Cheops, a service to blow away Cloud applications to the Edge

Marie Delavergne, Geo Johns Antony, Adrien Lebre

Cheops, a service to blow away Cloud applications to the Edge

Marie Delavergne, Geo Johns Antony, Adrien Lebre

RESEARCH REPORT
N° 9486
September 2022
Project-Team STACK

ISSN 0249-6399 ISRN INRIA/RR--9486--FR+ENG

Cheops, a service to blow away Cloud applications to the Edge

Marie Delavergne*, Geo Johns Antony*, Adrien Lebre †

Project-Team STACK

Research Report n° 9486 — September 2022 — 16 pages

Abstract:
One question to answer the shift from the Cloud to the Edge computing paradigm is: how distributed applications developed for Cloud platforms can benefit from the opportunities of the Edge while dealing with inherent constraints of wide-area network links?
Our solution to this question is to give the illusion of "single service images" spreading over the Edge infrastructure. Thanks to the modularity of micro-service based applications, one can deploy multiple instances of the same service (one per edge site) and deliver collaborations between them according to each request. This non-invasive approach is made possible by (i) a DSL that extends the application API and allows DevOps to program where/how the execution of each request should be executed, (ii) and its runtime, Cheops, a service that interprets and orchestrates each request in order to satisfy the geo-distribution parameters, allowing collaborations in a transparent manner for the underlying application.
We demonstrate the relevance of our proposal by illustrating how Cheops can successfully geo-distribute the Kubernetes vanilla code.

Key-words: Edge computing, Service composition, geo-distribution

* Inria
† IMT Atlantique

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Cheops, un service pour souffler les applications du nuage vers la périphérie

Résumé : Pour passer du paradigme de l'informatique en nuage à celui en périphérie, il faut se demander comment les applications distribuées développées pour les plates-formes d'informatique en nuage peuvent-elles bénéficier de l'informatique en périphérie tout en faisant face à ses contraintes inhérentes ?

Notre solution est de donner l'illusion d'une "un service à image unique" sur l'infrastructure en périphérie. Grâce à la modularité des applications basées sur les microservices, il est possible de déployer plusieurs instances d'un même service (une par site - en périphérie) et de fournir des collaborations entre elles en fonction des demandes des utilisateurs. Cette approche non invasive est rendue possible par (i) un DSL qui étend l'API de l'application et permet aux DevOps de programmer où et comment chaque demande doit être exécutée, (ii) et son service associé, Cheops, qui interprète et orchestre chaque demande pour satisfaire les paramètres de géo-distribution, permettant des collaborations d'une manière transparente pour l'application sous-jacente.

Nous démontrons la pertinence de notre proposition en illustrant comment Cheops peut géo-distribuer avec succès le code de Kubernetes sans avoir à le changer.

Mots-clés : informatique en périphérie, composition du services, géo-distribution

Contents

1 Introduction

Nowadays, there is an indubitable shift from Cloud Computing to the Edge [14]. The Internet of Things, smart cities, self-driving cars, augmented reality, are domains where the low latency plays a key role that can only be satisfied by Edge resources. However, the assumptions that are generally taken to develop Cloud applications are not valid anymore in the Edge context. For instance, the intermittent network connections should be considered as the norm in the Edge rather than the exception. If you consider the Google Doc service, users in the same vicinity cannot work on the same document if they cannot reach the data center, even though they are close to each other.

To reckon with the Edge constraints, and thus be able to satisfy requests locally at least, the most straightforward approach is to deploy an entire, independent instance of the application on every Edge sites. This way, if one site is separated from the rest of the network, it can still serve local requests. The next step is to offer collaboration means between these instances when needed. Git is an application that fulfills such requirements (even though it has not been designed specifically for that paradigm): Git operations can be performed locally and pushed to other instances when required.

In [5], the authors proposed the premises of a generalization of these Git concepts by presenting how an application can be geo-distributed without intrusive changes in its business logic thanks to a service mesh approach. A service mesh is a layer over micro-services that intercepts requests in order to decouple functionalities such as monitoring or auto-scaling [8]. Their initial idea was to rely on a service mesh to orchestrate any kind of requests between the various service instances each time it was needed. Concretely, they proposed to leverage the modularity and REST API's of micro-services based cloud applications to allow collaborations in an agnostic manner between multiple instances of the same system. They demonstrated the relevance of their proposal on top of OpenStack, a software composed of more than 100 hundred services.

In this paper, we extend their proposal to deliver a complete framework that allows multiple instances of a Cloud micro-service based application to behave like a single one. Thanks to our framework, entitled *Cheops*, DevOps can *share*, *replicate*, and *extend* resources between the different instances in agnostic manner. A service managed by Cheops can be seen as a *Single Service Image*. We found this analogy with past activities on Single System Images [9] relevant as *the interest of SSI clusters was based on the idea that they may be simpler to use and administer*. With Cheops, the challenge related to the collaboration between multiple instances of a system (the geo-distribution aspects) is reified at the level of the DevOps and externally from the business logic of the system itself.

The contributions of this article are as follows:

- A non intrusive approach relying on service mesh concepts to achieve three kind of collaborations between services: *sharing*, *replication*, and *cross*.
- A model of the different kind of relationships between resources that may exist in a micro-service based applications.
- A detailed description of the current Cheops prototype.
- A demonstration of the feasibility of the proposed framework and collaboration strategies in the Kubernetes ecosystem.

The rest of this paper is organized as follows: Section 1 motivates the current work. Section 2.2 presents the generalization idea of our proposal, while Section 3.4 deals with the current architecture we followed to implement our proof-of-concept. Section 5 presents related works. Finally, Section 6 concludes and discusses future work.

2 Motivations

2.1 Background

The Edge infrastructure we consider consists in globally distributed small data centers at multiple Edge sites (e.g., an airport, a city, a region, etc.). Between these sites, the round trip time can range from a few to hundreds of milliseconds, depending on the radius of the Edge infrastructure (metropolitan, national, global, etc.), with varying throughput constraints (LAN vs WAN, wired vs wireless links). More importantly, disconnections between Edge computing/storage units are expected to happen a lot more often than in Cloud infrastructures, leading to network split-brain situations [10].[1]

In order to deal with these specifics, applications in Edge computing have to manage the geo-distribution of resources themselves [13], following two major principles introduced in [5]:

Local-first: Minimize communications between sites and be able to deal with network partitioning issues by continuing to serve local requests at least.

Collaborative-then: Be able to take advantage of the different sites according to users' needs and infrastructure considerations.

Some approaches have been proposed in the literature to enable cloud applications to deal with the geo-distribution of the infrastructure:

- Revise the code to make it collaborative, on a service-to-service level [4];
- Use geo-distributed databases [1, 7], keeping in mind consistency challenges;
- Implement brokers for each service composing the application[2] [6].

Make the code *natively collaborative* requires tedious efforts of coding and maintenance. Additional efforts are required to revise and extend a code that already exists.

Database approaches have been proposed to make such distributed developments easier. However, diving in details, they tangle geo-distribution concerns in the code and consider resources as data only; missing their side effect (for example, adding a network through an infrastructure manager) [5].

Finally, the broker approach is interesting because it allows the externalization of collaboration aspects from the business code. However, it requires a lot of development effort also: in addition to requiring a broker stub for each service, each broker should implement once again a lot of what was made originally in the service (because the broker needs to offer the same capabilities as the underlying service in a geo-distributed and transparent manner).

2.2 Answering the geo-distribution principles

To mitigate the brokering efforts, the authors in [5] proposed to rely on a service mesh and only reify locality aspects at the programmer level to let them specify how each request should be handled throughout the different instances.

Service based applications usually are composed of services that fulfill one *functionality* and communicate between each other through REST API's to complete all the functionalities of the application. This modularity allows to use the same service located on another instance, or

[1] We underline we do not consider disconnections between users and their Edge location. Edge elements are supposed to be as close as possible to prevent this situation.

[2] https://github.com/kubernetes-sigs/kubefed Accessed 2022-07-06

another version of a service (newer or older), or the same service in the same location but on a different server (for load-balancing purposes), or even an entirely different service, as long as all these services expose the same API and have the same logic.

With a service-mesh intercepting messages going from one service to another, the authors demonstrated that it is possible to satisfy the two geo-distribution principles: (i) deploy one instance of each service on each edge site (this allows the use of the entire application locally, even if there is a network split) (ii) redirect requests between the different instances each time a remote resource is required. We extend this initial proposal and finalize the development of two new collaborations: replication and cross. Replication increases the robustness in case of network split brain while cross allows the extension of a resource between multiple instances; giving the illusion of a single service image. With these three collaborations, we claim we captured the operations that are mandatory to push a Cloud application to the Edge without requiring intrusive changes in their business logic.

3 Towards a Generalization of Collaborations

After a brief overview of *scope-lang*, the DSL presented in [5], we introduce the three collaborations that we consider in Cheops. We complete this section by introducing the relationship model that allows *Cheops* to handle dependencies between resources while performing the collaborations.

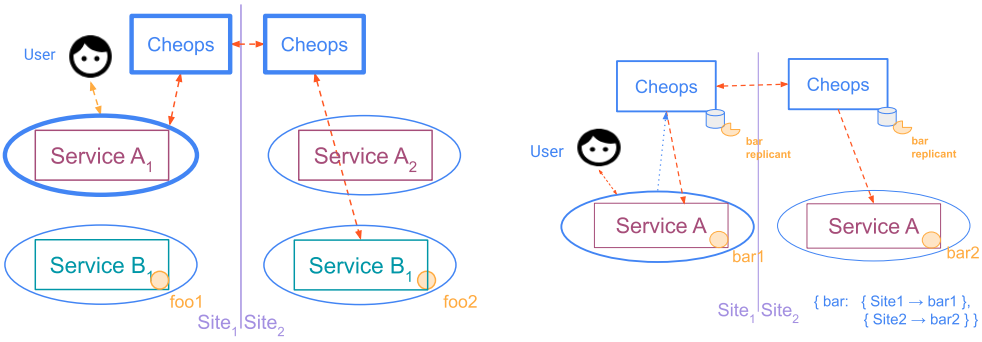
3.1 Scope-lang

Scope-lang is a language that extends the usual requests made from a user to its application in order to reify the locality aspects. A scope-lang expression (which we call *scope*) contains information on the location where a specific request, or part of the request, will be executed. As an example, a scope defined as "$s : App_1, t : App_2$" specifies to use service s from App_1 (the application *App* on Site 1), and the service t from App_2 (on Site 2). A more formal definition of the language is available in Figure 1. Users defines the scope of the request to specify the exact collaboration between instances required for the execution of their request. The scope

$$
\begin{array}{lcll}
App_i, App_j & ::= & \text{application instance} & \\
s, t & ::= & \text{service} & \\
s_i, t_j & ::= & \text{service instance} & \\
Loc & ::= & App_i & \text{single location} \\
 & | & Loc\&Loc & \text{multiple locations} \\
 & | & Loc\%Loc & \text{cross locations} \\
\sigma & ::= & s : Loc, \sigma & \text{scope} \\
 & | & s : Loc &
\end{array}
$$

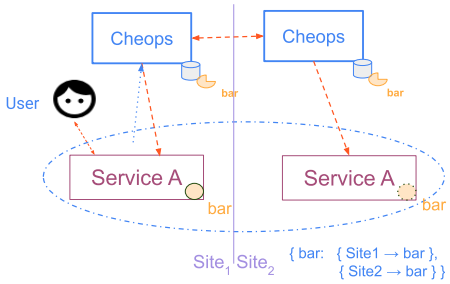
$$
\begin{aligned}
\mathcal{R}[\![s : App_i]\!] &= s_i \\
\mathcal{R}[\![s : Loc\&Loc']\!] &= \mathcal{R}[\![s : Loc]\!] \text{ and } \mathcal{R}[\![s : Loc']\!] \\
\mathcal{R}[\![s : Loc\%Loc']\!] &= \mathcal{R}[\![s : Loc]\!] \text{ spread to } \mathcal{R}[\![s : Loc']\!]
\end{aligned}
$$

Figure 1: Scope-lang expressions σ and the function that resolves service instance from elements of the scope $\mathcal{R}$.



(a) Sharing (uses services instances from different sites)

(b) Replication (replicates an operation on multiple instances)



(c) Cross (gives the illusion that multiple instances behave as a single one)

Figure 2: The different Cheops collaborations

is interpreted during the execution of the request workflow to choose the execution location accordingly. Scope-lang has been extended to allow cross collaborations.

3.2 Collaboration implementations (elementary resource)

Figure 2 depicts the three collaborations implemented in *Cheops*.

3.2.1 Sharing

Sharing is the collaboration which allows a service instance to use a resource from a service which is not the one assigned to its application instance.

The typical example is getting a resource from a service B on another site for a service A as presented by the red arrows in Figure 2a:

`application create a -sub-resource foo2 -scope {A:` $Site_1$`, B:` $Site_2$`}`

1. A user requests to create a resource on service A from $Site_1$ (Service A_1), using a sub-resource foo_2 from service B on $Site_2$ (Service B_2).
2. The request is intercepted and transferred to Cheops.
3. Cheops extracts the scope from the request and interprets it.
4. Cheops transfers the request to service A, that executes it until it needs the sub-resource.
5. The outgoing request is intercepted, and at this point, is transferred to $Site_2$.

6. Cheops on $Site_2$ uses its catalog to find Service B endpoint locally and transfers the request to get $foo2$.

7. The service response (containing the resource itself) is finally transferred back to Service A through Cheops.

Similarly to a local failure, if $ServiceB_2$ is not reachable, the request cannot be performed.

3.2.2 Replication

Replication is the ability for users to create and have available resources on different Edge sites to deal with latency and split networks. Replication main action is duplication: transfer the request to every involved sites and let the application execute the request locally. The operation does not simply consists though in forwarding the request to the different instances. Cheops keeps track of the different replicas in order to ensure that future CRUD operations achieved on any replica will be applied on all copies, maintaining eventually the consistency over time. To do that, Cheops relies on a data scheme, called the replicant, that links a meta-ID to the different replica IDs and their locations.

Figure 2b sums up the workflow to create a replicated resource on two sites:
`application create a -name bar -scope {A:` $Site_1$ `&` $Site_2$`}`.

1. A user sends a request on $Site_1$ to create two replicas of the *bar* resource.

2. The request is intercepted and transferred to Cheops.

3. Cheops extracts the scope and interprets it.

4. Cheops creates the replicant, and passes the request to create it to Cheops on the other involved site ($Site_2$), as well as the request of creation of *bar* on both sites, which is simply the request without the scope.

5. Both Cheops execute the request of creation; the response is intercepted to fill the local IDs on the replicants and the response is transferred to the user, replacing the local ID by the meta-ID of the replicant.

To provide eventual consistency, Cheops follows the Raft protocol, with one replicant acting as the leader.

3.2.3 Cross

Cross is the last collaboration we identified. The idea is to create a resource over multiple sites. The main difference with respect to the aforementioned replication concept is related to the aggregation/divisibility property. In the replication, each copy is independent, even if they all converge eventually based on the CRUD operation. A cross resource can be seen as an aggregation of all resources that constitutes the cross-resource overall. Some resources which cannot be divided by an application API will require an additional layer in the business logic to satisfy the divisibility property.

Similarly to the replicant data scheme, Cheops keeps tracks of the different resources in order to perform CRUD operations in the expected manner. A `CREATE` operation for instance can distribute the resource over different sites (if this resource is divisible), while a `REQUEST` will be performed on each ”sub resource” composing the cross-resource in order to return the aggregated result.

How Cheops deals with split brain issue for cross-resource is left as future work. However, it is worth noting that the unreachability of one site that hosts a part of the cross-resource faces multiple challenges.

An illustration of Cross is depicted in Figure 2c:
`application create a -name bar -scope {A:` $Site_1$`%` $Site_2$`}`.

1. A user sends a request to create a resource specifying the involved sites.
2. The request is intercepted and transferred to Cheops.
3. Cheops extracts the scope and interprets it.
4. Cheops creates the resource on the first site ($Site_1$) and passes the request to other involved sites.
5. Cheops on $Site_2$ identifies the extended resource and creates an identifier within Cheops to forward to the deployed resource site

3.3 Relationship model

Many resources have dependencies on each other (a virtual machine in the OpenStack ecosystem depends on an image, a network, an IP, etc.; a deployment file in Kubernetes is linked to several pods; etc.). Hence, it is mandatory to rely on a relationship model for replication and cross operations. This model will be used to keep track on each critical resource and ensure that CRUD operations are performed thoroughly.

We have identified and formulated three dependencies, depicted in Figure 3.

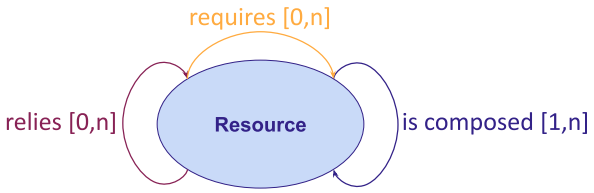


Figure 3: The different dependencies

3.3.1 Requirement

Requirement defines a relationship between two resources that is not critical for the survival of either of the resource but rather is a necessary link during a particular operation. The operation can be any operation performed upon either of the resource. While performing the operation the link is vital and if the link is severed the resultant operation will terminate and it will not succeed. If the link is maintained and no external factors affect the operation, the operation will be a success and after this the link between these resources is insignificant. Hence, a broken link after the operation does not affect either of the resource. An example for OpenStack is a VM requires an image, for the creation operation.

3.3.2 Reliance

Reliance defines a relationship between two resources that is critical for the survival of either one of the resource or both. If the link between these resources is cut at some point during the lifetime of these resources it will impact the existence of the resources and can lead to a failure condition. Involved resources are independent and one resource cannot alter the other resource. For example, in Kubernetes, a pod, when created with a secret, relies on this secret.

3.3.3 Composition

Composition consists of intrinsic dependencies between resources: the life cycle of the two resources are linked. The creation of resource A implies the creation (and respectively the destruction) of resource B. Composition is obviously wider than just two resources as one resource can

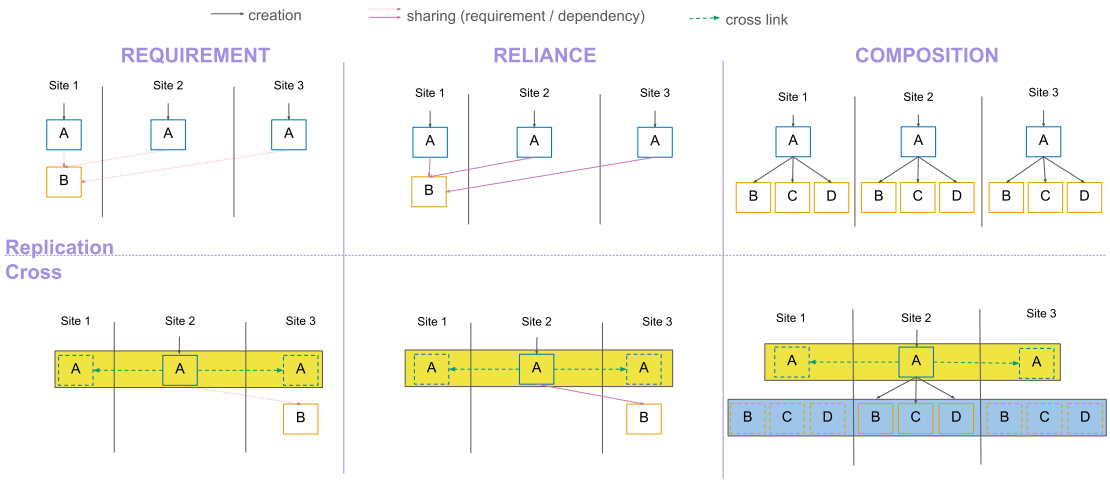


Figure 4: Behaviors to observe following the dependencies

be linked to a collection of other resources, which in their turn can also depend on sub-resources. For example, in OpenStack a stack can be composed of VM's, and in Kubernetes, a deployment is composed of pods.

3.4 Creation patterns for replication/cross operations

As mentioned, the goal of the relationship model is to ensure that Cheops operations are done thoroughly when the manipulated resource is not elementary, but depends on other resources. We discuss in this paragraph the various cases.

Requirement For a replication or a cross scenario, the user have the choice to first replicate B everywhere A will be; in this case, the creation of A can be executed without specifying the location of B, it will be executed locally on each site. The other choice is to specify the dependency in the creation request, which is represented in Figure 4.

1. Using the sharing operator in the scope, the user specifies that a resource B required is on $Site1$.
2. Cheops intercepts the request to get a resource from another site when it will be sent by the service needing it.
3. Cheops transfers the request to get resource B from $Site1$.
4. Resource B is received and the usual flow is executed.
5. Since Resource B is only required for some operations, this dependency is stored in Cheops database for further usage (in these operations).

Reliance This relationship follows a similar approach from requirement for replication and cross. The difference is with the involvement of Resource B through the life span of Resource A. At any point if Resource B fails for both collaborations, resource A will result in a failure state. The primary objective being to preserve the strong relationship between the resources A and B, Cheops needs to ensure the reachability of Resource B.

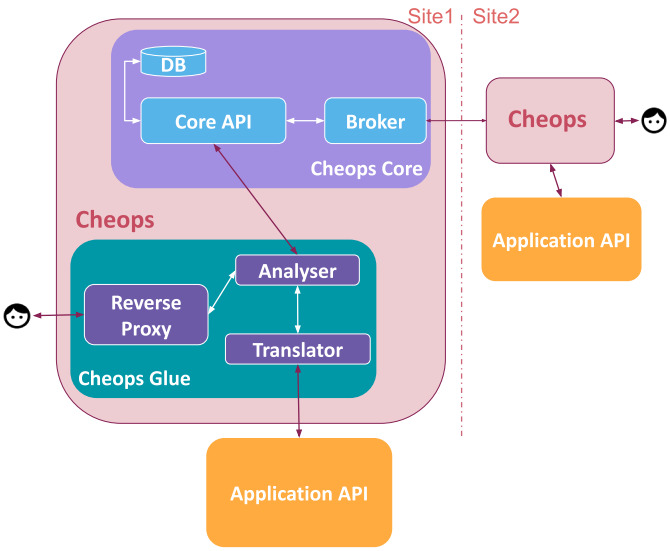


Figure 5: Cheops architecture

As before, a user can still replicate Resource B to ensure that Resource A will not suffer from a network partition. Otherwise, the process is the same as the requirement, except for: first, the dependency information needs to be stored in Cheops database for resource B to warn users against resources failures in replicas in case of a deletion of B. Second, Cheops needs to warn the users of affected resources (replicas of resource A) in case of network partition that affects B, because they will be in a failure state.

Composition For replication scenario, a copy of resource A is created on the involved sites which in turn creates resources B, C and D on each of these sites with a *cascading effect* from the normal, local execution of the creation of A. An update on resource A for a secondary layer resource B, C or D is propagated across the involved sites and also follows normal execution on each site. Network split brain is managed through the Raft protocol that ensures eventual consistency between all replicas.

For cross scenario, the process is similar and will also follow a cascading approach. Each compound resource will be created in a cross manner.

4 Cheops

The global architecture of our proof-of-concept is depicted in Figure 5. Cheops follows a modular approach, composed of various microservices, which are linked together through REST API protocol. There is one Cheops agent per site and agents monitor known Cheops agents through heartbeats.

4.1 Cheops internals

Cheops agents are divided into two main components which are Cheops Core API and Cheops Glue.

Cheops Core is the primary building block of a Cheops agent. This framework encapsulates the communication module, interface module and the database. The communication module provides the link between multiple Cheops instances, thus creating and maintaining a service mesh around the various involved cluster. Since Cheops focuses on minimising intrusive code changes of a deployed application, this module plays a significant role while addressing the collaboration feature of Cheops which includes various locations. This module also talks with the the core API module in the core. Core API is a management module created for the framework that acts as a service which interconnects all the services inside Cheops.

Cheops Glue is the second module of the Cheops framework. This module is designed to help Cheops Core translate Cheops API requests into the respective application API and vice-versa. Core is designed to handle agnostic API requests which are irrespective to all the applications. Cheops needs to convert these requests into application understandable API patterns. Since each application has its own pattern for intercepting API, Glue is developed with respect to individual applications such as Openstack, Kubernetes, etc. Glue acts as a first layer communication between users and Cheops by intercepting the data from the default CLI from the respective application using scope-lang. The analyser service in Glue evaluates the request and converts it into a generic request understandable by the Cheops core API service. On the contrary, Cheops Glue provides a translator service which converts the requests received from the Cheops Core API service. Cheops Glue also manages the creation of extra business logic for divisibility property of cross collaboration (see Figure 2c) for specific types of resource. It also handles the network requirements and implements the relationship model.

The core design of Cheops architecture is focused on a modular design which makes the framework distributed and easy to manage. The framework provides the flexibility of enhancing each component or individual service to scale it during peak usages. It is available for any request specifying the usage of its hosting site when it is not disconnected from the network and the DevOps are also able to control this instance locally, making it available at any time. Cheops obviously offers the three collaborations proposed earlier. It manages each collaboration without the need to modify anything in the deployed application. Finally, it maintains the consistency of the deployed applications across the locations.

4.2 Validation

We demonstrated the correctness of our proposal on the Kubernetes ecosystem. The feasibility for the collaborations were studied for replication and cross operations:

- For replication, we manipulated replicated pods across two sites.
- For cross, we analyzed the creation of a cross namespace and performed a few operations to validate the existence of the namespace across the two sites.

Experiments have been performed over two sites of the Grid'5000 experimental testbed [2] (Rennes and Nantes). These instances were completely independent of each other and local to the infrastructure. On each site, we deployed a Kubernetes cluster, composed of one master and one worker node, as well as a Cheops agent.

The goal of the experiments we performed was to validate the expected behaviour. Table 1 and Table 2 presents the results.

Operation	Location	Result
kubectl create pod purple –scope{Site1&Site2}	Site1	Pod *purple* created on Site1 and Site2
kubectl create pod violet –scope{Site1&Site2}	Site2	Pod *violet* created on Site1 and Site2
kubectl get pod violet	Site1	Pod *violet* from Site1 is displayed
kubectl get pod violet	Site2	Pod *violet* from Site2 is displayed

Table 1: Replication Kubernetes CLI requests

Operation	Location	Result
kubectl create ns foo –scope{Site1%Site2}	Site1	Namespace created on Site1 and Site2
kubectl create pod blue -n namespace foo –scope{Site1}	Site2	Pod created under namespace *foo* in Site1 extended to Site2
kubectl create pod yellow -n namespace foo –scope{Site2}	Site1	Pod created under namespace *foo* in Site2 extended to Site1
kubectl get pods -n namespace foo	Site1	Shows all resources from *foo* namespace from Site1 and Site2
kubectl create pod yellow -n namespace foo	Site1	Error: Pod already exist in Site2 under namespace *foo*

Table 2: Cross Kubernetes CLI requests

For replication, as presented in Table 1, we created two sets of pods *purple* and *violet*, each replicated on both sites. Then we requested the pods *violet* from each site to check if they were present. Though this tests basic functionalities, we were able to ensure that the *create* and *get* operations were working as intended. In the near future, we intend to test *update* and *delete*, as well as scenarios on the behavior of replicas in case of network split to check the consistency. A study on different types of resources to better check genericity is also required.

For cross, as presented in Table 2, we tested the creation of a namespace *foo* spanning across two sites. Then, two pods, namely *blue* and *yellow* were deployed on each sites in virtually the same namespace *foo*. These creations of pods were made from one site to the other, which validates the fact that the namespace is available on both site and we were able to forward the request to another site. The fourth operation, executed from one site, gets all pods associated with the cross namespace *foo*. As expected, this operation returns both pods (*blue* and *yellow*), showing this namespace is indeed spanning on both sites. Finally, the creation of pod *yellow* in this same namespace is not working, as expected, because, even though the pod was created on another site, a pod with this name already exists in the cross namespace *foo*.

5 Related Works

Popular solutions such as Rancher[3], Volterra[4] geo-distribute a service to the Edge with a centralised approach. In these approach, sites are not autonomous and additional details are added to the business logic, while our solution focuses on forming a decentralized set of autonomous

[3] https://rancher.com Accessed 2022-07-06
[4] https://medium.com/volterra-io/tagged/kubernetes Accessed 2022-03-20

application instances as well as avoiding changing the application code.

Google Anthos[5] is another framework which provides a way to collaborate and geo-distribute resources. It creates a single source of truth and maintains it to create deployments to multiple geo-distributed locations. Once again, relying on this single point of truth is not suited in an Edge infrastructure, with intermittent network connections, where our P2P solution focuses on creating multiple sources in order to provide autonomy to each locations.

Istio[6] is a prominent service mesh solution for Kubernetes. It uses a sidecar mechanism to intercept all the requests at the resource level. This implies each pod will need a sidecar attached to it. Even though it uses a lightweight Envoy, in a large configuration, it can be a overhead to the cluster. Cheops creates a single interception mechanism for all the requests, whether they are defined by the users or the application. This approach also allows DevOps to explicit the execution location of their requests.

MiCADO-Edge [15] framework aims at extending a Cloud orchestrator in the context of Edge infrastructures. The objective of this framework is to integrate non-cloud resources into a centralised Cloud. It uses the KubeEdge [16] solution to orchestrate between the control plane and worker nodes. KubeEdge and MiCADO Framework provides primarily a centralised approach to manage the resources without the Edge sites autonomy of our solution.

Mck8s [12] is another similar solution with geo-distribution application deployments. The primary objective being to deploy applications onto other locations including Edge sites. The solution is quite complex as it creates a wrapping around on KubeFed [7]. The centralised management and overhead on the business logic from KubeFed contradicts our approach of being decentralised and no change in business logic.

Hybrid control planes such as OneEdge [11] focus on enabling control over applications for Edge sites. The uniqueness of this approach is it brings autonomy to the Edge sites. This solution is quite similar to our proposal but the centralised aspect still makes this solution an unsuitable candidate for us.

TOSCA [3] is a framework which generalizes the transformation of an application to its respective Cloud provider based on the resource format. This pattern is being used by more frameworks where this solution tries to make this as a new standard on top of the business logic. Our approach is quite different from TOSCA since we are not aiming to create a new standard pattern to make a service a geo-distributed, and our approach functions for existing applications without modifying their code.

6 Future Work and Conclusion

In this paper we presented our service-mesh like framework, Cheops, that allows Devops to geo-distribute a micro-services based application without requiring intrusive changes in the business logic. This service mesh relies on these applications modularity and the deployment of instances of the application on each site composing the Edge infrastructure.

Cheops relies on scope-lang, a DSL previously introduced to allow users to explicit the execution location of their requests as well as the type of collaborations between the different service instances. To ensure the correctness of each collaboration, Cheops relies also on a relationship model we introduced. Finally, we demonstrated the relevance our approach on simple collaborations across different sites.

[5] https://cloud.google.com/anthos Accessed 2022-07-06
[6] https://istio.io Accessed 2022-07-06
[7] https://github.com/kubernetes-sigs/kubefed Accessed 2022-07-06

We are currently working on a model to introduce patterns for application code for cross collaboration which requires a step further. For example, if a resource is divisible at the API level the ad-hoc code required might be easier than for those which are not divisible at the API level. Intermittent networks are a major factor which can effect a completely geo-distributed management framework. Our main focus is working on solutions to adapt better in case of such failures occur. In case of cross, dedicated code is also required to bring back the entire resource availability on the different sites. Finally, one future area of focus will be to add control loops in Cheops in order to optimize the placement of resources. In our current version of the approach, DevOps need to manually specify the location, and finding the optimal site may be an additional overhead. By leveraging control loops, it might be possible to automatically recreate, relocate, etc. resources in order to cope with application or infrastructure changes.

Our approach is meant for Cloud applications based on (micro)services, which communicates through a REST API. Though it is restrictive to this set of applications, this generic approach is meant to include all of those. We are convinced that our approach to bring existing Cloud applications to the Edge without entangling any geo-distribution code in the business logic is crucial to stimulate the shift to include Edge sites in the global capabilities of the Cloud.

References

[1] Anshul Ahuja, Geetesh Gupta, and Subhajit Sidhanta. Edge applications: Just right consistency. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 351–3512. IEEE, 2019.

[2] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[3] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Tosca: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.

[4] David W Chadwick, Kristy Siu, Craig Lee, Yann Fouillat, and Damien Germonville. Adding federated identity management to openstack. *Journal of Grid Computing*, 12(1):3–27, 2014.

[5] Ronan-Alexandre Cherrueau, Marie Delavergne, and Adrien Lebre. Geo-distribute cloud applications at the edge. In *EURO-PAR 2021-27th International European Conference on Parallel and Distributed Computing*, 2021.

[6] Abdessalam Elhabbash, Faiza Samreen, James Hadley, and Yehia Elkhatib. Cloud brokerage: A systematic survey. *ACM Comput. Surv.*, 51(6), jan 2019.

[7] Adrien Lebre, Jonathan Pastor, Anthony Simonet, and Frédéric Desprez. Revising openstack to operate fog/edge computing infrastructures. In *2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, pages 138–148, 2017.

[8] W. Li et al. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.

[9] Renaud Lottiaux, Pascal Gallard, Geoffroy Vallée, Christine Morin, and Benoit Boissinot. Openmosix, openssi and ubeerrighed: a comparative study. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 1016–1023. IEEE, 2005.

[10] A. Markopoulou et al. Characterization of failures in an operational ip backbone network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, Aug 2008.

[11] Enrique Saurez, Harshit Gupta, Alexandros Daglis, and Umakishore Ramachandran. Oneedge: An efficient control plane for geo-distributed infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 182–196, 2021.

[12] Mulugeta Tamiru, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. mck8s: An orchestration platform for geo-distributed multi-cluster environments. In *ICCCN 2021-30th International Conference on Computer Communications and Networks*, 2021.

[13] Genc Tato et al. Split and migrate: Resource-driven placement and discovery of microservices at the edge. In *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 9:1–9:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[14] Tuyen X Tran, Abolfazl Hajisami, Parul Pandey, and Dario Pompili. Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges. *IEEE Communications Magazine*, 55(4):54–61, 2017.

[15] Amjad Ullah, Huseyin Dagdeviren, Resmi C Ariyattu, James DesLauriers, Tamas Kiss, and James Bowden. Micado-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum. *Journal of Grid Computing*, 19(4):1–28, 2021.

[16] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399