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[1 [0000−0001−8020−959X]], Geo Johns Antony[1 [0000−0002−9129−8281]], and Adrien Lebre[1 [0000−0002−0305−4130]]

Inria, Nantes, France firstname.lastname@inria.fr

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? Leveraging the modularity of microservice-based applications, we propose to deploy multiple instances of the same service (one per edge site) and deliver collaborations between them according to each request. Collaborations are expressed thanks to a DSL and orchestrated in a transparent manner by the Cheops runtime. We demonstrate the relevance of our proposal by geo-distributing Kubernetes resources.

Keywords: Edge computing, Service composition, Service mesh

1 Introduction

Nowadays, there is an indubitable shift from Cloud Computing to the Edge [8]. 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 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 datacenter, even though they are close to each other. To reckon with the Edge constraints, and thus be able to satisfy requests locally, 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[1]. 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 this paradigm): Git operations can be performed locally and pushed to other instances when required.

We 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 [3]. A service mesh is a layer

[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.

over microservices that intercepts requests in order to decouple functionalities such as monitoring or auto-scaling [4]. Concretely, we proposed to leverage the modularity and REST APIs of cloud applications to allow collaborations in an agnostic manner between multiple instances of the same system.

In this paper, we extend our proposal to deliver a complete framework that allows multiple instances of a Cloud microservice based application to behave like a single one. Thanks to our framework, called *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 [5] 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 nonintrusive 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 microservice 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 2 presents the generalization idea of our proposal, while Section 3 deals with the current architecture we followed to implement our proof-of-concept. Section 4 presents related works. Finally, Section 5 concludes and discusses future work.

2 Towards generic and noninvasive collaborations

2.1 Scope-lang

Scope-lang is a language introduced in [3], that extends the usual requests made from users to their 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). Users defines

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

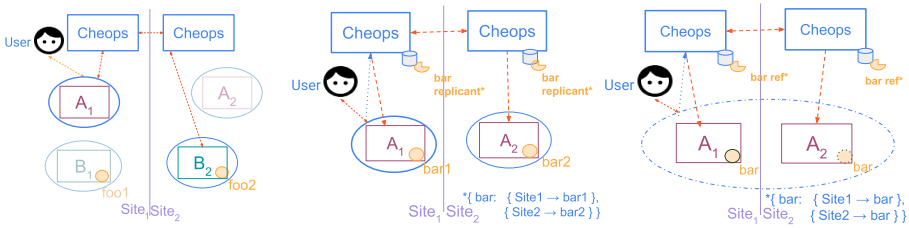
$$
\begin{aligned}
\mathcal{R}[\![s : App_i]\!] &= s_i \\
\mathcal{R}[\![s : Loc\&Loc']\!] &= \mathcal{R}[\![s : Loc]\!] \text{ and } \mathcal{R}[\![s : Loc']\!] \\
\mathcal{R}[\![s : Loc\%Loc']\!] &= \mathcal{R}[\![s : Loc]\!] \text{ spread to } \mathcal{R}[\![s : Loc']\!]
\end{aligned}
$$

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

the scope of the request to specify the exact collaboration between instances required for the execution of their request. The scope is interpreted during the execution of the request workflow to decide the execution location accordingly. A more formal definition of the language is available in Figure 1, which has been extended to allow cross collaborations.

2.2 Collaboration implementations for elementary resources

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



(a) Sharing (b) Replication (c) Cross

Fig. 2: The different Cheops collaborations

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, until A 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 and transfer the request to this service to get $foo2$.
7. The service response (containing the resource itself) is finally transferred back to Service A through Cheops.

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 in forwarding the request to the different instances, though. 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.

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 `READ` 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

2.3 Relationship model

Many resources have dependencies with 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.

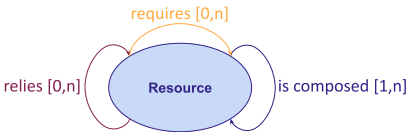


Fig. 3: The different dependencies

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 and while it is performed, the link is vital. If it is severed, the operation will terminate and 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.

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.

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, as one resource can 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 VMs, and in Kubernetes, a deployment is composed of pods.

2.4 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 sharing, 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.

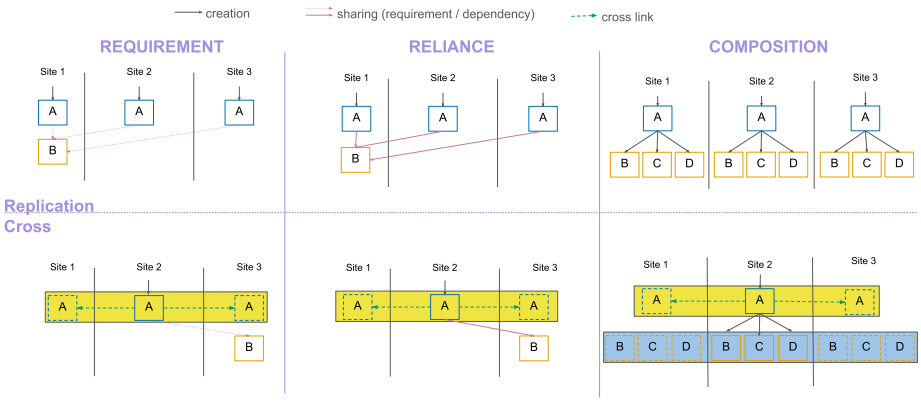


Fig. 4: Behaviors to observe following the dependencies

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 primary objective being to preserve the strong relationship between the resources A and B, Cheops needs to ensure the reachability of Resource B.

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.

3 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.

3.1 Cheops internals

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

Cheops Core consists of the communication, interface and database modules. The communication module creates a service mesh around the involved sites. This module also talks with 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.

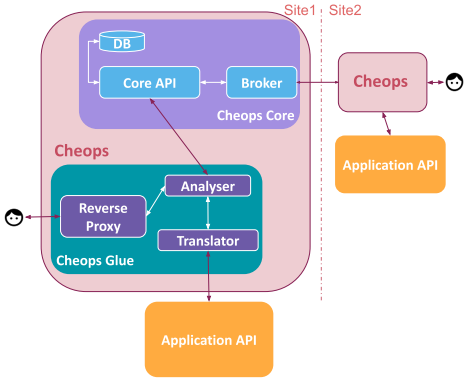


Fig. 5: Cheops architecture

Cheops Glue is designed to help Cheops Core translate Cheops API requests into the respective application API and vice-versa. Since each application has its own pattern for intercepting API, Glue helps in understanding these patterns and convert them to an agnostic API for Core. It is developed independently to an application. The analyser service in Glue evaluates the request from scopelang and converts it into a generic request understandable by the Core API service, while the translator do the reverse operation. Cheops Glue also manages the creation of extra business logic for divisibility property of cross collaboration for specific types of resource. It also handles the network requirements and implements the relationship model.

3.2 Validation

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

- For replication, we manipulated replicated pods across two sites.
- For cross, we created a 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 [1] (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 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

4 Related work

Popular solutions such as Rancher[2], Volterra[3] or Google Anthos[4] geo-distribute a service to the Edge with a centralised approach. In these approaches, sites are not autonomous, centralised single source of truth and additional details are added to the business logic. Our solution focuses on forming a decentralized P2P set of autonomous application instances without changing the application code.

Istio[5] uses a sidecar mechanism to intercept all the requests at the resource level. It creates an individual sidecar or a broker for each service as opposed to Cheops which manages all microservices at each site with a single broker.

MiCADO-Edge [9] extends cloud operator to edge. It uses the KubeEdge [10] to orchestrate. Mck8s [7] is another similar solution with geo-distribution application deployments. The solution is created as a wrap around KubeFed [6]. These Framework provides primarily a centralised approach to manage the resources without the Edge sites autonomy like our solution.

Hybrid control planes such as OneEdge [6] brings autonomony to the Edge. This solution is quite similar to our proposal but the centralised aspect still makes this solution an unsuitable candidate for us.

TOSCA [2] is a framework which standardises a deployment pattern across platforms. Our solution aims not at avoid creating any additional code, but to allow collaborations between sites without changes in code.

5 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 following a REST API 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 we 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

[2] https://rancher.com Accessed 2022-07-06
[3] https://medium.com/volterra-io/tagged/kubernetes Accessed 2022-03-20
[4] https://cloud.google.com/anthos Accessed 2022-07-06
[5] https://istio.io Accessed 2022-07-06
[6] https://github.com/kubernetes-sigs/kubefed Accessed 2022-07-06

each collaboration, Cheops relies also on a relationship model we introduced. Finally, we demonstrated the relevance our approach on simple collaborations across different sites.

We are currently working on a model to introduce patterns for application code for cross collaboration which requires a step further. As discussed above, we also need to focus on tolerance for intermittent networks across collaborations. 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.

We claim 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. Balouek, D., et al.: Adding virtualization capabilities to the Grid'5000 testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer International Publishing (2013). https://doi.org/10.1007/978-3-319-04519-1_1
2. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: Tosca: portable automated deployment and management of cloud applications. In: Advanced Web Services, pp. 527–549. Springer (2014)
3. Cherrueau, R.A., Delavergne, M., Lebre, A.: Geo-distribute cloud applications at the edge. In: EURO-PAR 2021-27th International European Conference on Parallel and Distributed Computing (2021)
4. Li, W., et al.: Service mesh: Challenges, state of the art, and future research opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). pp. 122–1225 (2019)
5. Lottiaux, R., et al.: Openmosix, openssi and kerrighed: a comparative study. In: CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005. vol. 2, pp. 1016–1023. IEEE (2005)
6. Saurez, E., Gupta, H., Daglis, A., Ramachandran, U.: Oneedge: An efficient control plane for geo-distributed infrastructures. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 182–196 (2021)
7. Tamiru, M., Pierre, G., Tordsson, J., Elmroth, E.: mck8s: An orchestration platform for geo-distributed multi-cluster environments. In: ICCCN 2021-30th International Conference on Computer Communications and Networks (2021)
8. Tran, T.X., Hajisami, A., Pandey, P., Pompili, D.: Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges. IEEE Communications Magazine **55**(4), 54–61 (2017)
9. Ullah, A., Dagdeviren, H., Ariyattu, R.C., DesLauriers, J., Kiss, T., Bowden, J.: Micado-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum. Journal of Grid Computing **19**(4), 1–28 (2021)
10. Xiong, Y., Sun, Y., Xing, L., Huang, Y.: Extend cloud to edge with kubeedge. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC). pp. 373–377 (2018). https://doi.org/10.1109/SEC.2018.00048