A service mesh for collaboration between geo-distributed services: the replication case

Marie Delavergne[1], Ronan-Alexandre Cherrueau[1], and Adrien Lebre[1]

LS2N, *Inria*, France - firstname.lastname@inria.fr

Abstract. Edge computing is becoming more and more present, with sites geo-distributed around the globe. Applications on these infrastructures must be able to manage the latency and disconnections inherent to their distribution. One way to deal with these concerns could be to deploy one entire instance of the application per site and use a service mesh to manage the collaboration between the geo-distributed instances. More precisely, we propose to reify the location of application instances in REST requests and allow redirections between these requests thanks to a dedicated language and a service mesh allowing three types of collaborations. This paper focuses on the replication of a resource between multiple instances. Though it is still a work in progress, we demonstrated the relevance of our approach in the OpenStack ecosystem.

1 Introduction

Edge computing is getting more important, with more and more small datacenters at the edge of the network. Nonetheless, lots of applications do not benefit from the geo-distribution of wide-area networks and are not designed to handle the high latencies and disconnections implied by these distributions [8]. To deal with these concerns, we advocate for the placement of an instance of the application on each site. This way, each site is autonomous and can fully work if disconnected from the rest of the network [2]. Unfortunately, the collaboration is still missing: instances are able to function by themselves, but they cannot collaborate between each other and so do not benefit from the geo-distribution.

To provide such a collaboration without changing the code, we propose to leverage the service mesh concept. Service meshes help cloud computing applications solve different problems with their built-in functionalities. For example, to improve overall performance, load-balancing is provided. More largely, by intercepting communications, they provide functionalities to ease different operations, like traffic monitoring, access control, fault tolerance [3]. In general, they are implemented with proxies as sidecars for the services, without interfering with their code as they only work on requests passing from services to services. Their ability to intercept and redirect communications offers an opportunity to orchestrate requests between endpoints of any instance of the same application.

In this paper, we propose Cheops, a service to use in combination with a service mesh to program on-demand collaborations between multiple instances of an application. To specify where a request will be executed at a fine-grained level, Cheops relies on the scope-lang proposal we initially developed [2].

Scope-lang extends applications API and allows the user to specify where (on which services) the request is executed. The language has been designed to provide different types of collaborations between application instances. For example, *sharing* is when a resource needed by a service has been created on another instance. This is the basic collaboration which allows to share resources between the instances. *Replication* allows operations on identical resources on different sites, to deal with availability of these resources in case of network partitioning or to improve overall performance. Finally, *cross* allows a resource to span across different sites. In this paper, we focus on *replication* (*sharing* has already been discussed [2], while *cross* is let as future work). By resources, we mean every entities managed by services, whether it is an entry in the database or something has complex and low-level as a network.

It is noteworthy that other frameworks or languages [4,7] have been proposed. However, they are invasive as they require to entangle geo-distribution in the business code. Non-invasive approaches generally follow the brokering approach: an entity is in charge of redirecting requests between the different instances. However, instances are not aware of the others. The goal of this proposal is to allow the instances to collaborate on-demand as if they were a single entity. Another way to see it is to allow the users to operate changes on resources on different sites through the API.

In this paper, we focus on how replication is interpreted and executed thanks to Cheops. We first explain scope-lang and how our general model works to allow DevOps to specify the location of a request execution. Then, we dive into the replication collaboration between different instances of the same service. In particular, how we manage replicas of a resource on different sites and how we handle disconnections, partitions or faults.

2 Scope-lang, a language to reify the geo-distribution of requests

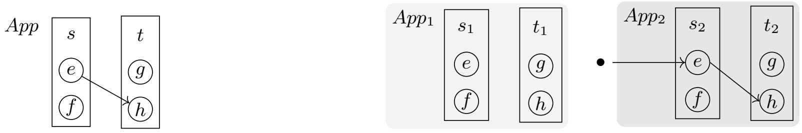
In this section, we dive deeper into how scope-lang, Cheops[1] and our general model work together to allow collaborations outside of the application, and so keep a clear separation of concerns.

2.1 General model

As a reminder, we have entire instances of the application on each site. Scope-lang parameters Cheops on a per-request basis in order to orchestrate collaborations between instances.

To explain collaboration between each instance of different services, let us take a look on how microservices based applications work. Each service composing the application exposes endpoints to communicate with other services. These endpoints are linked to a specific part of the business they achieve. When calling endpoints of other services, they form a workflow between services. For example,

[1] https://gitlab.inria.fr/discovery/cheops



(a) Application *App* made of two services s and t and four endpoints e, f, g, h. The $s.e \rightarrow t.h$ represents an example of a workflow.

(b) Two independent instances App_1 and App_2 of the *App* application. The • represents a client that executes the $s.e \rightarrow t.h$ workflow in App_2.

Fig. 1: Microservices architecture of a cloud application.

Fig. 1a shows an application *App* composed of two services s and t that expose endpoints e, f, g, h and one example of a workflow $s.e \rightarrow t.h$. Fig. 1b shows the instantiation of the application *App* on two different sites and their corresponding service instances: s_1 and t_1 for App_1; s_2 and t_2 for App_2. A client (•) triggers the execution of the workflow $s.e \rightarrow t.h$ on App_2. It addresses a request to the endpoint e of s_2 which handles it and, in turn, contacts the endpoint h of t_2.

2.2 Scope-lang

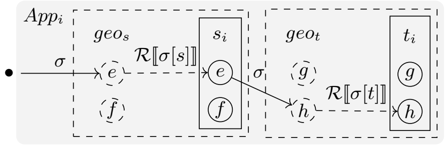
To parameter collaborations, we developed a domain specific language called scope-lang. A scope-lang expression (referred to as the *scope* or σ in Fig. 2a) contains location information that defines, for each service involved in a workflow, in which instance the execution takes place. The scope "$s : App_1, t : App_2$" intuitively means that the request must be achieved on the service s from App_1 and t from App_2. The scope "$t : App_1 \& App_2$" specifies to execute the request on the service t of App_1 and App_2. Users set the scope of a request to specify the collaboration between instances they want for a specific execution. The scope is then *interpreted* by a dedicated module entitled Cheops during the execution of the workflow to fulfill that collaboration. The main operation it performs is *request forwarding*. To be more precise, reverse proxies in front of each service

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

$$\mathcal{R}[\![s : App_i]\!] = s_i$$

$$\mathcal{R}[\![s : Loc\&Loc']\!] = \mathcal{R}[\![s : Loc]\!] \text{ and } \mathcal{R}[\![s : Loc']\!]$$

(a) scope-lang expressions σ and the function that resolves service instance from elements of the scope $\mathcal{R}$.


App_i
geo_s
$\mathcal{R}[\![\sigma[s]]\!]$
s_i
σ
e
f
e
f
σ
geo_t
g
h
$\mathcal{R}[\![\sigma[t]]\!]$
t_i
g
h

$\sigma = s : App_i, t : App_i$

(b) Scope σ interpreted by the geo-distribution service mesh *geo* during the execution of the $s.e \xrightarrow{\sigma} t.h$ workflow in App_i. Reverse proxies perform requests forwarding based on the scope and the $\mathcal{R}$ function.

Fig. 2: A service mesh to geo-distribute a cloud application

instance (geo_s and geo_t in Fig. 2b) intercept the request and interpret its scope to forward the request. "Where" exactly depends on locations in the scope.

The reverse proxy uses a specific function $\mathcal{R}$ (see Fig. 2a) to resolve the service instance at the assigned location. $\mathcal{R}$ uses an internal registry. Building the registry is a common pattern in service mesh using a *service discovery* [3].

In summary, scope-lang effectively parameters how Cheops will redirect the request. In the next section, we discuss how the replication is achieved.

3 Replication in Cheops

Replication is the ability to create and maintain identical resources on different sites: an operation on one replica should be propagated to the others, dealing with faults and disconnections and maintaining consistency based on our eventual model. Other consistency policies [1, 10] could be envisioned, but let as future work as they do not change the general concept of scope-lang/Cheops. To get a better understanding of the point of replication, imagine a user who needs a huge resource (like an ISO image) both at home and at work. The resource can be replicated at creation on both sites and it will be the only time when the entire resource will go through the network. This saves a lot of bandwith, and is especially useful if there is a partition between both sites.

3.1 Replication model

Modular applications based on microservices usually follow a RESTful HTTP API. In most cases, they generate an identifier for each resource, which will be used by the API to retrieve, update or delete it. When receigving a request to create replicas, Cheops unify these identifiers with a data model called *replicant*.

A replicant is simply a meta-identifier we generate along with a mapping $site \rightarrow local\_identifier$. A replicant can thus be implemented for example as: $meta\_identifier : [site_n : local\_identifier_n, ...]$. We only store the location (site) of the replica and not the service used since it is possible to deduce the service with the incoming request. This is subject to change depending of the evaluation of our prototype. We could store also the involved service and/or the type of resource involved.

These replicants are stored in a database co-located to the Cheops agents. A copy of the replicant is stored on each site where its replicas are (the sites involved in the replication). Cheops has an API of its own to allow the user to check the state of operations, sites and inspect replicants.

3.2 Architecture overview

Cheops agents are located on each instance site, with a reverse proxy besides every service transfering their requests to the agents. Agents communicate between each other and check each other status via heartbeats. Our implementation of Cheops uses Consul service mesh[2] and Envoy[3] as reverse proxy to intercept and

[2] https://www.consul.io/

[3] https://www.envoyproxy.io/

redirect, when needed, the requests. It is also worth noting that Envoy intercepts inbound and outgoing requests from services except for requests coming from Cheops agents.

In Fig. 3, we represented the reverse proxy and Cheops as one single entity that intercepts the request as it is in Fig. 2b to ease the comprehension.

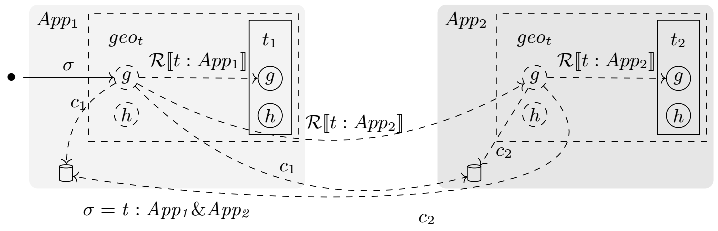

App_1
geo_t
t_1
σ
g
$\mathcal{R}[\![t : App_1]\!]$
g
c_1
h
h
$\mathcal{R}[\![t : App_2]\!]$
c_1
App_2
geo_t
t_2
g
$\mathcal{R}[\![t : App_2]\!]$
g
h
h
c_2
$\sigma = t : App_1\&App_2$
c_2

Fig. 3: Modelling of the replication by forwarding on multiple instances. c_1 arrows represents Cheops agent on App_1 updates to the databases, c_2 arrows the one from Cheops agent on App_2.

3.3 CRUD execution workflow

First, to define what is the creation, update or delete workflow, we have to define what they do in our consistency model and what are their boundaries. The creation of resources replicated in an eventual consistency implies that every replicas are identical at creation and will be created eventually. The update of resources created with the replication in an eventual consistency implies that all replicas will be updated eventually, whether the user specifies a scope or not in its request. It is the same for deletes.

The operation obviously begins when the user makes the request. But for the end, we could consider that an operation ends either when there is one response and is returned to the user, or when the operation is executed on every sites. In an eventual consistency model, the latter *end* can come a lot later than the first response. It is important to know what happens in case of failure (partition, disconnection, server failure) during the execution until the first response, but also after, because the operation must be executed on our replicas at some point.

In eventual consistency, since the first response to arrive goes to the user before it might be applied everywhere, it is the responsibility of the user to check with a request to Cheops or directly to involved services to know where the creation or updates are already applied. Users cannot assume because they received the answer the operation as already been applied everywhere.

Creation The replication process to create a resource on App_1 and App_2 happens as follows:

1. A request for replication is addressed to the endpoint of a service of one application instance. For example in Fig. 3: $\bullet \xrightarrow{t:App_1\&App_2} t.g$, where g is the endpoint for the creation of the resource managed by the service t.

2. The scope is extracted in the Cheops agent and the $\mathcal{R}$ function (from scope-lang) is used to resolve the endpoints that will store replicas. In Fig. 3: $\mathcal{R}[\![t : App_1 \& App_2]\!]$ is equivalent to $\mathcal{R}[\![t : App_1]\!]$ and $\mathcal{R}[\![t : App_2]\!]$. Consequently, t_1 and t_2 will be used for the resource creation.
3. The meta-identifier is generated and the replicant created using the meta-identifier and the location of execution. For example, if the generation yielded 72, we have: $\{$ 72 : $[App_1 : none, App_2 : none]\}$. The replicant on App_1 (where the request was made) becomes the leader of the replicants. A log is created for future operations on replicas, as well to make sure the creation will be applied on every involved sites.
4. Each request is forwarded to the corresponding Cheops agent on involved sites and a copy of the replicant is stored in the database on those sites simultaneously. In Fig. 3: geo_t forwards the request to $t_1.g$ and $t_2.g$ and stores the replicant $\{72 : [App_1 : none, App_2 : none]\}$ in App_1 and App_2 databases. In the figure, this is represented by the c_1 arrows going to the cylinders.
5. Each contacted service instance executes the request and returns the results to their local Cheops agent, which updates the replicant with the local identifier. In Fig. 3: t_1 and t_2 return their local identifier, e.g., 42 and 6.
6. Cheops agents then proceed to propagate the updated information to other agents involved. In parallel, they send the entire response to the Cheops agent that stores the leader replicant. In Fig. 3: the replicant is now $\{72 : [App_1 : 6, App_2 : 42]\}$ on App_1 and App_2 sites databases, thanks to the updates represented by c_1 and c_2 arrows.
7. When the agent where the leader is receives the first creation response, it transfers it to the user who asked for the replication, replacing the local ID with the replicant meta ID.

Read The process of reads is straightforward; to access a specific resource, users must either be on a site where one of its replicas is or specify in the scope on which location a replica of the resource to read is.

Update From now on, every request made to update (or delete) is filtered to check if the id given corresponds either to a replicant meta identifier or a local replica identifier. The process is quite similar to the creation, but does not generate a new replicant or change an existing one. It only applies an update to replicas.

1. A request for an update of a previously created replica is addressed to the endpoint of a service of one application instance.
2. Cheops checks if the ID in the request exists in a replicant. If not, the request is sent back to the service to be executed. If it is, the requests is transferred to the Cheops agent storing the replicant leader. It gets the corresponding replicant to find every replicas (and thus sites) involved. The operation is stored in its log.
3. The request is copied as many times as necessary (with the corresponding local identifier) and sent to the Cheops agent of involved sites.

4. Local Cheops agents send the request to the corresponding service on their site, which executes the request normally.
5. Each Cheops agent sends back the response to the Cheops agent where the replicant leader is.
6. This agent sends back the response to the user, once again, with the meta-identifier where the local-identifier would be expected to notify the user that the replicas were updated.

Delete As for the update, a delete on replicas can be identified either by a local identifier or the meta identifier. The process is identical as the update's.

3.4 Dealing with faults

We define a fault as: a partition of an involved site, or a failure from this site, whether it is shut down, out of order, or if the request cannot be executed for any reason (not enough memory to create a resource for example).

It is also important to mention that if the site where the user sent its request is faulty (does not work in any way), the request obviously cannot be executed. The user can make the request to a more distant site.

Moreover, the "during an operation" can refer to two distinct phases. As we discussed before, the end of an operation can be seen as: when a replica has been created/updated/deleted and the user has been notified, and when the operation is applied to all replicas. So "during an operation" is between the request of the user and before one of these end. In our consistency model, this conveys no difference to the process.

If a site fails where a replica is supposed to be, other Cheops will be informed due to its heartbeat (or rather lack of). Any other operation received by the leader will then be retried according to the log when the site comes back again. Therefore, a site is considered to be eventually available again unless it is removed. If a site is removed from the system, every replicant that were hosting a replica on this site must delete the site from their mapping (from the replicant). The leader will be in charge of this particular task.

Faults during operations The operation will be applied *eventually* on all involved sites. This eventual consistency uses a consensus protocol, and in our case, an implementation of Raft [6]. For example, the leader's log allows to replay operations that are not yet applied. It is the responsibility of the Cheops agent where the leader is to ensure that operations are applied eventually.

Faults while there are replicas When a site fails while there are replicas somewhere without any particular operation running, no heartbeat is received by other Cheops agent and the replica is considered unavailable temporarily.

If a site where a replica is was partitioned at some point but could be used locally, only read queries can be made, and these reads might be stale. When rejoigning the cluster, operations will be applied on the site so it is up-to-date thanks to the leader's log.

4 Discussion

In this paper, the service mesh and proxy mentionned were respectively Consul and Envoy, but this approach could work with other proxies or services meshes available such as Istio[4] or Open Service Mesh[5]. The approach differs from using a service mesh to redirect the requests with load-balancing or in case of failures by giving the users the ability to chose where their requests will be executed per-demand.

The users are thus responsible to trigger the request based on their needs and the availability of sites. In the case of a infrastructure such as OpenStack, this means that it gives back the DevOps the ability to decide where a request will be available. But for more common applications, the user might not need as much information about the execution of their request. In this case, it is then totally possible to apply usual quality of service techniques available in a service that would execute the request by adding a relevant scope itself.

4.1 Proof of concept

Though Cheops is still a work in progress, we demonstrated the relevance of sharing a resource in a proof of concept (PoC) on OpenStack [2]. This PoC gives DevOps the ability to make multiple independant instances of OpenStack collaborative. Using our approach with OpenStack would allow to manage a geo-distributed infrastructure as a usual IaaS platform. This is a breakthrough as several initiatives tried to propose a framework to manage edge infrastructures and processes [5, 9], but due to the difficulty of delivering a software as complex/complete as OpenStack, the work to be redone would be colossal.

4.2 Limitations

There are of course some limitations to our approach. First, it requires microservices-based applications that exposes an API for services communications. These applications need to be able to work on a single site since we will deploy them autonomously on every sites. Moreover, every instance of the application should have the same version for identical resources.

This approach ensures consistency at the service-level, but for the resources they manage. The only operations available to manipulate these resources are therefore the ones exposed by the API. Thus, the resources are maintained as identical as the API allows it, but nothing less. For example, nothing can be said about the consistency of two VMs booted through this process; their internal state will probably diverge, as expected.

5 Conclusion

In this paper, we presented the replication mechanisms of Cheops and how scope-lang allows its parametrization. The ultimate goal of this project is to allow

[4] https://istio.io/

[5] https://openservicemesh.io/

generic collaborations between multiple instances of the same application without applying intrusive changes in the business code. We presented especially the different workflows for the replication collaboration.

As future work, we identified other collaboration mechanisms that could be relevant. For example, our replication strategy could be extended in order to include a controller and propose an abstraction similar to the ReplicaSet and its controller in the Kubernetes ecosystem[6]. The point would be to add control loop capabilities into Cheops in order to maintain the desired number of replicas according to the infrastructure changes. We could also propose different ways to keep the consistency between replicas, giving more choice for the users (e.g., giving them the choice to change the location of a replica if its site fails).

Besides replication, additional collaborations can also be envisioned (such as an *otherwise* operator that will ask for a request to be executed on a specific site and if this one is unavailable, execute on the other specified). Any future implementation will depend on the needs observed when deploying this solution.

Acknowledgment

We would like to thank Matthieu Juzdzewski and Arnaud Szymanek for their work on Cheops.

References

1. Akkoorath, D.D., et al.: Cure: Strong semantics meets high availability and low latency. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE (2016)
2. Cherrueau, R.A., Delavergne, M., Lebre, A., Rojas Balderrama, J., Simonin, M.: Edge Computing Resource Management System: Two Years Later! Research Report RR-9336, Inria Rennes Bretagne Atlantique (2020)
3. 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)
4. Martin, B., Prosperi, L., Shapiro, M.: An environment for composable distributed computing. In: EuroDW 2020-14th EuroSys Doctoral Workshop (2020)
5. Mortazavi, S.H., Salehe, M., Gomes, C.S., Phillips, C., de Lara, E.: Cloudpath: A multi-tier cloud computing framework. In: Proceedings of the Second ACM/IEEE Symposium on Edge Computing. pp. 1–13 (2017)
6. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14) (2014)
7. Safina, L., Mazzara, M., Montesi, F., Rivera, V.: Data-driven workflows for microservices: Genericity in jolie. In: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA). IEEE (2016)
8. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1) (2017)
9. Wang, N., et al.: Enorm: A framework for edge node resource management. IEEE transactions on services computing (2017)
10. Zhu, Y., Wang, Y.: Shaft: Supporting transactions with serializability and fault-tolerance in highly-available datastores. In: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS). pp. 717–724 (2015)

[6] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/