the closest (lesser latency) nodes. 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 [10].



(a) Communication between nodes before the creation of a replica in C.



(b) Communication between nodes after the creation of a replica in C.

Figure 2.1: The effects of creating a new replica, on different consistency protocols.

The relevance of supporting multiple consistency models in the paradigm of edge computing has been further validated in [8], which proposes a data storage model that allows for different consistency levels in requests, depending on the context in which a client is inserted. This means that there are several consistency models being used in the system at the same time, and, as is easily noticeable, the different consistency criteria will represent different costs. Consider the example in Figure 2.1, in which, initially, there is a single replica of an object in node D. Two consumers access this replica. Consumer A is in a context which requires weak consistency, and therefore it merely needs to access the closest replica; consider also a consumer B, in a context that requires strong consistency, and therefore it must access a majority quorum of replicas. If a new replica is created in node C, as we may observe in Figure 2.1b, the latency of requests from node A decreases. However, requests from node B will now have to also contact C to have quorum, which increases latency. Thus, in this type of systems, a replica placement algorithm should take into consideration the global improvements/detriments in cost coming from each operation, according to their consistency models, and how they are carried out by the system.

In the work developed by Gao et al. [11], a data replication algorithm is proposed for the edge service architecture, which tunes the consistency of each read/write to achieve the ideal trade-offs between low latency, high availability, and strong consistency. However, selecting the right consistency model for an operation is a problem orthogonal to the topic of this thesis. Furthermore, in most cases, the consistency model is driven by the application semantics. Therefore, we have rather focused on the problem of finding the best placement for the replicas, *given* one or several target consistency models that have been previously chosen and used in the system.

2.4 Replica Placement Systems

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. Along with the criteria we have mentioned previously (Section 2.2.2) regarding the quality of the heuristic, we can also characterize each approach to the replica placement problem, by how the systems themselves operate. For this, the following criteria has been considered:

- *Distribution*, which roughly characterizes where the decisions and computations take place. On one end, we have *centralized* algorithms that perform these actions on a central node, possibly taking into account the whole of the network. On the other end, we have *fully distributed* algorithms, in which each node executes placement decisions in a parallel, with a limited view of the network, which normally results in rougher approximations to the optimal solution, but higher efficiency. In between, there are also *decentralized* algorithms, which decentralize the operation to multiple nodes, but not in a fully distributed way. An example of this type of distribution is *item-wise decentralization*, in which there is a single node performing the placement decisions for each item.
- *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 and dynamic 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.

- *Decision factors*, which are the factors considered when performing the placement decisions. Some of the factors that should be considered in the edge paradigm, are the popularity of requests for each item at each node, the latency between nodes and item replicas and the storage cost of each node. Alongside, we have seen prior, the cost of keeping replicas consistent is also a factor of interest.

There is quite an extensive amount of literature on the data placement problem but most of these system aim at different settings, and despite implementing interesting features, they cannot be easily adapted to edge computing. On the other hand, most systems that were designed for edge scenarios are tailored to specific applications. For instance, [24] defines a replica placement strategy designed for IoT workflows, while [25] defines a strategy for scientific workflows. These types of workflows focus on data processing tasks, and are modelled as dependency graphs which define the data dependencies between these tasks. The placement algorithms then apply strategies that are designed for these scenarios where tasks and data are generated at sensors and IoT devices and have to pass through a set of nodes for processing. As such, they have limited applicability to the general case. In our work we strove to abstract from specific workflows and applications that can be used within the edge paradigm, and focused on building a system that considers the general characteristics of the edge, and performs well within the criteria we defined in Section 2.2.2.

In the next sections we describe several prior solutions that provide interesting approaches to the replica placement problem as we have defined it. Some of these solutions define their process as replica placement, while others define it as cooperative caching. We consider these definitions as analogous and as so, we collocate them in the same category. We do not, however, consider on-demand caching techniques. This type of caching, was shown in [26] to have good performance with content that showcases ephemeral popularity; in this case, where most content is only requested once, a basic *cache on* k^{th} request policy was shown to provide the most benefits that are likely to be achievable in these types of workloads. In spite of this, when we consider workloads from applications that present less ephemeral content, when we consider the costs of storing replicas, the memory restrictions of the edge nodes, and the possible consistency protocols that may operate in the system; it is evident that these on-demand caching schemes have serious limitations, by operating in each node independent of one another. On the other hand, cooperative caching, as we will see in the following examples, allows nodes to collaborate with one another (e.g. sharing information about an object's popularity), which enables the creation of only the necessary replicas to benefit the system, and also enables the accountability of various other factors.

2.4.1 EAD File Replication

The Efficient and Adaptive Decentralized (EAD) File Replication System [27] 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, in a decentralized way.

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, periodically computes the visit rate of its files (query load l_t). If the load surpasses a threshold above the node's capacity, then it decides to create new replicas. First, the node verifies if it received any replication requests from traffic hubs. If so, it orders these requests according to the q_f of each request, and accepts the requests until the predicted load is bellow the threshold. However, if the list of replication requests 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 estimated 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.

By placing replicas in the query paths from requesters to file servers, the scheme allows the usage of a Distributed Hash Table (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 lesser approximation quality, when we consider the 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 time-varying file popularity and node interest. Furthermore, it mostly only requires synchronization in the case of node churn, leading to a low communication overhead. High replica utilization is ensured, but approximation quality is hindered by using replica-blind routing, restricting placement to query hubs. Convergence, however, is fast, since replicas are immediately placed in the best destinations that can be calculated by the algorithm, in a parallel fashion. However, the decision factors are restricted to the file popularity and congestion of the links of each node, not accounting for other factors that influence clients' experience, such as the access costs (latency of the requests).

2.4.2 Congestion-Aware Caching (CAC)

In CAC [28] the authors investigate cooperative cache management 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, if the 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.

First of all, the system makes an assumption that each response to a request, follows the opposite path of the corresponding request. Then, each node keeps track of the number of *flows* (active requests) currently passing over its links, and uses this information for estimating the available link bandwidth for new requests. File popularity is also estimated in each node, for each and every file whose requests pass through the node. These values are then used by the node to calculate the *gain* of replicating each file in that node, which is inversely proportional to the *minimum available link bandwidth* along the upstream path (from the content source to the consumer node). When a content delivery response arrives to the node, it then autonomously decides whether to cache the content, according to the *gain* that can be attained. 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, which allows nodes to forward their requests to replicas in less congested paths.

By accounting for link congestion, this scheme is able to reduce the average delay 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 somewhat aware of replicas, improving the approximation quality of the algorithm.

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.

2.4.3 Age-Based Cooperative Caching (ABC)

In ABC [29] 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, replicating the most popular contents at the network edge, while the least popular are only placed closer to the sources, focusing on the improvement of the average cache hit rate and load reduction at central storage units.

When an object is first requested from its source (origin) node, an initial *age* value is calculated in the neighbour node through which it is routed back to the requester. This value determines the lifetime of the replica, and is proportional to its estimated popularity, which is the observed access frequency in the source node. 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 (which reached the end of its age-defined lifetime), it is replaced by the arriving one. The item then continues to be routed from the source to the requester with its age value piggybacked, and every node through which it passes 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 access costs from nodes to the original items, but this cost is accounted in terms of node hops and not on the actual experienced latency. Another issue relies on the fact that the popularity factor, in the age values, is calculated only in the source nodes, which may result in inaccuracies, where items which have a global high popularity, are replicated, with an high age value, in edge nodes located in regions where they might only be requested a couple of times. This curbs approximation quality, and since nodes suffer from the departure of popular contents, when they reach their expiration date, a stable solution is never reached.

2.4.4 Associated Data Placement (ADP)

In ADP [30], a replica placement system for the cloud, the objective is to account for the associations between different data objects, and improve the co-location of associated data. Two algorithms are modelled, the first not allowing replication, and the second, allowing replication with fixed degrees for each data object. We focus on the study of the second algorithm, being more relevant to our context. As a precursor of this algorithm, the authors in [31] presented a data placement system that iteratively moves an object closer to both its clients and other objects it communicates with, but in contrast with [30], 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, instead of the actual observed latency of the connections between these nodes.

The system model of ADP assumes that geographically distributed nodes may request sets of specific replicas of a data object, from each other. These sets are called *request patterns*. Each placement decision requires global knowledge of these request patterns. The decisions are therefore performed in a centralized unit. In a time period, the request rate of each set (and each 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 set that is not handled by the same node.
- *Localized Data Serving*, whose value is proportional to the number of requests where the requesting node is not the one holding the data.

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 object, the k nodes with the highest request rate are used to store the replicas of the object. After this step, the scheme runs a loop, performing, in each round, a placement decision, but also a routing decision. The routing decision maps, for each request set performed by each node, each request of the request set to the adequate replica nodes. This decision is made considering the request patterns, and the coverage of the sets by each replica 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 the computation overhead that comes from calculating these factors represents a weakness for this algorithm.

The placement decision itself 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 sets/patterns, each assigned a weight proportional to its frequency. Alongside, all the pairings between each node and each replica are also weighted according to request 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. This incurs that several rounds must go by, before reaching the stable solution.

As we can see, this one of the most complex algorithms we have analysed, and although it is somewhat 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 large number of nodes. The algorithm accounts for very interesting decision factors, however the computation overhead to perform each 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.

2.4.5 AutoPlacer

AutoPlacer [32] is a replica placement system, developed for a cluster environment. The system operates solely on the migration of replicas, considering a fixed replication degree. In this work, the authors developed a way to efficiently implement replica-aware routing, which is accomplished as follows: When a replica is migrated, its location is placed in a relocation map. Since the map can grow too large to be efficiently distributed, the authors propose a structure that allows for efficient storage, and fast lookups. This structure is named Probabilistic Associative Array (PAA).

The replica placement scheme itself executes in rounds, and each optimization round consists in the following sequence:

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.

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.

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

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

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, however it 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 a 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). It then comprises a set of rules that take these values from a key, and output its correspondent replica set. The accuracy of this structure is then controlled by a few adjustable parameters, allowing the system administrator to define the trade-off that is most adequate for the underlying system's goals.

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 fully distributed fashion, the fact that every node has to synchronize with each other in each round, represents an overhead in communication, which is hardly a problem in the context the algorithm was developed for, but would affect its performance in other contexts (such as edge computing). The replication degree is fixed, and hence, the replica number is not adjusted, which hinders elasticity. Despite all this, the system proves its success at achieving efficient storage and fast lookups in a replica-aware routing scheme.

2.4.6 Distributed Cache Management in ICNs

In Sourlas et al. [12], the authors devise several distributed cooperative cache management algorithms directed at Information-Centric Network (ICN)s, 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) object replicas to nodes, based on their time-varying request popularity and distance to their consumers.

The network is modelled as a graph of 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 data objects, H as the current configuration, which maps each object to a set of nodes, with h_v^m being a variable that indicates whether object 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 data object, in the system. To achieve this, the authors propose several 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 *Cooperative* algorithm requires each node to maintain a global view of request patterns, and the current configuration. For this, the nodes must perform an all-to-all update each time the algorithm is run. Then, at each iteration, each node v estimates:

- For each object m that it replicates, the global performance loss l_v^m. This value is computed as the cost difference between a new configuration, in which m is removed from v, and the current configuration.
- For each object o that is not replicated in v, the expected global performance gain g_v^o, achieved by caching o in v, is also calculated.

In this system, the caches are always kept full, and thus whenever a replica is created, another is removed. The node then considers the object with maximum performance gain, i, as the candidate for insertion, and the object 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 is in charge of placement. This node then computes the maximum gain placement b, and broadcasts the report message, $Rep(b, v, i, k)$. After receiving this message, each node immediately updates H, and the changes are applied. The process is repeated, sequentially, for each node in the system.

The *Holistic-All* algorithm, on the other hand, is proposed by the authors, from the observation that, more than one replacement per node, in each iteration, can be beneficial to the system by speeding up convergence to a stable solution. Just like in the *Holistic* algorithm, in each iteration, a single node is in charge. However, instead of computing only one placement, the node computes all the beneficial placements, by selecting, from all possible objects in M, the set of candidates that minimize the total traffic cost.

The authors have also defined the *Myopic* algorithm, which differs from all the others in the fact that it assumes that each node v has no information about the demand patterns at the other nodes. Then each node performs placement decisions only with local information, independently from other nodes. The advantage of this approach is clearly the lower computation and communication overhead, with no information collection. However, its entirely local view of request patterns may result in low quality solutions, and high redundancy.

All of the algorithms adapt to dynamic workloads, by working with adaptive replication degrees and constantly gathering information about the varying popularity of objects. The Cooperative and Holistic-All algorithms don't have the problems of the Myopic algorithm, but take a long serial number of communication rounds to optimize all nodes, which hinders convergence speed, especially when the number of nodes is large. The Cooperative algorithm also specifically requires the communication between all nodes after computing the candidates in each of them, resulting in lower efficiency.

After studying this work, we have concluded that Holistic-All seems to be the one achieving the best performance, according to our criteria, despite its high overheads. This algorithm essentially represents a decentralized way to compute high quality solutions, and high convergence speed in small systems, but has scalability problems due to its inefficient operation.

2.4.7 Decentralized Replica Placement (D-ReP)

In D-Rep [4], the authors propose a dynamic algorithm for replica placement in the paradigm of edge computing. This algorithm takes into account the popularity of the data objects, the latency between nodes (access costs) and their storage costs. This algorithm considers all nodes as Internet as a Service (IaaS) providers, and aims at pushing the contents from the cloud to the edge (increasing the proximity of replicas to the data consumers), in a fully distributed and efficient way.

In D-Rep, 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 algorithms of Section 2.4.6), 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 h executes an instance of the placement algorithm, and is aware only of itself and its immediate neighbours. As so, a node h, computes, for each neighbour n of h:

- *latency*$_{nh}$, which is the latency from n to h.
- *unit_price*$_{n}$, which is the unit storage price of each neighbour node.
- *num_requests*$_{knh}$, which is (for each replica k that is stored in the node) the frequency of requests for k, coming from neighbour h.

The algorithm is executed in epochs, where an object may be *duplicated* or *migrated* from node h to a 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, the object is duplicated. If this condition does not hold, the migration condition is tested. For this condition, the algorithm calculates the reduction in storage cost from the removal of the replica in h, and estimates an augment in the latency cost. This augment comes from the assumption that the latency of requests coming from the other neighbours $i \neq n$ will increase, due to the removal of the replica. The estimation of this value is very simple, but lies on an assumption that may not hold true, which may stop the algorithm from performing beneficial migrations. Finally, the algorithm evaluates a condition to *remove* each replica. This is done by estimating, in the creation of the replica, its expected utilization. This value is then compared against the current number requests, and the replica is removed if it reaches a lower-bound.

In D-Rep, placement is fully distributed and executed with very reduced information, displaying a very low computation overhead per node. The algorithm can adapt to dynamic workloads, but its limited horizon of information may lead to the creation of unnecessary replicas and high fluctuation. Routing is only aware of replicas in certain cases, since the authors define a lightweight solution for replica discovery. On the other hand, the fact that, in a single epoch, replicas can only be forwarded from a node to one of its neighbours, means that several epochs are required for a replica to reach its optimal location, which results in a slow convergence.

2.4.8 QoS-Aware Replica Placement

In Tang et al. [33] the authors define a Quality of Service (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 for a single object, and it can then be intuitively extrapolated to a system that operates with multiple objects. Every node v in the network has a QoS requirement $q(v)$, specifying an upper bound on the access costs (latency of requests). These access costs are the distance $d(v, u)$, from v to the closest node u that holds a replica of the requested object. The objective of the QoS-aware placement problem is to find a placement solution that satisfies the QoS requirements of all nodes, while minimizing the *replication cost*. This cost is a weighted sum of the *storage costs* of all placed replicas, and the *update costs*, which refer to the necessary communication to update the replicas. For updates, the model assumes that the network is organized in a tree structure, rooted at the source (origin) node of the object. When a write operation is performed, the request is sent to the source node, which then propagates the updates to every replica, through the aforementioned tree structure. Therefore, the update costs rise with the number of replicas and their latency to the origin.

Building on the previous model, the system proposed in Cheng et al. [34] also considers the storage costs and update costs. Furthermore, this work also takes into account the access costs, which are represented by the distances between the consumers and replicas. The set of consumers served by a replica node u is denoted $SS(u)$, and the sum of the workloads $W(v_u)$ from each consumer v of the replica in node u, must not surpass u's capacity constraint, $C(u)$. The replication strategy is considered feasible if no replica nodes 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*. The algorithm assigns a replica of the object, to each and every node, and subsequently removes each replica, according to the largest possible cost reduction. In the first iteration, the service set $SS(v)$ of each replica node v contains only itself, since the closest replica to v is in v. These sets are then adjusted for every pair of nodes, while maintaining feasibility, considering two separate cases:

- *Case 1:* The replica is removed from node v by shifting all nodes in $SS(v)$ to the service set $SS(u)$ of another replica node u. This is only possible when u is within the QoS range of every node in $SS(v)$, and the workload addition doesn't exceed $C(u)$.

- *Case 2:* A portion of the workload of v is shifted to another replica node u, by moving only some of the nodes in $SS(v)$ to $SS(u)$. While in case 1, replicas are removed according to the reduction in storage costs, update costs, and access costs, in this case the only affected factor is access costs, 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 respects the QoS restrictions, but, until the total cost is impossible to reduce. 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 an item-wise decentralized scheme, where the placement decisions can be performed at the source nodes for each data item. The goal of the algorithm is to satisfy latency constraints, and reduce costs without considering request patterns, therefore it isn't made to support dynamic workloads.since the algorithm calculates the near-optimal solution in a single round, convergence is fast. 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 the authors do not provide information on how synchronization and replica discovery would be implemented, we make no assumptions on the communication overhead.

2.4.9 Write-Aware Replica Placement (WARP)

In [35] the authors propose a replica placement algorithm for the cloud that accounts for the costs of keeping replicas consistent. The algorithm defines the operations of several consistency models and evaluates delay (access costs) and reliability for each possible operation. This algorithm regards the placement of replicas for a single object. The authors detail the calculations that are necessary to perform the placement decisions, but abstain from describing where the calculations should be performed. By omission, we may assume that, for each item, a single origin node is responsible for these calculations, making an item-wise decentralized scheme. The system model assumes several cloud datacenters (DC) each with an arbitrary number of nodes. The data regarding a cloud application A is then replicated among multiple partition groups. Each partition group $g \in G$ has a single leader l_g, and several masters and slaves. m_g then represents a master in g, and s_g represents a slave.

When a write operation W is requested by a client, the request is sent to the closest master, which forwards it to the leader. The leader then transmits the updates to all masters, and each master relays the updates to its slaves, asynchronously. The transmission delay of the W operation for time interval t, from client c near DC_i is then represented by:

$$d^i(W,t) = d(c, m_g, t) + d(m_g, l_g, t) + d(l_g, m_g, t) + d(m_g, c, t) \tag{2.4}$$

where $d(c, m_g, t)$ denotes the delay from the client to its master, $d(m_g, l_g, t)$ the delay from the master to the leader, *etc.* . A read operation, on the other hand, may be performed in multiple ways, depending on the consistency protocol used by the algorithm. The authors consider 3 consistency protocols: Weak Consistency (WC), in which the users simply obtain the object from the closest slave; Strong Consistency (SC), in which the users should get the last version of the object; and Bounded Consistency (BC), in which the latest version is obtained when slaves are not sufficiently up-to-date. For each of these consistency protocols, a read operation is defined, along with the cost function for the transmission delay, much as in Equation (2.4). The total cost function then takes these delay cost functions, along with the number of operations for each consistency protocol, and calculates the average delay for all datacenters. The cost function is then subject to several constraints, which: indicate the required QoS for all operations, limit the number of replicas in a DC, limit the number of slaves to the processing rate (computational capacity) of their masters, and limit the number of masters to the processing rate of their leaders.

Firstly, the authors define a leader selection algorithm, which uses the delay cost functions and the frequency of operations to calculate the best leaders for each partition. Likewise, the replica placement algorithm also uses the delay cost functions to calculate weights for masters and slaves (w_m and w_s), that represent the average delay of all operations that use, respectively, masters and slaves. For example, Weak Consistency read operations only access the closest slave, and so they are only used in the equation to calculate w_s, and not w_m. After these weights are calculated, the DC with the lowest w_s value becomes DC_{slave} and the DC with lowest w_m becomes DC_{master}. Slaves and master replicas are then placed in each of these two DC's according to the number of read and write operations.

This algorithm requires a whole lot of information about the system, regarding the delays between datacenters and the operations performed in each of them. In systems with a large number of nodes, this is less feasible and may lead to larger storage and computation overheads, since we may assume, by omission, that the calculations regarding the placement of each replica are not computed in parallel, but performed in a single *origin* node of that replica. Convergence is somewhat fast, since the best achievable solution is calculated in a single round. However, the algorithm only considers replica creation, disregarding replica removal, since, as the authors note, this would add additional complexity. This makes it much less elastic, since replicas that become unnecessary are not removed. Routing is replica-aware, and we make no assumptions on the communication overhead, since the authors do not specify how all the necessary information to execute the algorithm, is made available.

2.5 Discussion

The aforementioned schemes were each developed for different systems, and considering different goals. Each of them succeeds in some factors and flounders in others. In Table 2.1 we can see a comparative overview of these schemes, where we lay out how well each of them performs according to the performance criteria, and how the systems differ from each other, according to their characteristics. Performance Criteria is classified by: *High Performance* (★★★), *Medium Performance* (★★), and *Low Performance* (★).

Placement schemes that use replica-blind routing, such as [27, 29] are very efficient, but are restricted to placing replicas along the request paths, hindering approximation quality. EAD [27] also only considers the popularity of data objects, and node congestion, not accounting for factors such as the experienced latency from the data consumers to the data objects (or, equivalently, the access costs). ABC [29] considers the distance between consumers and data sources, but its mechanisms may perform inaccurate estimations, and convergence may be impossible due to the constant removal decisions that may come to remove useful replicas. CAC [28] attempts at maintaining efficiency, while improving quality, and accounts for network congestion, however, it still places replicas along request paths, which prevents the scheme from achieving a significantly more optimal solution. Along with that, it also suffers from weak elasticity, since a replica may be kept indefinitely on a node, regardless of its usefulness.

To deal with the inefficiencies of replica-aware routing, AutoPlacer [32] develops a probabilistic structure to store and query the information efficiently. However, the replica placement scheme itself requires

Placement Systems	Performance Criteria					System Characteristics			
	Approximation Quality	Comp. Overhead	Comm. Overhead	Convergence Speed	Elasticity	Distribution	Replication Degree	Routing	Decision Factors
EAD [27]	★★	★★★	★★★	★★★	★★★	Decentralized (item-wise)	Adaptive	Replica-blind	Object popularity, Node congestion
CAC [28]	★★	★★★	★★★	★★★	★	Fully distributed	Adaptive	Mixed	Object popularity, Link congestion (used bandwidth)
ABC [29]	★	★★★	★★★	★	★★★	Fully distributed	Adaptive	Replica-blind	Object popularity, Consumer-source distance (hops)
ADP [30]	★★★	★	★	★	★★	Centralized	Fixed	Replica-aware	Object popularity, Co-location of associated data
AutoPlacer [32]	★★★	★★	★	★★	★★	Fully distributed	Fixed	Replica-aware	Object popularity
Holistic-All [12]	★★★	★	★	★★	★★★	Decentralized	Adaptive	Replica-aware	Object popularity, Access costs
D-Rep [4]	★★	★★★	★★★	★	★★★	Fully distributed	Adaptive	Mixed	Object popularity, Access costs, Storage costs
Cheng et al. [34]	★★★	★	N/A	★★★	★	Decentralized (item-wise)	Adaptive	Replica-aware	Access costs, Storage costs, Update costs, Load balance, QoS
WARP [35]	★★	★	N/A	★★★	★	Decentralized (item-wise)	Adaptive	Replica-aware	Access costs, Update costs, Consistency models, Node processing rate, QoS

Table 2.1: Comparative overview of the existing replica placement schemes.

an all-to-all communication process, and the replica degree is fixed, limiting its elasticity to dynamic environments. ADP [30] computes near-optimal solutions, but it is centralized, requiring very complicated computations, and the collection of data from all nodes; which makes it infeasible in the edge computing paradigm. The fact that the algorithm considers the co-location of associated data, is very interesting, however, it requires a somewhat global view of the system. Because of this, it is hard to develop an efficient solution for this cost model, while achieving high approximation quality.

Holistic-All [12] is distributed, has high elasticity and represents one of the best alternatives to achieve high approximation quality, but not without sacrificing efficiency. This is because it requires a long serial number of communication rounds, which is a big problem in systems with a high number of nodes, and in a large diameter networks (where the latency between nodes is high). On the other opposite, D-Rep [4] trades approximation quality for efficiency. This algorithm has been designed specifically for the edge and requires very little communication among nodes, however, the approximation quality is hindered by the fact that each node works with partial information, and, since it requires multiple optimization rounds to approximate the global optimum, convergence is also, consequently, hindered.

As we have mentioned, one of the main causes of the development of edge systems, is to reduce the delay from consumers to content. Thus, the aim of a placement scheme for the edge is always to place data objects closer to the nodes that request it the most, which depends on factors such as object popularity, and distance from consumers to replicas. This distance can be defined as the experienced latency, or access costs, hence, we can affirm that both of these factors (object popularity and access costs) are accounted for in [4, 12, 34, 35]. However, only one of these strategies considers the possibility of differential consistency in requests. Cheng et al. [34] considers update costs, associated with the write operation, but it assumes a specific topology and routing algorithm. WARP [35], on the other hand, accounts for the costs of multiple consistency models, independent of routing, but requires a very large amount of information regarding the network and the state of each node, along with extensive calculations. Since the authors do not provide a specification for how to obtain the information and effectively distribute the calculations, the algorithm presents difficulties in its application to the edge paradigm (and the same can be said for Cheng et al. [34]). Alongside, it only considers replica creation and disregards replica removal, which thwarts its elasticity.

To the best of our knowledge, there are no replica placement systems for the edge (nor for other contexts) that provide a viable and scalable structure to compute placement solutions that account for the costs of keeping replicas consistent, according to multiple consistency protocols. D-Rep, being one of the most relevant replica placement algorithms for the edge, operates in a fully distributed structure, in which each node has a very limited horizon of information. This design makes it difficult to extend the system to consider the costs of different consistency protocols, which would require the nodes to fetch more information. In this thesis we have developed Cathode, an algorithm that provides an operational structure that is able to gather and use the necessary information to perform placement decisions that consider these costs, in a scalable and flexible fashion. Cathode operates in an item-wise decentralized fashion, just like [27, 34, 35], and provides several parameters that allow the system administrator to customize the system's operation to the priorities of the underlying system. Cathode also highly focuses on providing fast convergence (which is a necessity in a dynamic environment such as the edge), while attempting to balance quality and efficiency.

Summary

This chapter has provided the background and definitions of the replica placement problem. We have explored how the problem can be applied to the edge and explored the importance of considering the multiple consistency protocols that can be used in these systems. We have described several replica placement systems that present relevance to our work, highlighting their operation, capabilities, and limitations, and finally, we have explained how Cathode places itself in this world. In the next chapter we delve into the design and implementation of Cathode.

3

Cathode

Contents

This chapter introduces Cathode, a decentralized consistency-aware replica placement algorithm for the edge. In Section 3.1 we present the design goals for the algorithm. Section 3.2 describes the system model and Section 3.3 explains how Cathode supports the consistency models, and defines their operations. Then, Section 3.4 showcases the cost function and decision factors that the system accounts for, and Section 3.5 delves into the algorithm that aims to minimize the cost function. In Section 3.6 we present an analysis of the algorithm, its problems and applicability, and finally, in Section 3.7 we provide a description of the implementation of Cathode that was used in our evaluation.

3.1 Goals

As we have seen in the previous chapter, in the area of replica placement, some systems have focused on improving approximation quality, while sacrificing efficiency, such as Holistic-All [12], while others opted for efficiency at the detriment of quality, such as D-Rep [4]. The latter algorithm is one of the most relevant attempts at building a general replica placement system for the edge/fog paradigm, however, it makes use of a very small horizon of information. When designing Cathode, one of the goals was to broaden the horizon of information, in order to have better approximation quality, but without completely sacrificing efficiency. This is accomplished by having a near global, but approximate, view of the problem. Alongside, the overall structure of Cathode shall allow it to support different consistency models, adapting the placement of replicas according to the observed latencies of each operation. The objective is to distribute the extensive calculations required to account for these factors, all the while considering both the creation and removal of replicas, providing the necessary elasticity for it to adapt to dynamic changes in the request patterns.

3.2 System Model

This section presents a description of the system model, detailing how the nodes are organized in the network, and how objects are deployed.

3.2.1 Node Network

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 *capacity*(n_i), and a unit storage cost *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.

3.2.2 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 *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*$_i^k$; and each operation may have a different cost, according to its semantics (whether it is a read operation or a write operation, which consistency model it supports, etc) and to the current data placement.

3.2.3 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, awaiting for the response without observing the inner processes of the system. Each request is characterized by a target object o_i and the specific operation op^k performed on that object. Since the replica placement system only cares about *which* objects are requested by *which* nodes, we do not model the inner parameters of the operations, which are application-dependent.

To give an example of how a request is handled, consider the case of a simple read operation. If the object is replicated on the edge node to which the request was submitted to, the edge node performs the operation and replies to the client. Otherwise, the edge node forwards the request to some other node that replicates the object, collects the reply, and forwards it back to the client.

Since we assume that clients are attached to the closest edge nodes, we can abstract the system model from clients, and think of requests as emerging in nodes. These requests can then be satisfied in those same nodes, or may require communication and cooperation with other nodes, depending on the current placement of data, and on the consistency protocol being used. This way, Cathode is only aware of the set of communicating edge nodes, N.

3.2.4 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.

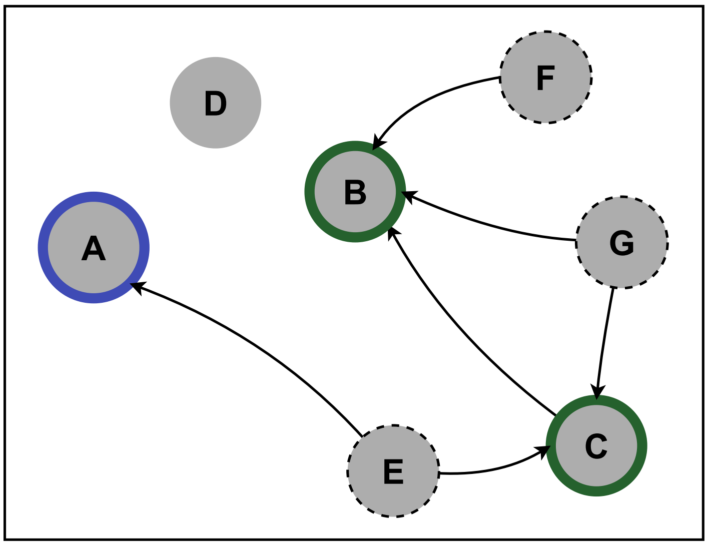


Figure 3.1: Example of a deployment for a data object.

We assume that for each object o_i there is a single edge node that serves as a *source* replica for the object, denoted *src*(o_i). The full set of edge nodes that keep a replica of a given object o_i, including its source, is denoted *replicas*(o_i). The replica of object o_i in node n is denoted *replica*(o_i, n). The edge nodes that perform requests for given object, are denoted the *consumers*(o_i). The deployment of an object is a tuple defined by the identifier *id* of the source and the set of replicas:

$$\textit{deployment}(o_i) = (\textit{id}(\textit{src}(o_i)), \textit{replicas}(o_i)) \tag{3.1}$$

In Figure 3.1 we can see the example of a deployment, which node A is the source node, nodes B and C, with the green outline, are the other replicas, and the nodes with a dotted outline (F, G, and E), are the consumers of the data object, which perform operations in either the source or any of the other replicas. In the context of the edge/fog, the placement of the *src* can be done in multiple ways, depending on both the underlying architecture, and the application that is running. For example, it can be thought of as the node in which geographically-local data originates. In scenarios like vehicular networks (as presented in [36]), road-side units store the information regarding road hazards in the area. These road-side units, in our model, can be considered the *src* of the data regarding those regional road hazards. This is called georeferencing. Another example, in hierarchical fog networks such as the layered architecture present in [7], the *src* of several objects, relevant to a region, can be a more reliable higher tier node, with greater computation and storage capacities. This said, the algorithm used to assign the source node to a given data object is, hence, orthogonal to our work. Currently, we assume the source does not change its position in the run time of the system. Cathode can be extended to dynamically re-assign the source of an object based on current demand. However, extensions to support such tasks are not discussed in the paper and are postponed for future work, since they add extra complexity to the problem.

3.3 Operations and Consistency Models

As we have stated, each data object o_i supports a set of operations, and these operations are often attached to consistency models. For example, if an object supports a weak consistency model, it should support a read operation and a write operation that guarantee weak consistency.

Each operation on an object o_i, as it is evident, will differ in cost, according to both the semantics of the operation, and the current deployment of o_i. We therefore assume that the cost of a given operation $\textit{op}_i^k$ on object o_i, when it originates in a given edge node n, can be expressed as a function of n and of the object deployment $\textit{deployment}(o_i)$:

$$\textit{cost}(\textit{op}_i^k, n) = \textit{costfunction}_i^k(n, \textit{deployment}(o_i)) \tag{3.2}$$

The $\textit{costfunction}_i^k$ thus allows to model the costs of different replica consistency protocols, and the cost may be modelled as a range of different factors (e.g. latency, network congestion, number of communication messages, etc.), depending on what the system administrator wants the replica placement system to optimize.

3.3.1 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 also an object that supports a *primary-backup consistency* protocol, where the source of the object plays the role of the primary replica. To support this protocol, we define a write operation named WRITESOURCE and a read operation named READCLOSEST.

The WRITESOURCE 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 WRITESOURCE operation can be captured by the following cost function:

$$\textit{wcost}(n, \textit{deployment}(o)) = 2[\delta(n, \textit{src}(o)) + \max_{\forall j \in \textit{replicas}(o)} \delta(\textit{src}(o), j)] \tag{3.3}$$

The READCLOSEST 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-CLOSEST operation can be captured by the following cost function:

$$\textit{rcost}(n, \textit{deployment}(o)) = 2 \min_{\forall j \in \textit{replicas}(o)} \delta((n, j) \tag{3.4}$$

Note that an object may allow several consistency models, and thus, several types of operations, each with their own semantics. It will then be the role of Cathode to optimize the placement of that object, accounting for the observed costs of each operation, according to their cost functions. In Section 3.7 we detail an implementation of Cathode, and provide more examples of operations and their cost functions.

3.4 Optimal Placement

In this section we describe the problem and the total cost function. The main objective of the replica placement will be to reduce the value of the total cost. We assume that each node n, in each epoch, keeps track of the frequency $f_n(\textit{op}_i^k)$ of each type of operation $\textit{op}_i^k$ that originates in it (i.e., requests received from attached clients).

In order for the replica placement system to estimate these values, time is divided in intervals of the same duration, named *epochs*. The f_n value can then be estimated by using an Exponential Moving Average (EMA) technique, such as the one employed in [27]. In this technique, the f_n value of a given epoch e will then be the weighted sum of the recorded value in epoch e and the f_n value computed in epoch $e-1$ (the previous epoch). A larger weight for the value recorded in epoch e means that the newest observations of the request patterns have larger influence, while a larger weight for the f_n value computed in $e-1$ means that the history of observations has larger influence on the current f_n value.

The cost of maintaining a given deployment for an object o_i is defined as:

$$\textit{cost}(\textit{deployment}(o_i)) = \textit{oCost}(\textit{deployment}(o_i)) + \textit{sCost}(\textit{deployment}(o_i)) \tag{3.5}$$

with *oCost* representing the operations' cost:

$$\textit{oCost}(\textit{deployment}(o_i)) = \sum_{\forall_n} \sum_{\forall_k} \textit{cost}(\textit{op}_i^k, n) \cdot f_n(\textit{op}_i^k) \tag{3.6}$$

and *sCost* representing the storage cost:

$$\textit{sCost}(\textit{deployment}(o_i)) = \sum_{\forall_n} h_i^n \cdot \textit{storageCost}(n) \cdot \textit{vol}(o_i) \tag{3.7}$$

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:

$$\textit{totalcost} = \sum_{\forall_i} \textit{cost}(\textit{deployment}(o_i)) \tag{3.8}$$

Building up from the definitions above, the optimal placement is a set of object deployment solutions that would minimize the *totalcost*, while respecting the capacity constraints at each node. The capacity constraint can be expressed as:

$$\forall_n : \textit{capacity}(n) \geq \sum_{\forall_i : n \in \textit{replicas}(o_i)} \textit{vol}(o_i) \tag{3.9}$$

Since this optimization problem is NP-Hard, as we have mentioned in Section 2.2, we resort to an item-wise decentralized heuristic strategy to approximate the optimal solution, which we describe in the next section.

3.5 Data Placement Algorithm

On a first level, we decentralize the placement algorithm, by letting the source of each object be in charge of the decisions regarding the deployment of that object. On a second level, the calculations needed for those same decisions, are distributed among the replicas and consumers of the object. The algorithm operates in epochs. During the course of an epoch, information is gathered in these nodes, and then, at the end of the epoch, an optimization round commences. The replicas then send their information to the source of the object, which then operates the placement process that focuses mainly on two sequential phases: a *shrinking phase*, in which unnecessary/disadvantageous replicas are removed; followed by an *expansion phase*, in which replicas are created in the positions which bring more benefit to the system. In the following sections we provide a detailed description of this process. As a companion of the text, we also provide an illustration of the process in Figure 3.2.

3.5.1 Information Collection

As we have stated, the algorithm operates in epochs, and in each epoch, each node n estimates $f_n(\textit{op}_i^k)$ for each operation $\textit{op}_i^k$ on object o_i, that has been requested by that node. Hence, n is considered a consumer of each object o_i (i.e. $n \in \textit{consumers}(o_i)$).

Alongside, the node n also computes two data structures for each object o_i that it replicates (each object o_i for which $n \in$ *replicas*(o_i)). These structures are called the *candidateSet*, and the *summaryTuple*.

3.5.1.A *candidateSet*

The *candidateSet* of *replica*(o_i, n) is a set of k_{cand} consumers of that replica, that are considered the best candidates in which to create a new replica of o_i.

To compute this set, the replica node n needs to estimate, for each consumer m, the following values:

- $a_n(o_i, m)$, the frequency of requests from consumer m, regarding object o_i, that n satisfies (or equivalently, the *access frequency* of m to *replica*(o_i, n));
- $\overline{\delta}(m, n)$, the average observed latency of m to n.

These values are updated at every access to the replica of o_i in n. In the general case, each time *replica*(o_i, n) receives a request from a node m, $a_n(o_i, m)$ is incremented, and the observed latency is computed into the average value $\delta(m, n)$. For each m, both values are then used to compute a weight, $w_n(o_i, m)$, that represents a predicted benefit for creating a replica in m:

$$w_n(o_i, m) = a_n(o_i, m) \cdot \delta(m, n) \tag{3.10}$$

The *candidateSet* of *replica*(o_i, n) is the set of tuples $(m, w_n(o_i, m))$ of the k_{cand} consumers with the largest $w_n(o_i, m)$.

3.5.1.B *summaryTuple*

The *summaryTuple* of the replica of o_i in n (*replica*(o_i, n)) provides a summarised representation of the consumers of *replica*(o_i, n). To compute this set, the replica node n needs to calculate, for each consumer m, the following values:

- $a_n(o_i)$, the frequency of requests that it satisfies for that object, or equivalently, the *access frequency* to *replica*(o_i, n);
- *consumers*(o_i, n), the set of consumers that accessed *replica*(o_i, n), in the current epoch;
- $\overline{\delta}_n(o_i)$, the average latency of the set *consumers*(o_i, n) to n.

Each time *replica*(o_i, n) receives a request, the identifier of the consumer node is added to the set *consumers*(o_i, n). The $a_n(o_i)$, and $\overline{\delta}_n(o_i)$ values, on the other hand, are computed, respectively, by summing up the values $a_n(o_i, m)$, and computing the average of the values $\overline{\delta}(m, n)$, which we have described above (Section 3.5.1.A). These values are then used to compute a weight $w_n(o_i)$ that represents the predicted weight of the set *consumers*(o_i, n) on the total cost of the deployment:

$$w_n(o_i) = a_n(o_i) \cdot \overline{\delta}_n(o_i) \tag{3.11}$$

The *summaryTuple*(o_i, n) is then composed by the set of consumers and the predicted weight of the set:

$$\textit{summaryTuple}(o_i, n) = (\textit{consumers}(o_i, n), w_n(o_i)) \tag{3.12}$$

3.5.1.C Sending the information

After both the *candidateSet*(o_i, n) and the *summaryTuple*(o_i, n) are computed during the course of an epoch, they are sent, at the end of the epoch, from n to *src*(o_i). This is the Step 1, represented in Figure 3.2a. Once the *src*(o_i) receives this information from all the replicas, it coordinates the replica placement algorithm, starting with the candidate selection process.

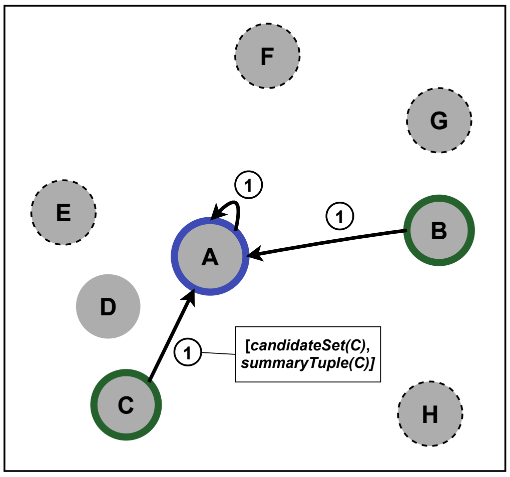
3.5.2 Candidate Selection

As stated before, the placement process is constituted by a shrinking phase, in which replicas are removed, and an expansion phase, in which replicas are created. Consider an object o_i. In both phases several nodes are considered to remove/create replicas, and various combinations of removals/creations are evaluated in the process. In the shrinking phase, the nodes that will be considered in the computations are the ones belonging to *replicas*(o_i), which is expected to be a relatively small set of nodes. In the expansion phase, the nodes that will be considered in the computations belong to *consumers*(o_i). Since it is expected that *consumers*$(o_i) >>$ *replicas*(o_i), we have devised a way to reduce the number of nodes considered in the expansion phase, in order to reduce the computation overhead.

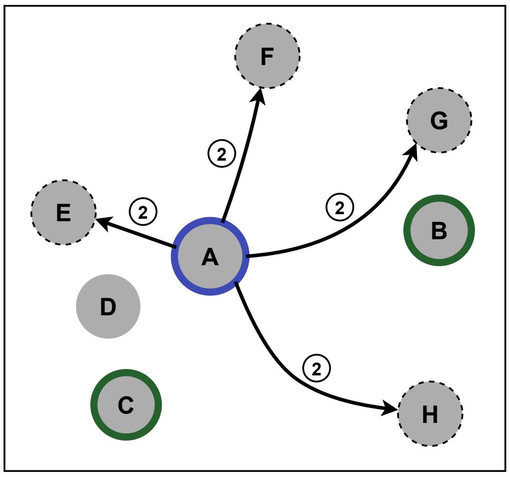
This is done by the $src(o_i)$, which takes the *candidateSet*(o_i, n) of each replica node n, and calculates the global k_{cand} consumers with the largest weights. The resulting set of consumers contains the candidates that will be considered for replica creation, and is denoted by *candidateSet*(o_i).

3.5.3 Shrinking Phase

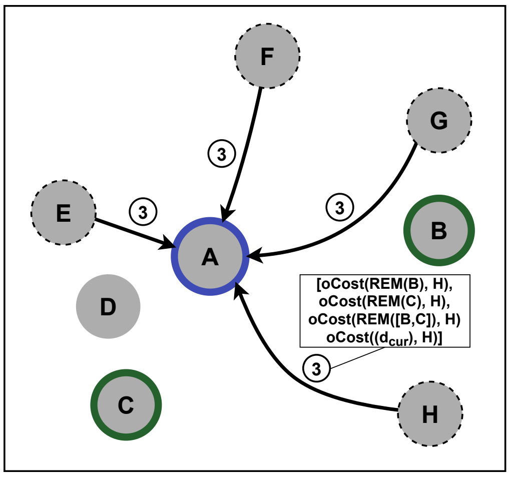
The goal of the shrinking phase is to check if it is possible to reduce the cost of the current deployment by removing one or more replicas. It does so by estimating, for every combination of k_{comb} (≥ 1) replicas of o_i (from the set *replicas*(o_i)), the cost of an alternative deployment where that combination of replicas



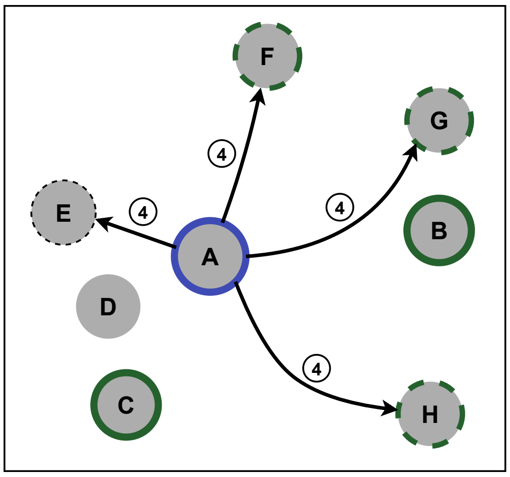
(a) Step 1: Source collects information from replicas.



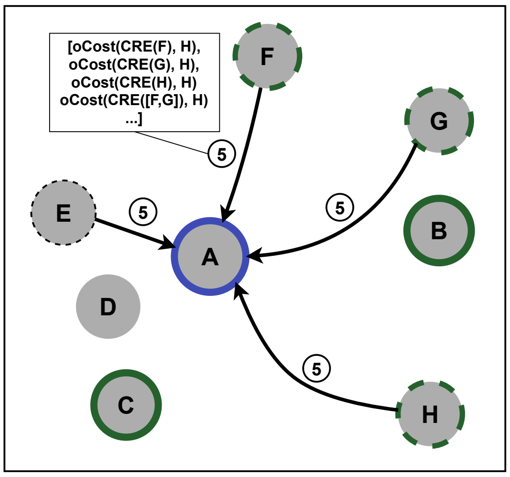
(b) Step 2: Source delegates computations to participants (shrinking phase).


F
G
3
E
3
3
A
B
D
[oCost(REM(B), H), oCost(REM(C), H), oCost(REM([B,C]), H) oCost((d_cur), H)]
3
C
H

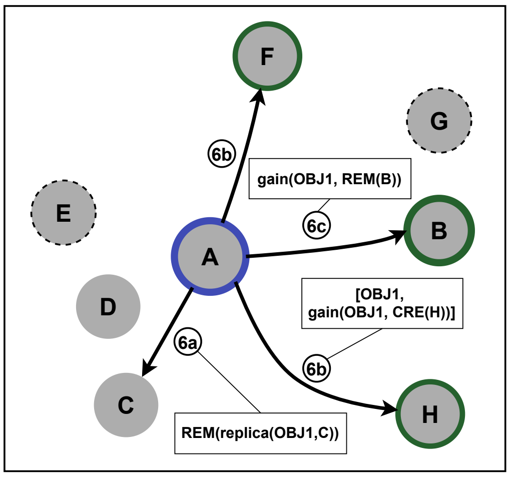
(c) Step 3: Participants return results to source (shrinking phase).



(d) Step 4: Source delegates computations to participants (expansion phase).


[oCost(CRE(F), H), oCost(CRE(G), H), oCost(CRE(H), H) oCost(CRE([F,G]), H) ...]
F
G
5
E
5
5
A
B
D
5
C
H

(e) Step 5: Participants return results to source (expansion phase).



(f) Step 6: Optimized deployment is applied to the system.

Figure 3.2: Example of the execution of the algorithm, for an object *OBJ1*.

is removed. The *gain* that can be achieved by selecting a target deployment, is the difference of the cost of the current deployment and the estimated cost of the target deployment.

For example, in Figure 3.2 the current deployment has 3 replicas: A, B, and C. Considering $k_{comb} = 2$, an alternative deployment is achieved by removing replica B, (*REM*(B)), another by removing replica C, (*REM*(C)) , and the last by removing replicas B and C, (*REM*$([B, C])$). The removal of the source is not considered since it is the node coordinating the placement decisions.

Cathode, after estimating the gain $gain(o_i, d_x(o_i))$ for each alternative deployment d_x of the object o_i, will select the target deployment with the largest estimated gain and, if the estimated gain is above some minimal threshold T_{gain}, that deployment is selected, by removing the corresponding replica(s).

However, in order to avoid overloading the *src* when estimating the costs/gains, we parallelize the computation of these costs by distributing it across the nodes of a set *participants*(o_i). For now let us assume that the set of participants is equal to the set of consumers, therefore, *participants*$(o_i) =$ *consumers*(o_i). The set of consumers is known by the source, from computing the union of the sets *consumers*(o_i, n). (Note: these sets are obtained from the *summaryTuple*(o_i, n) that the source received from each replica n)

$$\textit{consumers}(o_i) = \bigcup_{n \in \textit{replicas}(o_i)} \textit{consumers}(o_i, n) \tag{3.13}$$

In Section 3.5.7 we detail how the participant set can be reduced, but for now, let us assume that this is how it is calculated. After computing the set, the *src* will contact the nodes in *participants*(o_i), as seen in Figure 3.2b. Note that, normally, the replica nodes of an object are also consumers of that same object, however, to simplify our illustration, in Figure 3.2 we assume that both replicas B and C are not consumers of the object.

After the participants receive the message from the source, they will each calculate their part of the *oCost*$(d_x(o_i))$ for each alternative deployment $d_x(o_i)$. Note that the formula of the *oCost* (seen in Equation 3.6), defines it as the total sum of the costs of the operations requested by each consumer. Therefore, it is possible for the consumers to locally compute their own contribution to this cost function, and then send it to the *src*, which will aggregate the results.

As an example, consider node m as a consumer of object o_i. As we have stated in Section 3.4, node m will have estimated the frequency values, $f_m(\textit{op}_i^k)$. Consider that, in an epoch, this value is 4 for READCLOSEST (RC) operations and 2 for WRITESOURCE (WS) operations. Node m will then have to compute the following *oCost* value for each alternative deployment d_x of object o_i:

$$\textit{oCost}(d_x(o_i), m) = \textit{costfunction}_i^{\text{RC}}(m, d) \times 4 + \textit{costfunction}_i^{\text{WS}}(m, d) \times 2 \tag{3.14}$$

Each participant, after computing the partial *oCost* values for each alternative deployment $d_x(o_i)$, and computing the partial *oCost* value for the current deployment $d_{\textit{cur}}(o_i)$, will send these values to the source node (Step 3 in Figure 3.2c). The source will then aggregate the values and estimate the total operation cost *oCost*$(d_x(o_i))$ of each deployment:

$$\textit{oCost}(d_x(o_i)) = \sum_{\forall m \in \textit{participants}(o_i)} \textit{oCost}(d_x(o_i), m) \tag{3.15}$$

This cost, is summed up with the total storage cost *sCost*$(d_x(o_i))$, giving us the *cost*$(d_x(o_i))$, as seen in Equation 3.5. The source then takes these values, and compares them to *cost*$(d_{cur}(o_i))$, computing the *gain*$(o_i, d_x(o_i))$ for each deployment. The largest gain deployment is picked (if it is above the minimal threshold $T_{\textit{gain}}$) and denoted *shrinkedDeployment*(o_i). Afterwards, the expansion phase commences.

3.5.4 Expansion Phase

The goal of the expansion phase is to check if it is possible to reduce the cost of *shrinkedDeployment*(o_i), 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 deployments that, in this case, include 1 to $k_{\textit{comb}}$ more replicas than *shrinkedDeployment*(o_i). These new potential replicas are picked from the *candidateSet*(o_i), which was computed as described previously in Section 3.5.2. In the example of Figure 3.2d, we represent the candidates with a green outline.

The expansion phase is then executed in a way that is analogous to the shrinking phase. The *src* contacts the participants (Step 4, in Figure 3.2d), the costs are computed in a distributed way, sent to the *src* (Step 5, in Figure 3.2e), which then estimates the gains for each deployment, selecting the deployment with the highest gain, if the estimated gain is above the minimal threshold $T_{\textit{gain}}$. This deployment is denoted *optimizedDeployment*(o_i).

3.5.5 Applying the Deployment

After the expansion phase, the replica removals and creations are applied to the system. The source checks the divergence between the current deployment $d_{\textit{cur}}$ and *optimizedDeployment*(o_i). For removals, a message is sent to the corresponding replica nodes, to remove their replicas of o_i (Step 6a in Figure 3.2f). For creations, the messages, which are sent to the new elected replica nodes, include a copy of o_i and a *gain*$(o_i, d_{cre(n)})$ value (Step 6b in Figure 3.2f). This gain value is not of the actual deployment that was picked by the *src*, but of the deployment d_n which creates a single replica in the corresponding node n. (Note that, since $k_{\textit{comb}} \geq 1$, these values are always calculated). This value represents, essentially, the *gain* of creating this replica, and its use is explained in Section 3.5.6.

Along with contacting the nodes in which replicas are removed and the nodes in which replicas are created, the source, at the end of an optimization round, also contacts the other remaining replica nodes.

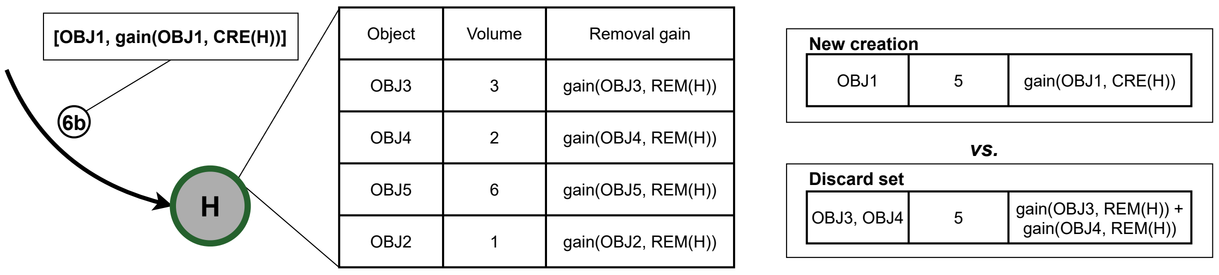


Figure 3.3: Creation of an object in a node with full storage.

It sens to each remaining replica node n, a message containing the gain of removing the corresponding replica, *gain*$(o_i, d_{rem(n)})$. This is seen in Step 6c of Figure 3.2f. These gain values, of course, are expected to be low, or even negative (depending on the T_{gain} parameter), and their use is explained in Section 3.5.6.

3.5.6 Storage Constrains

When a target node receives, from the source node of a given object o_i, an instruction to create a replica for that object, the target starts by checking its storage constraints. If the creation of the new replica would exceed its storage capacity, the target node needs to do the following additional checks before creating the replica: The target node sorts all local replicas according to their associated *removal gain* value (defined in the previous section (Section 3.5.5)); Starting from the lowest gain replica, it puts replicas in a *discard set*, until it has released enough space for the new replica that needs to be created; Then, the target computes the gain (which is probably negative) of removing the discard set:

$$gain(REM(discard\_set(n)) = \sum_{\forall o_j : discard\_set(n)} gain(o_j, d_{rem(n)}) \tag{3.16}$$

This value is compared with the gain that the new replica brings, *gain*$(o_i, d_{cre(n)})$. The new replica is only created if the estimated gain of the replacement is above the threshold T_{gain}. This process is represented in Figure 3.3, where node H receives a creation request, but its storage is full. The volume of the new object is 5 units, and so the discard set also constitutes a volume of 5 units, and is composed of two objects. The algorithm will compare the gains of the new creation with the gains of the discard set and act accordingly.

3.5.7 Paticipant Reduction

According to what was stated in the prior sections, for the placement of an object o_i, in both the shrinking and expasion phases, *src*(o_i) contacts the nodes in the set *consumers*(o_i). However, in some scenarios, the number of consumers for an object may be quite large, and contacting each and every one of these consumers represents a communication overhead. To prepare Cathode for these scenarios, we have devised an optimization, with the goal of reducing the number of consumers that must be contacted in these phases.

This is done by taking each *summaryTuple*(o_i, n), and picking the consumer sets with the largest weights. Then, instead of the source node contacting every set *consumers*(o_i, n), from each and every replica node n, it only contacts the sets with the largest weights.

Let $w_{\textit{total}}(o_i)$ be the sum of the weights $w_n(o_i)$ of each and every set *consumers*(o_i, n) (these values are obtained from the respective *summaryTuples*). Let *participants*(o_i) be the final set of consumers to be contacted by the source. Let $w_{\textit{part}}(o_i)$ be the sum of the weights in the set *participants*(o_i). The final set is computed by adding the *consumers*(o_i, n) sets with the largest weights into *participants*(o_i), until the following condition is verified:

$$w_{\textit{part}}(o_i) > T_{\textit{part}} \cdot w_{\textit{total}}(o_i) \tag{3.17}$$

In this equation, $T_{\textit{part}} \in [0, 1]$ is a parameter which regulates the amount of participants in the placement computations. Lower values lead to less participants, thus, less nodes to be contacted and more efficiency, but fewer approximation quality. Higher values, on the other hand, lead to lesser efficiency, but greater approximation quality.

Note that there can be alternative ways to reduce the participants in the algorithm, by calculating w_h differently. In our design, the weight is calculated as a factor of request frequency and latency, so we cut the consumers that have less predicted weight on the total cost equation, which means they are less important for achieving high approximation quality. If the objective was to cut the consumers that are located farthest from the source, then w_h could be calculated merely as an inverse of the latency of the consumers to the source. This is a completely valid approach which may potentially improve the time efficiency of both the shrinking and expansion phases, since the farthest nodes from the source would not need to be contacted. However, approximation quality could be impaired, since these nodes can have a large influence on the total cost. In our design of Cathode, we have decided to aim for approximation quality, and thus, we chose the former approach, instead of the latter.

3.6 Analysis of Cathode

In this section we provide an analysis on Cathode, regarding its overheads, regarding the orthogonal problems that are part of the replica placement system itself, and regarding the applicability of Cathode to real scenarios. We start with an examination of the overheads in terms of communication, computation, and metadata. Secondly, we provide a brief analysis of some orthogonal problems, and finally, we provide some examples on how Cathode may be applied to various architectures within the category of edge/fog systems.

3.6.1 Communication Overhead

In our analysis regarding the communication overhead of the algorithm, we focus on latency and the steps required to complete a single round of the algorithm, since it is the aspect that primarily affects its performance.

Each iteration of the algorithm requires several communication steps. We define a communication step as a moment in which communication between nodes is needed (for the algorithm to progress), and performed accordingly. A step includes several messages which, in theory, are sent simultaneously. These steps occur between sets of nodes that replicate or consume given item o_i:

- Firstly, *replicas*(o_i) start the optimization round, sending information to *src*(o_i). (1 communication step);
- Shrinking phase ensues, and *src*(o_i) delegates work to *consumers*(o_i). Then, these nodes, after computing the results, send them to *src*(o_i). (2 communication steps);
- Expansion phase ensues, and *src*(o_i) delegates work to *consumers*(o_i). Then, these nodes, after computing the results, send them to *src*(o_i). (2 communication steps);
- The optimized deployment is applied to the system, with *src*(o_i) contacting the *consumers* in which replicas are to be created, and, at the same time, contacting *replicas*(o_i) (1 communication step).

The algorithm is thus composed by 6 communication steps. The latency of each step depends then, of course, on the latency between the nodes in the network, especially between the source node and the sets of consumers and replicas. Of course, since replicas are placed in the nodes of *consumers*(o_i), then we can say that the latency of the steps essentially depends on the latency between the source and the consumers. This latency, in itself, depends on the scenario in which the system is deployed. In Section 3.6.5, we discuss some examples of viable scenarios.

3.6.2 Computation Overhead

When the algorithm is executed, computations are performed in several occasions. The bulk of these computations, however, occur during the shrinking and expansion phases, particularly in the calculation of the partial costs by the consumers. These partial costs, as we have previously detailed, are represented by *oCost*(d_x, n), for a deployment d_x and a node n, in which they are being computed. The *oCost* values must be calculated for each deployment considered in these phases. In the shrinking phase, the number of possible deployments is the following:

$$\textit{nDepls}_{\textit{shr}} = \sum_{k=1}^{k_{\textit{comb}}} \binom{\textit{size}(\textit{replicas}(o_i))}{k} \tag{3.18}$$

In the expansion phase, on the other hand, the number is the following:

$$\textit{nDepls}_{\textit{exp}} = \sum_{k=1}^{k_{\textit{comb}}} \binom{k_{\textit{cand}}}{k} \tag{3.19}$$

The computation of the *oCost* itself then considers all the cost functions of the possible operations. The calculation of these cost functions depends both on the cost function and on the available information for the nodes. For example, consider the *wcost* function defined in Equation 3.3. When *wcost* is calculated in a consumer node n, if the distance from the source to the the replicas is not immediately known by n, then n must consult a local index, or obtain the information remotely. The form by which the information is obtained is out of the scope of this thesis, and so, we abstain from analysing the overhead of computing the cost functions themselves. Hence, in our analysis we merely consider that the calculation of *oCost* function is influenced by the operations' cost functions, and must be calculated, by each participant, $\textit{nDepls}_{\textit{shr}}$ times in the shrinking phase, and $\textit{nDepls}_{\textit{exp}}$ times in the expansion phase. We can thus conclude that the number of calculations in each phase is proportional to its *nDepls* value, and that it is in these values that resides the problematic of the computation overhead.

As it is evident, these numbers depend greatly on the $k_{\textit{comb}}$ and $k_{\textit{cand}}$ parameters, and can raise significantly when these parameters are set to high values, increasing the overhead. The configuration of these parameters, nonetheless, will always depend on the characteristics of the system. It is of the responsibility of the system administrator to set them according to the goals and characteristics of the system.

3.6.3 Metadata Overhead

We separate our analysis of the metadata overhead in *bookkept metadata* and *transmitted metadata*. The former refers to the metadata (relating to the algorithm) that must be stored in each node in order for the algorithm to be executed, while the latter refers to the metadata (relating to the algorithm) that must be transmitted between the nodes. Firstly we analyse the latter, and then the former.

3.6.3.A Transmitted Metadata

Metadata is transmitted at various stages of the algorithm, however, in some cases, the size of this metadata is very small and hardly depends on any factors or parameters of Cathode. In our analysis of transmitted metadata we focus on the most relevant cases, where the size of the metadata is larger and can be influenced by the system state. In some of these cases, it may also be influenced by the decisions made by the system administrator, as we are about to see.

In the information collection stage, prior to candidate selection (Step 1 of Figure 3.2), each replica must send the *candidateSet* and the *summaryTuple* to the source replica. We assume, for simplification, that all identifiers and *weight* values are stored as integers. The space required for these structures is thus defined by the following equations:

$$s_{\mathit{candidateSet}} = k_{\mathit{cand}} \times (2 \times \mathit{size}(\mathit{integer})) \tag{3.20}$$

$$s_{\mathit{summaryTuple}}(o_i, n) = (|\mathit{consumers}(o_i, n)| \times \mathit{size}(\mathit{integer})) + \mathit{size}(\mathit{integer}) \tag{3.21}$$

From these equations we can see that the space required for the *candidateSet* will always depend on the k_{cand} value, which is defined by the system administrator, while the space for the *summaryTuple* solely depends on the amount of consumers that access the replica. As an example, consider that 100 consumers accessed the replica in the last epoch, and $k_{\mathit{cand}} = 20$. Considering the size of an integer as 4 bytes, the size of the *candidateSet* will be merely 160B, while the size of the *summaryTuple* will be 404B.

The other stages in which a considerable amount of metadata is sent between the nodes are during the shrinking and expansion phases, more specifically, when the participants return the results to the source (Steps 3 and 5 of Figure 3.2). In this case, the participants will have to send a set of *oCost* values, whose length is equal to *nDepls*$_{\mathit{shr}}$ in the shrinking phase, and *nDepls*$_{\mathit{exp}}$ in the expansion phase (whose functions are displayed in Section 3.6.2). The space required for transmitting the *oCost* values in the shrinking phase, s_{oCostShr}, and in the expansion phase, s_{oCostExp}, is thus defined by:

$$s_{\mathit{oCostShr}} = \mathit{nDepls}_{\mathit{shr}} \times \mathit{size}(\mathit{integer}) \tag{3.22}$$

$$s_{\mathit{oCostExp}} = \mathit{nDepls}_{\mathit{exp}} \times \mathit{size}(\mathit{integer}) \tag{3.23}$$

Again, in this case, the size of the metadata will highly depend on parameters that shall be tuned by the system administrator. As an example, consider that the number of replicas of a given object is 50. Consider also that $k_{\mathit{comb}} = 2$ and that $k_{\mathit{cand}} = 20$. If an integer's size is 4 bytes, then s_{oCostShr} will be, approximately, 5 Kilobyte (KB), while s_{oCostExp} will be 840 B.

3.6.3.B Bookkept Metadata

The space $s_{\textit{node}}$ required for bookkeeping in a node n is made up of the space required to store information regarding n both as a consumer of data objects ($s_{\textit{consumer}}$), and as a replica node of data objects ($s_{\textit{replica}}$). Just as before, in this analysis we focus on the most relevant aspects that will influence the overhead of metadata, and refrain from details.

As a consumer, n must store the frequencies $f_n(\textit{op}_i^k)$ of each type of operation that it requests to other nodes. This means, that for each type of operation, n stores the identifier (*id*) of that operation and its frequency value. On the other hand, as a replica node, n must store for each object o_i that it replicates, the information used to compute the *candidateSets* and the *summaryTuples*. This is, for each object o_i, and for each consumer m, n must store the the consumer's identifier, the access frequency $a_n(o_i, m)$, and the observed latency $\overline{\delta}(m, n)$.

We assume, for simplification, that all identifiers and frequency values are stored as integers, and that latency values are stored as doubles. Accordingly, the space $s_{\textit{node}}$ required for bookkeeping in a node n, is defined by the following set of equations:

$$s_{\textit{node}}(n) = s_{\textit{consumer}}(n) + s_{\textit{replica}}(n) \tag{3.24}$$

$$s_{\textit{consumer}}(n) = \left(\sum_{\forall_i : n \in \textit{consumers}(o_i)} \textit{reqOpTypes}(n, i) \right) \times (2 \times \textit{size}(\textit{integer})) \tag{3.25}$$

$$s_{\textit{replica}}(n) = \sum_{\forall_i : n \in \textit{replicas}(o_i)} \left(\sum_{m \in \textit{consumers}(o_i, n)} (2 \times \textit{size}(\textit{integer}) + \textit{size}(\textit{double})) \right) \tag{3.26}$$

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 the size of a double as 8 bytes, the space necessary for bookkeeping consumer metadata ($s_{\textit{consumer}}(n)$) in this node would be of 16 KB, while the space necessary for bookkeeping replica metadata ($s_{\textit{replica}}(n)$) would be of, approximately, 1.6 Megabyte (MB).

3.6.4 Orthogonal Problems

In this section we briefly discuss some of the problems that are part of the replica placement system, but which are not the focus of our work, and therefore we consider orthogonal to the contributions of this thesis. In these paragraphs we aim at giving insight into these problems and how they can be tackled.

3.6.4.A Replica Discovery

In the design of Cathode we assume that there is an underlying replica discovery system that allows consumers to have knowledge of where the replicas are located. These systems can be implemented in many different ways. In the example of D-Rep [4], the authors propose a very simple and lightweight solution, in which, when a replica is created in a node n_2, the closest replica node to n_2 (which we name n_1), sends a notification to the consumers that have requested the replica from n_1, in the last epoch. Furthermore, from the example of Autoplacer [32], we know that there are ways to efficiently store the information necessary for replica-aware routing. In that particular data placement system, the authors achieve this by creating a probabilistic data structure that allows for efficient look-ups and storage of replica-related information.

3.6.4.B Knowledge of Latency Values

Cathode assumes that each consumer has all the necessary information regarding latency values to perform the cost calculations in each round. In weak operations, like READCLOSEST, in which only the closest replica is contacted by the consumer, this information is easily collected by having the consumer measuring the RTT of the request. However, in operations like WRITESOURCE (whose cost function is detailed in section 3.3.1), collecting the necessary information is not so obvious. Such can be achieved in a number of different ways, e.g., by having the source/replicas propagate the experienced latency values in the responses to the consumers.

In this thesis, we will not delve into how this information can be stored and collected, since it would mean to develop a whole underlying system, which should be orthogonal to Cathode. However, since such a system can further contribute to the validation of Cathode, these problems should be tackled in future work.

3.6.4.C Data Object Granularity

Currently, 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 each treated as a single object by the algorithm. This way, Cathode would collect statistics for each of these partitions, instead of keeping track of the individual objects in each partition.

Furthermore, in the expansion phase, when a new replica is created, all the objects of the partition would be replicated on the new replica node; while in the shrinking phase, when a replica is removed, all the objects of the partition would be removed from the node. This allows to keep the bookkeeping costs of Cathode within some target limits defined by the system administrator.

3.6.4.D Fault Tolerance and Partitioning

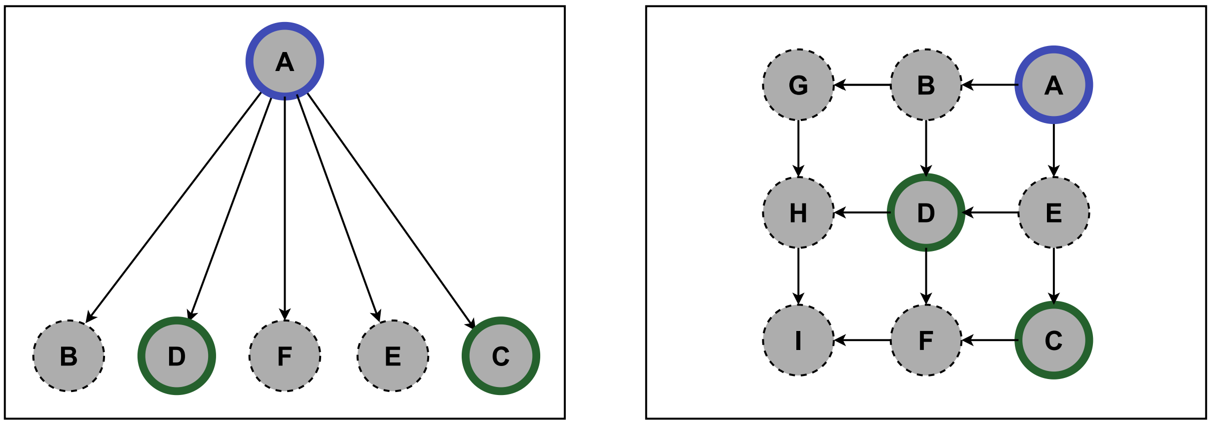
Fault tolerance is also another problem that is orthogonal to our contributions. As we have seen, Cathode has one of the replicas (the source replica) playing a special role in the algorithm. If this replica fails, a leader election algorithm should elect another replica as the new source. In face of a network partition, the algorithm would elect a source in each partition, so that placement decisions can be executed in both partitions, in parallel, without blocking the placement algorithm. When the network partition heals, the leader election mechanism would ensure that only one source remains active. This can also be implemented in different ways, which we do not develop in this thesis.

Note that some data consistency models, namely strong consistency models, may block operations in the presence of network partitions. For instance, some operations may require a majority of replicas, and thus, may block when operating in partitions where there is only a minority of replicas. This is unavoidable, and has been captured by the now well known CAP theorem [37]. However, in this case scenario, Cathode will still continue to adjust placement for all objects that use weak consistency and that remain live during the partition.

3.6.5 Applicability of Cathode to Edge Architectures

Cathode is flexible enough to be applied to a variety of systems, however, it was specifically designed to perform well in edge/fog scenarios. The architecture defines a *source* node for each object, which acts as the coordinator for the decisions regarding the placement of its replicas. This type of architecture is not far off from the architectures of other edge systems, such as Gesto [38], a data storage system for the edge, that enables causal consistency in a scalable manner. Gesto's architecture is composed of three tiers. The highermost tier is composed of the datacenters in the cloud, which store all application data. The lowermost tier is composed of the partial replicas of the application state, located in edge cloudlets within regions. Then, in each region, a *broker* unit composes an intermediate layer, being responsible for the metadata management in that region, and the communication between the cloud and the cloudlets. It is then intuitive to see the parallel between the role of *broker units* in Gesto, which manage the metadata of data objects, and the *source nodes* in Cathode, which manage the replicas of data objects.

We have already talked about the Cloudpath platform [9] in Section 2.3. This is another multi-tier architecture for data storage, establishing a node hierarchy from cloud to edge. In this platform, stronger



(a) Multi-tier edge network. **(b)** Lower-tier grid edge network.

Figure 3.4: Examples of system architectures and how Cathode may be deployed.

consistency operations may be achieved by forwarding requests to higher tier nodes, which maintain a higher level of consistency. For example, an operation that is forwarded to a city-level node, provides a consistent view for all the consumers (lower-tier edge nodes) in the city. Cathode can be deployed in such an architecture, by placing the source of the data in the city-level node (Figure 3.4a), which then manages the placement of data replicas within that city, distributing computations among the lower-tier nodes, and aggregating their results. In these systems, communication between lower-tier nodes and their higher-tier master node is frequent and necessary for synchronization (in the example of Cloudpath, updates are periodically propagated to higher levels of the hierarchy); Thus, for such a system to function well, it is expected that the latency between, e.g., the lower-tier edge nodes and the city-level node, must be relatively low. Therefore, when applying Cathode to those systems, the latency between source (city-level) and the consumers (lower-tier) is expected to be low, and, consequently, the communication overhead should be tolerable.

Another viable manner in which Cathode may be deployed is within the lower-tier layer itself. In a smart city scenario, edge nodes may be installed at various spots in the city, each managing an area, while generating, collecting, and storing data. The sources of data objects in this case can be any node within the city (Figure 3.4b). These edge nodes might reduce latency by communicating between each other in a grid-like network, instead of having to establish connections through higher tier nodes. However, in such networks, the latency between source and consumers may vary greatly. In scenarios where a high number of consumers are distant to the source, the parameter T_{part} can be set to a lower value, reducing the number of consumers to be contacted. Note that this does not mean that the farthest consumers will not be contacted; but that the only nodes to be contacted will be only the most important ones (as it was explained in Section 3.5.7).

In some smart city applications, each node may be the source of data that specifically regards the area which is delegated to that node. For example, in [36], the authors propose an edge data collection system for vehicular networks, in which road-side units store the information regarding road hazards in an area. This concept can be applied to smart city scenarios, where the road-side units take the form of edge nodes distributed across a city, each being the *source* of the road hazard data of the corresponding area. In these types of applications it is intuitive that a great part of the consumers of the data generated in an area, will be consumers located in neighbouring areas. We call this a *locality-driven* workload. In the example of the road hazard application, it makes sense that the clients, which should take the form of smart vehicles, will want to access road hazard information from neighbouring areas, and the same would apply to any other data regarding traffic conditions. Applications such as Tripadvisor, where users search for nearby interest locations also display such a workload. Assuming the edge nodes in a city communicate between each other through their neighbours, in a grid-like network, it is expected that Cathode will perform placement decisions with especial communication efficiency in these locality-driven workloads, since the consumers are very near from the sources.

3.7 Implementation

In this section we describe the implementation of Cathode that was used in our evaluation process. To perform this evaluation we also implemented two other state of the art replica placement algorithms, namely D-Rep [4] and Holistic-All [12], whose implementation we shall also describe here. D-Rep was chosen due to the fact that it is, in our view, the most relevant generic replica placement algorithm made specifically for the edge. Holistic-All is another decentralized replica placement algorithm, but represents a polar opposite of D-Rep in terms of what criteria it prioritizes.

While Holistic-All aims at achieving high approximation quality, at the cost of requiring a slower, more coordinated operation, D-Rep aims at reducing the amount of coordination required to optimize placement, sacrificing approximation quality in the process. Cathode, however, strikes a balance between the two approaches, all the while being able to account for the different operations and consistency models used in the system, in order to provide higher quality solutions.

We start this section by providing a brief description of the development environment, then describing the various consistency models that were implemented, along with their corresponding operation types and cost functions. We finish the section by providing an overview on how each algorithm was implemented, focusing on the differences between our implementation and their original description.

3.7.1 Development Environment

The prototypes of the algorithms have been implemented in Java (OpenJDK 8). As a base for the implementation, the PureEdgeSim framework [13] was used. This is a simulation framework based on

CloudSim Plus, designed to simulate cloud and edge computing environments. This framework was extended with several general classes that are then used by all the algorithms. Each algorithm then has its own implementation under the general framework, extending it with its own protocol. It is also in the general framework, that the various operations (implementing several consistency models) can be found. We delve into these operations in the following section.

3.7.2 Operations and Consistency Models

In our framework we include just two major classes of operations: READ operations, that do not change the state of the object, and WRITE operations, that do change the state of the object. For each of these operation classes we then implemented multiple operation types, according to various consistency criteria. We have included four different consistency algorithms in our framework:

- *linerizability:* We consider the non-blocking implementation of atomic registers proposed in [17]. 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; these operations have been described in detail in Section 3.3.1.

- *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 3.1 (for a rationale of these cost function, see Section 3.3.1). Note that all of these cost functions only account for the experienced latency of the synchronous operations (in which a node has to await for responses, in order to proceed). As it is obvious, cost functions can also account for factors such as the network congestion caused

Operation	**Cost Function**
WriteSource	$2[\delta(n, \textsf{\textit{src}}(o)) + \max_{\forall j \in \textit{replicas}(o)} \delta(\textsf{\textit{src}}(o), j)]$
ReadClosest	$2 \min_{\forall j \in \textit{replicas}(o)} \delta((n, j)$
WriteClosest	$2 \min_{\forall j \in \textit{replicas}(o)} \delta((n, j)$
ReadMajority	$2 \max_{\forall j \in \textit{closestQuorum}(n,o)} \delta((n, j)$
WriteMajority	$2 \max_{\forall j \in \textit{closestQuorum}(n,o)} \delta((n, j)$
ReadWriteMajority	$4 \max_{\forall j \in \textit{closestQuorum}(n,o)} \delta((n, j)$

Table 3.1: Cost functions of the implemented operations

by asynchronous updates. This could be implemented by simply extending the cost functions we have defined, however, to simplify the evaluation process, we solely considered the factor of latency.

3.7.3 Algorithms

In general, the implementation of all the algorithms follows their description, however, in order to perform a fair comparison, some of their features were tweaked to fit into the general framework we have developed. In this section we analyse the main differences between the description and implementation of the algorithms.

3.7.3.A Creations, Removals, Replacements

In our implementation, we consider three types of decisions regarding replica placement:

- *Creations*, in which a replica is created in a node;
- *Removals*, in which a replica is removed from a node;
- *Replacements*, in which a replica is removed from a node, and replaced by a new replica of another object.

In the case of Cathode, only creations and removals are considered, with the implementation being similar to its description. In the case of D-Rep and Holistic-All, however, there are a few differences.

D-Rep, according to its description, considers creations, removals and replacements, and as so, we implemented all three of them. However, in the description of D-Rep, the authors note that, for the removal of a replica, D-Rep estimates a factor that determines the expected replica utilization. This factor is then compared with its current observed utilization, and the replica is removed if a lower bound is reached. Since the authors do not specify how this is implemented, we opted to implement a precise estimation process, which accounts for information about the system in its totality. This way, we consider that the algorithm operates in a best case scenario, so that in our evaluation, its performance is as good as possible. It is our aim to prove that, even when running this perfect estimation on removals, the performance of D-Rep is still excelled by Cathode.

Holistic-All, on the other hand, also considers creations, removals and replacements. This deviates from its description, in which it only considers the replacement of replicas in a full cache. In our implementation, along with replacements, Holistic-All considers the benefits of simple creations and removals, which are calculated in an analogous way, putting the algorithm on a fair ground with Cathode and D-Rep, while hardly adding any complexity. Just like in D-Rep and Cathode, we have extended Holistic-All to consider storage costs when calculating the gains of each possible deployment. This also does not add any overhead to the performance of the algorithm.

3.7.3.B Replica Discovery

Since neither Cathode nor Holistic-All define an algorithm for replica discovery discovery, we have not implemented such a feature in any of the algorithms. We thus assume that, in all algorithms, the consumers of an object know, a priori, where all the replicas are located, and, therefore, requests are always sent to the nearest replica. This way, for the same deployment, the form by which requests are propagated is the same for each algorithm.

Summary

This chapter addressed Cathode, a decentralized consistency-aware replica placement algorithm for the edge. By collecting information from various nodes, this algorithm is able to have an ample view of access patterns, which it uses to calculate deployments that present a high benefit for the global system. Alongside, its item-wise decentralized structure also allows it to account for the costs of multiple different consistency protocols. Unlike other algorithms that sacrifice efficiency for quality, Cathode tries to maintain efficiency, by distributing computation intensive tasks among multiple nodes, all the while providing multiple customization parameters to the system administrator, which may be used to adapt the operation of the algorithm to the goals of the system. In the sections above, we have provided a detailed description of the algorithm, an analysis of its overheads, problems, and applicability; and finally, we went over some of the details of its implementation along with the implementations of two other algorithms that were used in our evaluation process.

4 Evaluation

Contents

This chapter presents the experimental evaluation of Cathode's performance and compares it with the performance of Holistic-All [12] and of D-Rep [4]. We aim to show that Cathode manages to strike a balance between the advantages of these two algorithms, and surpass their performance in certain edge scenarios. Additionally, we assess the gains that can be achieved by Cathode, derived from the fact that it considers the different operations and consistency models used by the underlying system.

In Section 4.1 we present the main goals of the evaluation, and describe the experimental setup in Section 4.2. Section 4.3 then presents a short analysis and selection of the system parameters that were used in the evaluation. In Section 4.4 we present and discuss the results of the experiments regarding convergence speed; in Section 4.5 we analyse how the algorithms perform in terms of approximation quality; in Section 4.6 we delve into the results regarding the capacity of the algorithms to adapt to dynamic workloads; and in Section 4.7 we analyse the performance of the different algorithms with multiple consistency protocols. Finally, we discuss and relate the results of each experiment in Section 4.8.

4.1 Goals

The evaluation process of Cathode mainly aims at answering the following questions:

- What is the quality of the deployments provided by Cathode and how fast is it at converging to a stable solution, when compared to other placement algorithms?
- How well does Cathode adapt to dynamic workloads, when compared to other placement algorithms?
- Do the semantic-aware mechanisms of Cathode bring advantages over simpler oblivious approaches?

4.2 Experimental Setup

Our evaluation is performed by simulating the execution of the algorithms, using the PureEdgeSim framework [13], as we mentioned in the prior chapter. In this section we delve into some of the details of the experimental setup, that are indispensable to comprehend the results we present in the rest of the chapter.

4.2.1 Time Measurement

The evaluation we have performed is purely a comparison of Cathode to the other replica placement algorithms. Therefore, to represent time, we use an abstract unit named *epoch*, whose value is not absolute, but relative. As we have stated prior, the algorithms we evaluate are “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, computing a new deployment.

In our evaluation, we use these epochs as a measure of time. However, the (real) time interval that corresponds to an epoch would depend on factors such as the request rate of the underlying system. In our evaluation, which focuses on evaluating the quality of deployments, the request rate is not a factor that exerts 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, in our experiments, 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.

Note that, this way, the epoch is merely an abstract time window with the same duration for each algorithm, independent of its operation and speed. Note also that, in the default case, at the end of each epoch, an optimization round starts; however, if the current optimization round isn’t finished by the time the new epoch ends, then no new optimization round is started; what happens, instead, is that the current optimization round continues running until it is finished, and then, only at the end of the next epoch, a new optimization round is started. As we are about to see, this case is common in Holistic-All, which often takes multiple epochs to finish an optimization round.

In the evaluation of D-Rep, the authors have studied how the duration of epochs affected the effectiveness of the placement algorithm. This type of evaluation is not relevant our goals, thus we have not performed it, however, as a reference, the evaluation of D-Rep has shown that epochs of 4 to 7 minutes provide satisfactory results [4].

4.2.2 Object Size and Storage Constraints

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.

4.2.3 Network Topology

In the evaluation process we consider two network topologies, which allows us to analyse how Cathode performs in different architectures:

- *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. These networks are characterized by having a few highly connected hubs which reduce the average distance between nodes. Such a model was used, for example, in the original evaluation of D-Rep [4].
- *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 low-tier edge nodes may be placed along the roads, or areas, of a city. These networks are characterized by their large diameter, with each node being connected to very few nodes, in a uniform way. Network hubs are absent in this model.

4.2.4 Workloads

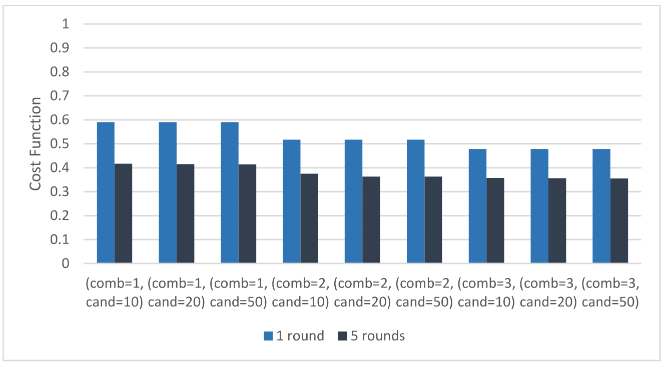
In the evaluation process we consider two different types of workloads, which allows us to analyse how Cathode performs in different scenarios:

- *Client-driven workload:* In this workload, clients access objects according to their own interests, regardless of their position within the network. This workload captures applications such as social networks, news websites, etc., and it has been observed that these workloads are highly skewed [39, 40]. 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 α ranging between 0.7 and 0.8, an interval that can represent several types of environments as shown in [39].

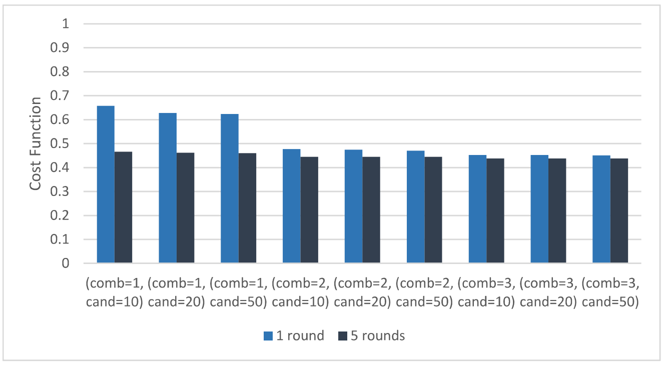
  We divide this workload type in two subtypes, *homogeneous* and *heterogeneous*. In the *homogeneous client-driven workload*, the global distribution of requests also follows a Zipf law. In the *heterogeneous client-driven workload*, clients have independent distributions from one another, making the global distribution 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. In this workload, every time a consumer 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 consumer and the object (consumers access nearby objects more frequently than far away objects).

Despite Cathode featuring full support for multiple consistency models, D-Rep and Holistic-All will only be prepared to work within a *weak consistency* model, as we defined it in Section 3.7.2. We show the advantages of Cathode in this regard in section Section 4.7. However, in order to showcase how Cathode still manages to surpass the performance of the other algorithms in the weak consistency model, we have performed all the other experiments (Sections 4.2, 4.4, 4.5 and 4.6) solely allowing weak consistency operations.



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



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

Figure 4.1: Cost function after 1 and 5 epochs of runtime, for several parameter configurations of (k_{comb}, k_{cand}) (static workloads).

4.3 Configuring the Parameters of Cathode

Before performing our comparative evaluation of Cathode against the other algorithms, we must first set its configuration parameters to the values that will bring the most benefit in the workloads/topology we are operating with. As we have seen, the operation of Cathode can be tuned by four parameters:

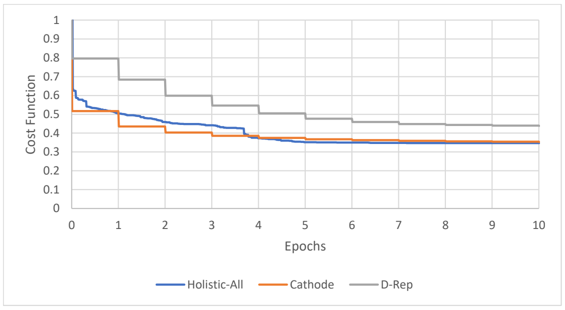
- T_{gain}, the minimum gain (cost reduction) that justifies adding or deleting a replica;
- k_{cand}, that controls how many candidates are considered in the optimization process;
- k_{comb} that controls how many combinations (of creations/removals) are considered in the optimization process;
- T_{part},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

The value of T_{gain} is application dependent, as it defines the minimum gain that can bring business benefits. This value can be used to refrain from performing new deployments whose resulting gain is barely positive. Because in the evaluation we wanted to observe the full optimization potential of the different algorithms, we let all algorithms perform new deployments as long as the gain is positive, so we set $T_{gain} = 0$.

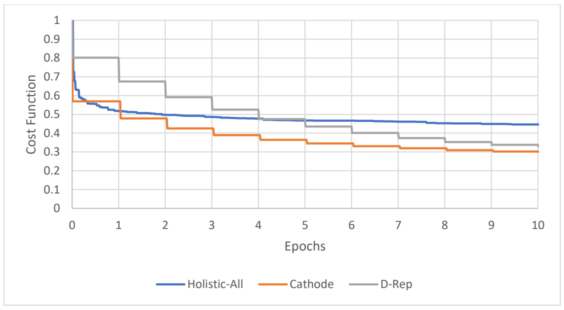
The other threshold, T_{part}, affects the approximation quality, with lower values leading to less participants in the optimization process, which may reduce the overheads of communication, but may also decrease quality. Essentially, this parameter constitutes a simple way to reduce network congestion and the latency (experienced in the shrinking/expansion phases), in specific scenarios where these turn out to be a problem. Since we aim to prove that Cathode is able to surpass the convergence speed of other algorithms, even when all consumers participate in replica placement, we have set this parameter as its maximum value, $T_{part} = 1$.

At last, k_{cand} and k_{comb} affect the convergence speed, with higher values leading the algorithm to consider more deployments, and thus, more information, in each epoch. However, it also increases the amount of computational resources consumed by the algorithms, as we have seen in Section 3.6.2. Thus, 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. To determine a good configuration of these values, instead of evaluating every possible combination, we merely picked several plausible configurations and measured the cost function of the deployment computed by Cathode after 1 round of the algorithm, and after 5 rounds of the algorithm, for a static workload. The results are depicted in Figure 4.1. In general, the parameter that appears to exert more influence is k_{comb}. As can be observed, in both networks, the the most noticeable performance improvement, is when k_{comb} is switched from the value of 1 to 2. Despite this, the result after 5 epochs is fairly similar for all configurations, hinting at the fast convergence of Cathode, which we will study further on in this document. It is possible to see that increasing k_{comb}

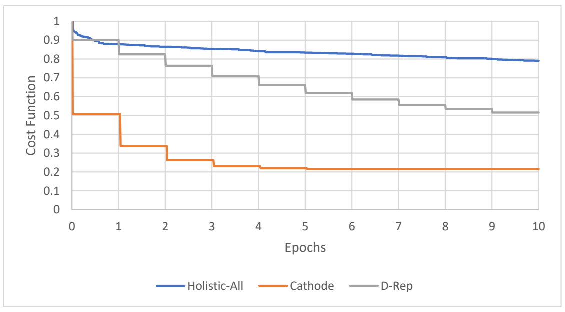
above 2 or increasing k_{cand} above 10 does not bring significant improvements. Thus, in our experiments we have used $(k_{comb} = 2, k_{cand} = 10)$.



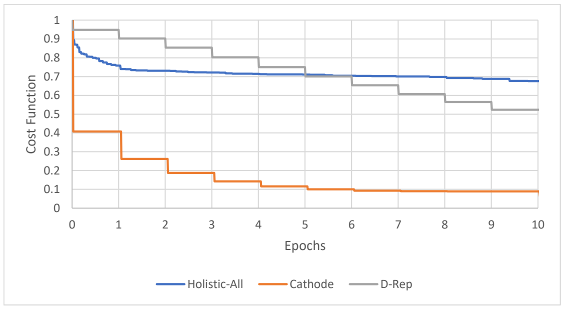
(a) Scale-free network, homogeneous client-driven workload, N=100



(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

Figure 4.2: Evolution of the cost function during 10 consecutive epochs (static workloads).

4.4 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 will then access objects using one of the workloads described in Section 4.2.4, while the system runs for several epochs. We then plot the evolution of the cost function as the experiment progresses. Note that, in this experiment, the workload is static, so the access patterns stay constant along the epochs.

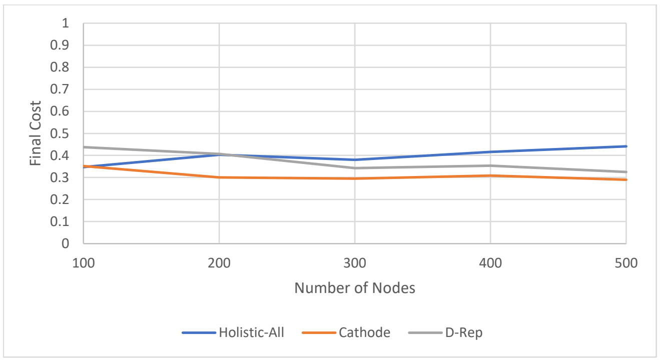
Figure 4.2 depicts the evolution of the cost function in time, for a varying number of nodes, using different combinations of network topologies and workloads. One of the first observations that can be made is the fact that Holistic-All presents a more gradual progress than D-Rep and Cathode. This is because, in the latter two, the optimization process is fairly quick and finishes in the same epoch it started. However, after applying a deployment, D-Rep and Cathode are designed to wait a full epoch to observe the requests patterns, in order to run the optimization process again. In contrast, Holistic-All deploys replicas gradually, and a single optimization process can span several epochs, as it tries to find the best possible deployment, with a global view of the problem.

As shown in Figures 4.2a and 4.2b, Holistic-All and D-Rep work best in scale-free networks. This happens because these networks have a low diameter, which translates to less hops between the nodes, 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, resulting in lesser quality deployments. Then, these deployments are often temporary, since each node, in an epoch, can only propagate a replica to its neighbours. Thus, for a replica to reach its optimal location, it may have to hop through multiple nodes, taking multiple epochs to reach its destination.

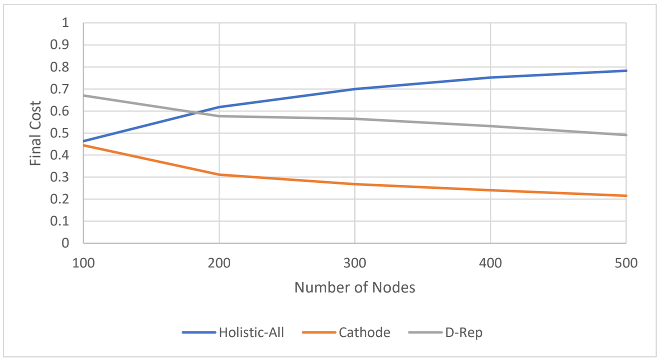
Figures 4.2c and 4.2d show that Holistic-All also has a much slower convergence on grid networks. Figure 4.2a, on the other hand, shows that Holistic-All performs better in networks with a small number of nodes. This can be explained by the fact that Holistic-All requires nodes to wait, for several serial rounds, for the placement decisions performed in other nodes. This process turns out to be quite inefficient in systems with a large diameter and a large number of nodes. As we have seen, Cathode is able to outperform both Holistic-All and D-Rep on large diameter networks, in both the locality-driven workloads and the heterogeneous client-driven workloads. This happens because, on one hand, the information and communication needed to run the algorithm is less than in Holistic-All, with calculations being executed in parallel in the consumer nodes. On the other hand, placement is performed using a data horizon that allows the algorithm to place replicas exactly where they are needed, instead of having to propagate these replicas from neighbour to neighbour, as it is done in D-Rep. This way, Cathode is able to place the most ”important” replicas, in their optimal locations, in a single epoch. Another aspect to note is the fact that Cathode is able to reduce cost further in grids than in scale-free networks. This is observed due to the fact that, in a grid network, because of its large diameter, initial costs are larger than in a scale-free network, since consumers are farther from replicas. Thus, there is more potential for cost reduction in grid networks.

4.5 Approximation Quality and Scalability

Approximation quality has been defined as the degree to which a placement algorithm can approximate to the optimal solution of the total cost function. We assess this quality in Figure 4.3, which shows the total cost values of the final deployment, for several system sizes. These final costs were measured in relation to the initial cost, after 10 epochs of runtime with static workloads. It can be observed that, on scale-free networks, in all of the algorithms, the quality of the solution is less affected by the system



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

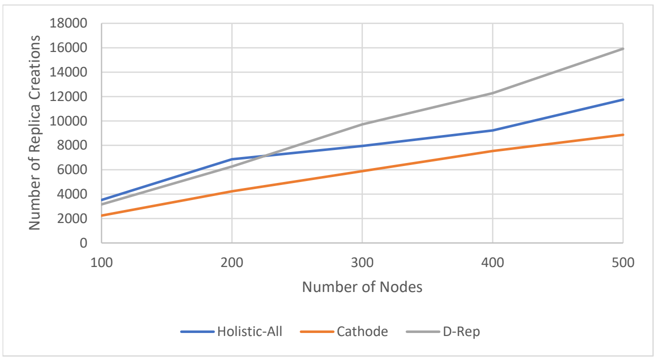


(b) Grid network, locality-driven workload

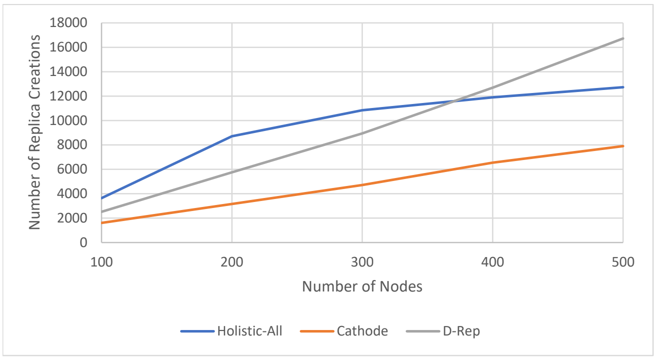
Figure 4.3: Final cost after 10 epochs of runtime, as a function of system size (static workloads).

size. This is because the quality of the solution mainly depends on the placement of replicas on the few network hubs, whose number grows logarithmicaly. Despite this, as it can be seen, even in scale-free networks, the performance of Holistic-All is affected by the number of nodes, with Cathode achieving a cost 10% lower than Holistic-All when $N = 500$. On grid networks, on the other hand, a larger system size directly translates to a larger diameter, and thus, a higher costs. As we have seen, convergence is slower in large grid networks both for Holistic-All and D-Rep, therefore, even after 10 epochs, the quality of the deployments provided by these algorithms is still considerably worse than that of Cathode, which, by epoch 10, has already converged to a somewhat stable solution.

In Figure 4.4 we have the number of replica creations performed in the same network/workload pairs of Figure 4.3. As we can see, Cathode is the most conservative algorithm, creating the least number of replicas in each case. This is explained by the fact that Cathode attempts at calculating the deployment that appears to be the most optimal in the present workload. On the other hand, 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. However, out of the three algorithms, D-Rep is the one that creates the most replicas in larger systems. Again, this is because replicas need to hop through multiple nodes, in order to reach their locations. In Holistic-All, we can also see that the evolution of the number of creations



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



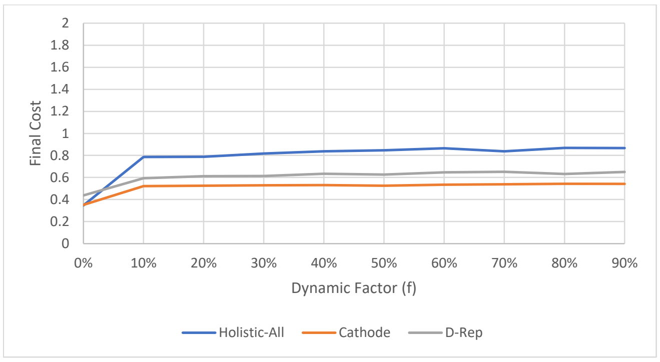
(b) Grid network, locality-driven workload

Figure 4.4: Number of replica creations in 10 epochs of runtime, as a function of system size (static workloads).

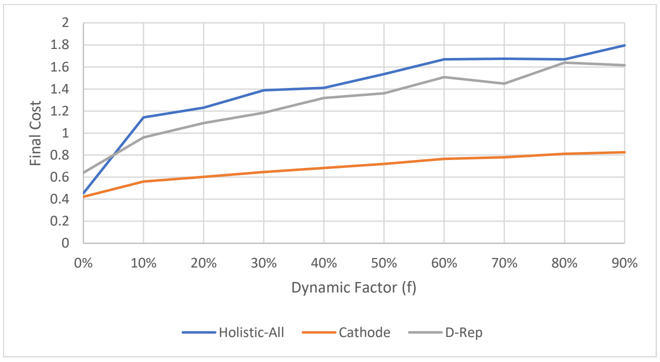
starts to flatten for higher numbers of nodes, which happens due to the fact that Holistic-All's operation is slower in large systems. There are several disadvantages for an algorithm that creates a large amount for replicas. Firstly, when replicas are created, they must be transfered between nodes, which incurs an overhead. Secondly, replicas will take up the storage capacity in a node, which can be a problem in real systems. Both of these disadvantages depend on the size of the objects, which in itself, is related to their granularity, and the application that is running on the system. Another problem of creating a larger number of replicas arises when dealing with dynamic workloads. In the next section we will explore how this problem is relevant and especially affects the approximation quality of D-Rep.

4.6 Adaptability to Dynamic Workloads

We will now analyse the performance of the different algorithms when dealing with dynamic workloads. For these experiments we let each algorithm run for 10 epochs. However, in each epoch, the set of objects accessed by some clients is changed. The way we change the access pattern is as follows. Let $\mathit{Interests}_c$ be the set of objects that is accessed by a consumer c during a given epoch. At the end of each epoch, we change a fraction f of these sets, in order that some of the objects to be accessed are



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

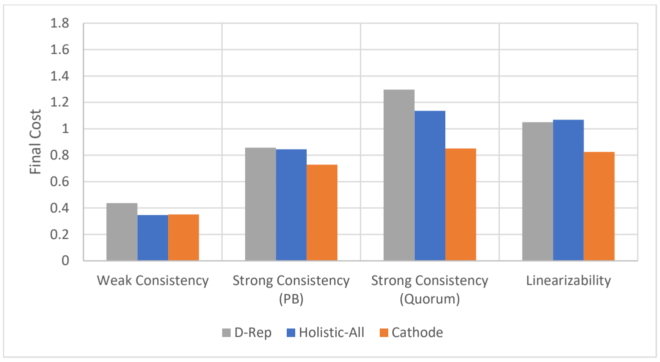


(b) Grid network, locality-driven workload, N=100

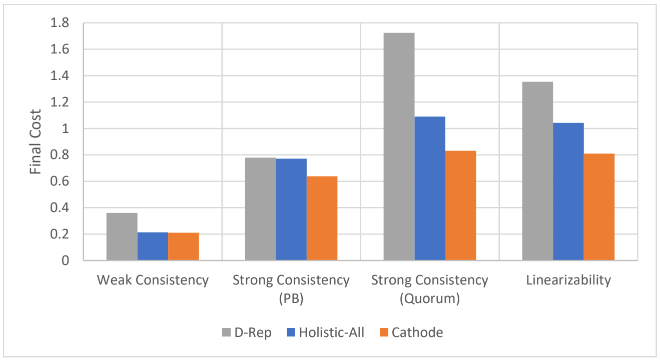
Figure 4.5: Final cost after 10 epochs, as a function of the dynamic factor (dynamic workloads).

different. The new (changed) sets in each epoch are generated from a Zipf distribution correspondent to that epoch. In a real system these may represent sudden new interests, for data, that arise for a limited time and quickly fade away. In Figure 4.5, the resulting final cost for a system of 100 nodes, is depicted as a function of the fraction of change f. Note that if $f = 0$, the workload is static, and if $f = 1$ the workload changes drastically during the experiment.

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. It is worth noting that, despite this, Cathode still manages to achieve a better performance than the others, maintaining virtually the same final cost from $f = 10\%$ to $f = 90\%$. In grid networks, however, the differences are more noticeable. As f rises, even in a system of only 100 nodes, D-Rep and Holistic-All seem to not be able to adjust fast enough to dynamic workloads, and the cost of the deployment increases. In sharp contrast, Cathode still offers considerable benefits even in highly dynamic workloads, reducing cost in almost 20%, after 10 epochs, when $f = 90\%$. Again, the convergence speed plays a significant role in how the algorithms are able to adapt to the changes in these dynamic workloads. If an algorithm takes many epochs to achieve high quality deployments, then it can never keep up with the changing interests of the consumers. In the case of Holistic-All, a single optimization round can endure



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



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

Figure 4.6: Final cost after 10 epochs with different consistency protocols (static workloads).

multiple epochs, in which Holistic-All is operating on the information available at the time it started the optimization process. This means that, in dynamic workloads, Holistic-All will calculate deployments with outdated information, that may at certain points be barely representative of the consumer interests. D-Rep, is also affected by the convergence speed, for the same reasons we stated in the prior sections, but aside from that, it has another problem. D-Rep always creates a higher number of replicas than Cathode, as we have seen in Figure 4.4. In scale-free networks, most of the replicas will tend to be created in the network hubs, which are optimal locations, thus making it easier to adapt to dynamic workloads, since network hubs are easy to reach and very useful. In grids, however, when the workload changes, many replicas will be made useless. Therefore, they can come to represent high storage costs (*sCost*), with no reduction of operation costs (*oCost*). Contrasting these two algorithms, as we have explained, Cathode tries to place the most important replicas in their optimal positions in a single epoch, hence its positive results.

4.7 Consistency-Awareness

We now explore the advantages that can be obtained by explicitly considering multiple consistency protocols of the system, in the operation of the placement algorithm. Figure 4.6 shows the cost of the final deployment of each algorithm, for the 4 different replica consistency protocols described in Section 3.7.2, in a system of 100 nodes, with a static workload (75% reads and 25% writes). In both networks, a homogeneous client-driven workload was used, which better showcases the performance of the algorithms when using different consistency models.

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. In this type of model, creating a new replica always reduces the latency experienced in the system. However, this is not the case when a quorum based replication scheme is used. Contrasting D-Rep and Holistic-All, Cathode is able to decrease costs by 17% on average, when quorum strong consistency, or linearizability are used, due to the fact that it is aware of the semantics of the operations, taking into account the cost function and frequency of each of them.

4.8 Discussion

What motivated the development of Cathode was the absence of a consistency-aware replica placement algorithm for the edge, that operated in a scalable way that would allow for high convergence speed. In the evaluation process, we expected Cathode to outperform previous systems in terms of convergence speed, and we expected that this factor would be crucial to the overall approximation quality of these algorithms, especially when operating in large diameter networks, and when dealing with dynamic workloads. Also, we expected Cathode to be effective at reducing costs with different consistency protocols. From the experimental results, we have shown that our expectations meet with reality.

Despite Cathode having a fast convergence in all scenarios, it is in large grid networks that it manages to excel the other algorithms. This property of Cathode is due to the fact that it attempts to place the most important replicas in their optimal locations, in a single epoch. Such a strategy is useful especially in grids, where there are no network hubs, and nodes are far from each other. This same property (convergence speed) was shown to highly affect approximation quality, especially when the workload is dynamic, in which Cathode presents improvements from other algorithms in all types of networks and workloads. We have also seen how Cathode is more conservative, creating much less replicas (in some cases, half of the replicas created by other algorithms), despite achieving better quality deployments.

Finally, Cathode is able to reduce costs in all consistency protocols, contrary to previous algorithms which are consistency-blind. 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.

Summary

This chapter presented the experimental evaluation of Cathode. With the use of simulation software we tested Cathode against two previous algorithms, using various combinations of workloads and network topology. We evaluated several factors, namely, convergence speed, approximation quality, adaptability to dynamic workloads, and the performance with different consistency protocols. The results showcase that Cathode is able to achieve higher quality deployments in a shorter amount of time than the other algorithms in grid networks, while showcasing approximately the same performance as these algorithms in scale-free networks. When dealing with dynamic workloads, Cathode achieves better quality in every scenario, always being able to reduce costs, while the other algorithms increase costs in some cases. The same is true when using stronger consistency protocols, like linearizability and quorum-consistency, in which Cathode is able to reduce costs in an average of 17%, while the other algorithms increase costs.

5 Conclusion

Contents

5.1 Conclusions

In this dissertation, we have addressed the problem of replica placement in the edge computing paradigm. Most of the previous work either didn't consider the possibility of optimizing placement for multiple consistency protocols, or didn't provide a distributed and scalable way to do it. Furthermore, replica placement systems either focused on the quality of their deployments, or on efficiency. In this report, we have described the design, implementation and evaluation of Cathode, a replica placement algorithm that provides a scalable operational structure to achieve high quality solutions, and support multiple consistency protocols. In an experimental evaluation, Cathode was compared with two state of the art solutions. The results show that these solutions are outperformed by Cathode in most of the explored scenarios. When they do outperform Cathode, the quality of the latter is always within, approximately, 5% of the quality achieved by the other algorithms. In some environments, such as large grid networks, Cathode is able to greatly surpass the quality of the other solutions in real time, because of its fast convergence, occasionally improving the utility by a factor of $1.7\times$. Furthermore, while the other solutions fail at reducing costs in certain consistency models, Cathode succeeds.

5.2 Future Work

In our analysis of Cathode we delved into some orthogonal problems, which are worth exploring in the future. Currently, the system assumes that there is an underlying replica discovery system where consumers know where the closest replicas are located, which allows them to perform the cost calculations of alternative deployments. The system also assumes that the nodes are informed regarding the latency from the source to some of the replicas. It is of interest to explore how this information can be collected and stored by the nodes in an efficient way. Alongside, since network partitions are commonplace in the edge, and can disrupt the functioning of Cathode, it would be relevant to develop a partition tolerance system that mitigated these disruptions.

Due to time constraints, the evaluation of Cathode was entirely based on simulations and synthetic workloads. One of the plans for future would be to perform tests with real workloads and develop a real deployment of Cathode. Furthermore, in our evaluation, we explored 4 different consistency protocols, namely weak consistency, primary-backup based strong consistency, quorum-based strong consistency, and atomic registers. As future work it would be interesting to explore other weak consistency models, such as causal consistency and session guarantees.

Bibliography

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

[3] A. Kasprzok, B. Ayalew, and C. Lau, "Decentralized traffic rerouting using minimalist communications," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, 2017, pp. 1–7.

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

[5] M. Breitbach, D. Schäfer, J. Edinger, and C. Becker, "Context-aware data and task placement in edge computing environments," in *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom). IEEE*, 2019.

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

[7] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, "iFogStor: an IoT data placement strategy for fog infrastructure," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, 2017, pp. 97–104.

[8] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, "Fogstore: toward a distributed data store for fog computing," in *2017 IEEE Fog World Congress (FWC)*, pp. 1–6.

[9] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[10] S. Mortazavi, B. Balasubramanian, E. de Lara, and S. Narayanan, “Toward session consistency for the edge,” in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[11] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, “Dual-quorum: a highly available and consistent replication system for edge services,” *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 2, pp. 159–174, 2008.

[12] V. Sourlas, L. Gkatzikis, P. Flegkas, and L. Tassiulas, “Distributed cache management in information-centric networks,” *IEEE Transactions on Network and Service Management*, vol. 10, no. 3, pp. 286–299, 2013.

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

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

[15] 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.

[16] M. Herlihy and J. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[17] H. Attiya, A. Bar-Noy, and D. Dolev, “Sharing memory robustly in message-passing systems,” *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 124–142, 1995.

[18] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, “Spanner: Google’s globally distributed database,” *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[19] C. Papadimitriou, “The serializability of concurrent database updates,” *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[20] N. Afonso, “Mechanisms for providing causal consistency on edge computing,” Master’s thesis, Instituto Superior Tecnico, Universidade de Lisboa, Nov. 2018.

[21] 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.

[22] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[24] 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.

[25] B. Lin, F. Zhu, J. Zhang, J. Chen, X. Chen, N. N. Xiong, and J. L. Mauri, “A time-driven data placement strategy for a scientific workflow combining edge computing and cloud computing,” *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 4254–4265, 2019.

[26] N. Carlsson and D. Eager, “Ephemeral content popularity at the edge and implications for on-demand caching,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1621–1634, 2016.

[27] H. Shen, “An efficient and adaptive decentralized file replication algorithm in p2p file sharing systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 6, pp. 827–840, 2009.

[28] M. Badov, A. Seetharam, J. Kurose, V. Firoiu, and S. Nanda, “Congestion-aware caching and search in information-centric networks,” in *Proceedings of the 1st ACM Conference on Information-Centric Networking*, 2014, pp. 37–46.

[29] Z. Ming, M. Xu, and D. Wang, “Age-based cooperative caching in information-centric networking,” in *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2014, pp. 1–8.

[30] B. Yu and J. Pan, “Location-aware associated data placement for geo-distributed data-intensive applications,” in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 603–611.

[31] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, “Volley: Automated data placement for geo-distributed cloud services,” 2010.

[32] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, “Autoplacer: Scalable self-tuning data placement in distributed key-value stores,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 4, p. 19, 2015.

[33] X. Tang and J. Xu, “Qos-aware replica placement for content distribution,” *IEEE Transactions on parallel and distributed systems*, vol. 16, no. 10, pp. 921–932, 2005.

[34] 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*, 2009.

[35] 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.

[36] W. Hu, Z. Feng, Z. Chen, J. Harkes, P. Pillai, and M. Satyanarayanan, “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, pp. 161–170.

[37] E. A. Brewer, “Towards robust distributed systems,” in *PODC*, vol. 7, no. 10.1145. Portland, OR, 2000, pp. 343 477–343 502.

[38] N. Afonso, M. Bravo, and L. Rodrigues, “Combining high throughput and low migration latency for consistent data storage on the edge,” in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–11.

[39] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, “Web caching and zipf-like distributions: Evidence and implications,” in *IEEE INFOCOM’99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, vol. 1. IEEE, 1999, pp. 126–134.

[40] V. N. Padmanabhan and L. Qiu, “The content and access dynamics of a busy web site: Findings and implications,” in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2000, pp. 111–123.