Geo-distribute Cloud Applications at the Edge

Ronan-Alexandre Cherrueau, Marie Delavergne(✉), and Adrien Lèbre

Inria, LS2N Laboratory, Nantes, France
marie.delavergne@inria.fr

Abstract. With the arrival of the edge computing a new challenge arises for cloud applications: How to benefit from geo-distribution (locality) while dealing with inherent constraints of wide-area network links? The admitted approach consists in modifying cloud applications by entangling geo-distribution aspects in the business logic using distributed data stores. However, this makes the code intricate and contradicts the software engineering principle of externalizing concerns. We propose a different approach that relies on the modularity property of microservices applications: (i) one instance of an application is deployed at each edge location, making the system more robust to network partitions (local requests can still be satisfied), and (ii) collaboration between instances can be programmed outside of the application in a generic manner thanks to a service mesh. We validate the relevance of our proposal on a real use-case: geo-distributing OpenStack, a modular application composed of 13 million of lines of code and more than 150 services.

Keywords: Edge computing · Resource sharing · DSL · Service composition · Service mesh · Modularity

1 Introduction

The deployment of multiple micro and nano Data Centers (DCs) at the edge of the network is taking off. Unfortunately, our community is lacking tools to make applications benefit from the geo-distribution while dealing with high latency and frequent disconnections inherent to wide-area networks (WAN) [12].

The current accepted research direction for developing geo-distributed applications consists in using globally distributed data stores [1]. Roughly, distributed data stores emulate a shared memory space among DCs to make the development of geo-distributed application easier [13]. This approach however implies to *entangle* the geo-distribution concern in the business logic of the application. This contradicts the software engineering principle of externalizing concerns. A principle widely adopted in the cloud computing where a strict separation between development and operational (abbreviated as *DevOps*) teams exists [6,8]: Programmers focus on the development and support of the business logic

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 301–316, 2021.
https://doi.org/10.1007/978-3-030-85665-6_19

of the application (i.e., the services), whereas DevOps are in charge of the execution of the application on the infrastructure (e.g., deployment, monitoring, scaling).

The lack of separation between the business logic and the geo-distribution concern is not the only problem when using distributed data stores. Data stores distribute resources across DCs in a pervasive manner. In most cases, resources are distributed across the infrastructure identically. However, all resources do not have the same scope in a geo-distributed context. Some are useful in one DC, whereas others need to be shared across multiple locations to control the latency, scalability and availability [3,4]. And scopes may change as time passes. It is therefore tedious for programmers to envision all scenarios in advance, and a fine-grained control per resource is mandatory.

Based on these two observations, we propose to deal with the geo-distribution as an independent concern using the service mesh concept widely adopted in the cloud. A service mesh is a layer over microservices that intercepts requests in order to decouple concerns such as monitoring or auto-scaling [8]. The code of the Netflix Zuul[1] load balancer for example is independent of the domain and generic to any modular application by only considering their requests. In this paper, we explore the same idea for the geo-distribution concern. By default, one instance of the cloud application is deployed on each DC, and a dedicated service mesh *forwards* requests between the different DCs. The forwarding operation is programmed using a domain specific language (DSL) that enables two kinds of collaboration. First, the access of resources available at another DC. Second, the replication of resources on a set of DCs. The DSL reifies the resource location. This makes it clear how far a resource is and where its replicas are. It gives a glimpse of requests probable latencies and a control on resources availability.

The contributions of this paper are as follows:

- We state what it means to geo-distribute an application and illustrate why using a distributed data store is problematic, discussing a real use-case: OpenStack[2] for the edge. OpenStack is the defacto application for managing cloud infrastructures (Sect. 2).
- We present our DSL to program the forwarding of requests and our service mesh that interprets expressions of our language to implement the geo-distribution of a cloud application. Our DSL lets clients specify for each request, in which DC a resource should be manipulated. Our service mesh relies on guarantees provided by modularity to be independent of the domain of the application and generic to any microservices application (Sect. 3).
- We present a proof of concept of our service mesh to geo-distribute OpenStack.[3] With its 13 million of lines of code and more than 150 services,

[1] https://netflixtechblog.com/open-sourcing-zuul-2-82ea476cb2b3. Accessed 2021-02-15. Zuul defines itself as an API gateway. In this paper, we do not make any difference with a service mesh. They both intercept and control requests on top of microservices.

[2] https://www.openstack.org/software. Accessed 2021-02-15.

[3] https://github.com/BeyondTheClouds/openstackoid/tree/stable/rocky. Accessed 2021-02-15.

OpenStack is a complex cloud application, making it the perfect candidate to validate the independent geo-distribution mechanism that we advocate for. Thanks to our proposal, DevOps can make multiple independent instances of OpenStack collaborative to use a geo-distributed infrastructure (Sect. 4).

We finally conclude and discuss about limitations and future work to push the collaboration between application instances further (Sect. 5).

2 Geo-distributing Applications

In this section, we state what it means to geo-distribute a cloud application and illustrate the invasive aspect of using a distributed data store when geo-distributing the OpenStack application.

2.1 Geo-distribution Principles

We consider an edge infrastructure composed of several geo-distributed micro DCs, up to thousands. Each DC is in charge of delivering cloud capabilities to an edge location (i.e., an airport, a large city, a region ...) and is composed of up to one hundred servers, nearly two racks. The expected latency between DCs can range from 10 to 300 ms round trip time according to the radius of the edge infrastructure (metropolitan, national ...) with throughput constraints (i.e., LAN vs WAN links). Finally, disconnections between DCs are the norm rather than the exception, which leads to network split-brain situations [10]. We underline we do not consider network constraints within a DC (i.e., an edge location) since the edge objective is to bring resources as close as possible to its end use.

Cloud applications of the edge have to reckon with these edge specifics in addition to the geo-distribution of the resources themselves [14]. We hence suggest that geo-distributing an application implies adhering to the following principles:

Local-first. A geo-distributed cloud application should minimize communications between DCs and be able to deal with network partitioning issues by continuing to serve local requests at least (i.e., requests delivered by users in the vicinity of the isolated DC).

Collaborative-then. A geo-distributed cloud application should be able to share resources across DCs on demand and replicate them to minimize user perceived latency and increase robustness when needed.

Unfortunately, implementing these principles produces a code that is hard to reason about and thus rarely addressed [2]. This is for instance the case of OpenStack that we discuss in the following section.

2.2 The Issue of Geo-distributing with a Distributed Data Store

OpenStack is a resource management application to operate one DC. It is responsible for booting Virtual Machines (VMs), assigning VMs in networks, storing operating system images, administrating users, or any operation related to the management of a DC.

A Complex but Modular Application. Similarly to other cloud applications such as Netflix or Uber, OpenStack follows a modular design with many services. The compute service for example manages the compute resources of a DC to boot VMs. The image service controls operating system BLOBs like Debian. Figure 1 depicts this modular design in the context of a boot of a VM. The DevOp starts by addressing a boot request to the compute service of the DC (Step 1). The compute service handles the request and contacts the image service to get the Debian BLOB in return (Step 2). Finally, the compute does a bunch of internal calls – schedules VM, setups the network, mounts the drive with the BLOB – before booting the new VM on one of its compute nodes (Step 3).[4]

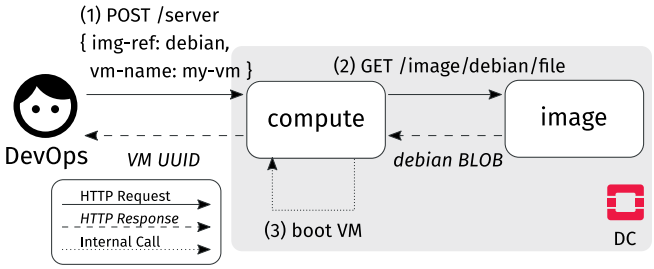


Fig. 1. Boot of a Debian VM in OpenStack

Geo-distributing Openstack. Following the local-first and collaborative-then principles implies two important considerations for OpenStack. First, each DC should behave like a usual cloud infrastructure where DevOps can make requests and use resources belonging to one site without any external communication to other sites. This minimizes the latency and satisfies the robustness criteria for local requests. Second, DevOps should be able to manipulate resources between DCs if needed [3]. For instance, Fig. 11 illustrates an hypothetical sharing with the "boot of a VM at one DC using the Debian image available in a second one".

[4] For clarity, this paper simplifies the boot workflow. In a real OpenStack, the boot also requires at least the network and identity service. Many other service may also be involved. See https://www.openstack.org/software/project-navigator/openstack-components. Accessed 2021-02-15.

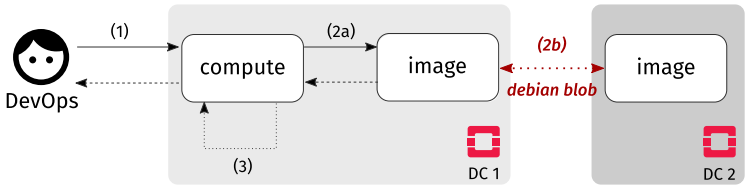


Fig. 2. Boot of a VM using a remote BLOB

Code 1.1. Retrieval of a BLOB in the image service

```
1 @app.get('/image/{name}/file')
2 def get_image(name: String) -> BLOB:
3   # Lookup the image path in the data store: `path = proto://path/debian.qcow`
4   path = ds.query(f'''SELECT path FROM images WHERE id IS "{name}";''')
5
6   # Read path to get the image BLOB
7   image_blob = image_collection.get(path)
8   return image_blob
```

To provide this resource sharing between DC 1 and DC 2, the image service has to implement an additional dedicated means (Step 2b). Moreover, it should be configurable as it might be relevant to replicate the resource if the sharing is supposed to be done multiple times over a WAN link. Implementing such a mechanism is a tedious task for programmers of the application, who prefer to rely on a distributed data store [4] (Fig. 2).

Distributed Data Store Tangles the Geo-distribution Concern. The OpenStack image service team currently studies several solutions to implement the "booting a VM at DC 1 that benefits from images in DC 2" scenario. All are based on a distributed data store that provides resource sharing between multiple image services: a pull mode where DC 1 instance gets BLOBs from DC 2 using a message passing middleware, a system that replicates BLOBs around instances using a shared database, etc.[5] The bottom line is that they all require to *tangle* the geo-distribution concern with the logic of the application. This can be illustrated by the code that retrieves a BLOB when a request is issued on the image service (code at Step 2 from Fig. 11).

Code 1.1 gives a coarse-grained description of that code. It first queries the data store to find the path of the BLOB (l. 3,4). It then retrieves that BLOB in the `image_collection` and returns it to the caller using the `get` method (l. 6–8). Particularly, this method resolves the protocol of the `/path/` and calls the proper library to get the image. Most of the time, that path refers to a BLOB on the local disk (e.g., `file:///path/debian.qcow`). In such a case, the

[5] https://wiki.openstack.org/wiki/Image_handling_in_edge_environment. Accessed 2021-02-15.

method `image_collection.get` relies on the local `open` python function to get the BLOB.

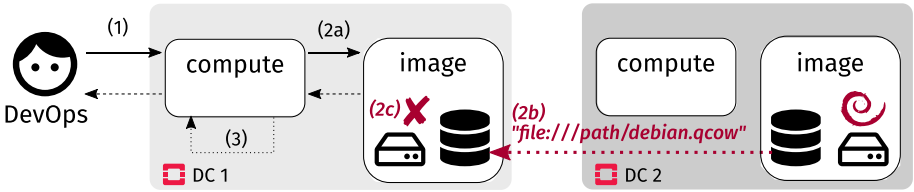


Fig. 3. Booting a VM at DC 1 with a BLOB in DC 2 using a distributed data store (does not work)

The code executes properly as long as only one OpenStack is involved. But things go wrong when multiple are unified through a data store. If Code 1.1 remains unchanged, then the sole difference in the workflow of "booting a VM at DC 1 using an image in DC 2" is the distributed data store that federates all image paths (including those in DC 2—see Fig. 3). Unfortunately, because DC 2 hosts the Debian image, the file path of that image returned at Step 2b is local to DC 2 and *does not exist* on the disk of DC 1. An *error* results in the `image_collection.get` (2c).

The execution of the method `image_collection.get` takes place in a specific environment called its *execution context*. This context contains explicit data such as the method parameters. In our case, the image `path` found from the data store. It also contains implicit assumptions made by the programmer: "A path with the `file:` prototype must refer to an image stored on the local disk". Alas, such kind of assumptions are wrong with a distributed data store. They have to be fixed. For this scenario of "booting a VM at DC 1 using an image in DC 2", it means changing the `image_collection.get` method in order to allow the access of the disk of DC 2 from DC 1. More generally, a distributed data store constrains programmers to take the distribution into account (and the struggle to achieved it) in the application. And besides this entanglement, a distributed data store also strongly limits the collaborative-then principle. In the code above, there is no way to specify whether a particular BLOB should be replicated or not.

Our position is that the geo-distribution must be handled outside of the logic and in a fine-grained manner due to its complexity.

3 Geo-distribute Applications with a Service Mesh

In this section, we first introduce the foundations and notations for our proposal. We then build upon this to present our service mesh that decouples the geo-distribution concern from the business logic of an application.

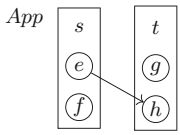
3.1 Microservices and Service Mesh Basics

An application that follows a microservices architecture combines several services [7]. Each service defines endpoints (operations) for managing one or various resources [5]. The combination of several services endpoints forms a series of calls called a workflow. Deploying all services of an application constitutes one application instance (often shortened instance in the rest). The services running in an application instance are called service instances. Each application instance achieves the application intent by exposing all of its workflows which enables the manipulation of resource values.

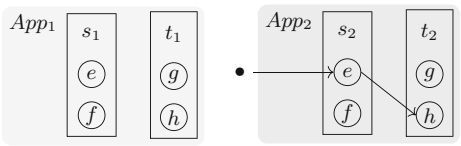
Figure 4 illustrates such an architecture. Figure 4a depicts an application *App* made of two services s, t that expose endpoints e, f, g, h and one example of a workflow $s.e \rightarrow t.h$. *App* could be for example the OpenStack application. In this context, service s is the compute service that manages VMs. Its endpoint e creates VMs and f lists them. Service t is the image service that controls operating system BLOBs. Its endpoint g stores an image and h downloads one. The composition $s.e \rightarrow t.h$ models the boot workflow (as seen in Fig. 1). Figure 4b shows two application instances of *App* and their corresponding service instances: s_1 and t_1 for App_1; s_2 and t_2 for App_2. A client ($\bullet$) 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.

Microservices architectures are the keystone of DevOps practices. They promote a modular decomposition of services for their independent instantiation and maintenance. A service mesh takes benefit of this situation to implement communication related features such as authentication, message routing or load balancing outside of the application. In its most basic form, a service mesh consists in a set of reverse proxies around service instances that encapsulate a specific code to control communications between services [8]. This encapsulation in each proxy *decouples* the code managed by DevOps from the application business logic maintained by programmers. It also makes the service mesh *generic* to all services by considering them as black boxes and only taking into account their communication requests.

Figure 5 illustrates the service mesh approach with the monitoring of requests to have insight about the application. It shows reverse proxies mon_s and mon_t



(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 $\bullet$ represents a client that executes the $s.e \rightarrow t.h$ workflow in App_2.

Fig. 4. Microservices architecture of a cloud application

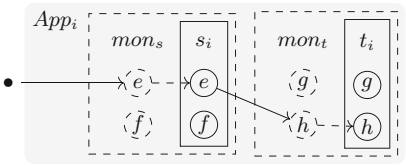


Fig. 5. Service mesh *mon* for the monitoring of requests

that collect metrics on requests toward service instances s_i and t_i during the execution of the workflow $s.e \rightarrow t.h$ on App_i. The encapsulated code in mon_s and mon_t collects for example requests latency and success/error rates. It may send metrics to a time series database as InfluxDB and could be changed by anything else without touching the *App* code.

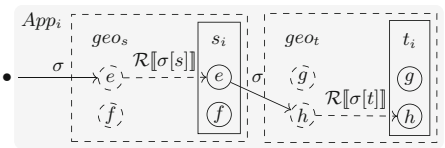
3.2 A Tour of Scope-lang and the Service Mesh for Geo-distributing

Interestingly, running one instance of a microservices application *automatically* honors the local-first principle. In Fig. 4b, the two instances are independent. They can be deployed on two different DCs. This obviously cancels communications between DCs as well as the impact of DCs' disconnections. However for the global system, it results in plenty of concurrent values of the same resource distributed but isolated among all instances (e.g., $s1$ and $s2$ manage the same kind of resources but their values differ as time passes). Manipulating any concurrent value of any instance requires now on to code the collaboration in the service mesh and let clients to specify it at will.

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

$$
\begin{array}{l}
\mathcal{R}[\![s : App_i]\!] = s_i \\
\mathcal{R}[\![s : Loc\&Loc']\!] = \mathcal{R}[\![s : Loc]\!] \text{ and } \mathcal{R}[\![s : Loc']\!]
\end{array}
$$

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



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

(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. 6. A service mesh to geo-distribute a cloud application

In that regard, we developed a domain specific language called scope-lang. A scope-lang expression (referred to as the *scope* or σ in Fig. 6a) 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 tells to use the service s from App_1 and t from App_2. The scope "$t : App_1 \& App_2$" specifies to use the service t from App_1 and App_2.

Clients set the scope of a request to specify the collaboration between instances they want for a specific execution. The scope is then *interpreted* by our service mesh during the execution of the workflow to fulfill that collaboration. The main operation it performs is *request forwarding*. Broadly speaking, reverse proxies in front of service instances (geo_s and geo_t in Fig. 6b) intercept the request and interpret its scope to forward the request somewhere. "Where" exactly depends on locations in the scope. However, the interpretation of the scope always occurs in the following stages:

1. A request is addressed to the endpoint of a service of one application instance. The request piggybacks a scope, typically as an HTTP header in a RESTful application. For example in Fig. 6b: $\bullet \xrightarrow{s:App_i, t:App_i} s.e$.
2. The reverse proxy in front of the service instance intercepts the request and reads the scope. In Fig. 6b: geo_s intercepts the request and reads σ which is equal to $s : App_i, t : App_i$.
3. The reverse proxy extracts the location assigned to its service from the scope. In Fig. 6b: geo_s extracts the location assigned to s from σ. This operation, notated $\sigma[s]$, returns App_i.
4. The reverse proxy uses a specific function $\mathcal{R}$ (see Fig. 6a) 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* [8] and therefore is not presented here. In Fig. 6b: $\mathcal{R}[\![s : \sigma[s]]\!]$ reduces to $\mathcal{R}[\![s : App_i]\!]$ and is resolved to service instance s_i.
5. The reverse proxy *forwards* the request to the endpoint of the resolved service instance. In Fig. 6b: geo_s forwards the request to $s_i.e$.

In this example of executing the workflow $s.e \xrightarrow{\sigma} t.h$, the endpoint $s_i.e$ has in turn to contact the endpoint h of service t. The reverse proxy geo_s propagates the scope on the outgoing request towards the service t. The request then goes through stages 2 to 5 on behalf of the reverse proxy geo_t. It results in a forwarding to the endpoint $\mathcal{R}[\![t : \sigma[t]]\!].h$ that is resolved to $t_i.h$.

Here, the scope only refers to one location (i.e., App_i). Thus the execution of the workflow remains *local* to that location. The next sections detail the use of forwarding in order to perform collaboration between instances.

3.3 Forwarding for Resource Sharing

A modular decomposition of the code is popular for programmers of microservices architecture. It divides the functionality of the application into *independent and interchangeable* services [9]. This brings well-known benefits including ease of reasoning. More importantly, modularity also gives the ability to *change* a service with a certain API by any service exposing the same API and logic [11].

A load balancer, such a Netflix Zuul mentioned in the introduction, makes a good use of this property to distribute the load between multiple instances of the

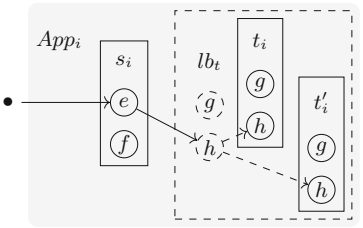


Fig. 7. Load balancing principle

same modular service [5]. Figure 7 shows this process with the reverse proxy for load balancing lb_t of the service t. lb_t intercepts and balances incoming requests within two service instances t_i and t'_i during the execution of the workflow $s.e \rightarrow t.h$ in App_i. From one execution to another, the endpoint $s_i.e$ gets result from $t_i.h$ or $t'_i.h$ in a safe and transparent manner thanks to modularity.

We generalize this mechanism to share resources. In contrast to the load balancer that changes the composition between multiple instances of the same service *inside* a single application instance. Here, we change the composition between multiple instances of the same service *across* application instances. As a consequence, the different service instances share their resources during the execution of a workflow.

Figure 8 depicts this cross dynamic composition mechanism during the execution of the workflow $s.e \xrightarrow{s:App_1,t:App_2} t.h$. The service instance s_1 of App_1 is dynamically composed thanks to the forwarding operation of the service mesh with the service instance t_2 of App_2. This forwarding is safe relying on the guaranty provided by modularity. (If t is modular, then we can swap t_1 by t_2 since they obviously have the same API and logic.) As a result, the endpoint $s_1.e$ benefits from resource values of $t_2.h$ instead of its usual $t_1.h$.

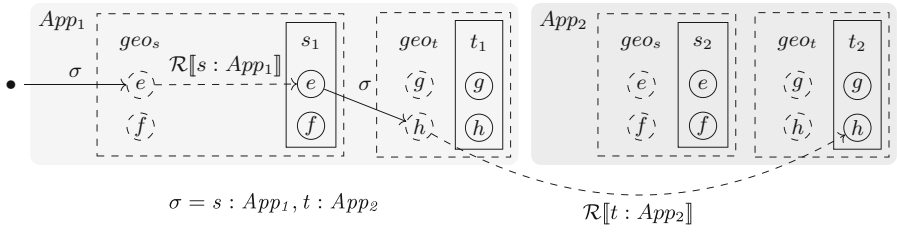


Fig. 8. Resource sharing by forwarding across instances

3.4 Forwarding for Resource Replication

Replication is the ability to create and maintain identical resources on different DCs: an operation on one replica should be propagated to the other ones

according to a certain consistency policy. In our context, it is used to deal with latency and availability.

In terms of implementation, microservices often follow a RESTful HTTP API and so generate an identifier for each resource. This identifier is later used to retrieve, update or delete resources. Since each application instance is independent, our service mesh requires a meta-identifier to manipulate replicas across the different DCs as a unique resource.

For example, the image service t exposes an endpoint g to create an image. When using a scope for replication, such as $t : App_1 \& App_2$, the service mesh generates a meta-identifier and maps it $\{metaId : [App_1 : localID_{t_1}, App_2 : localID_{t_2}]\}$. In Fig. 9, if t_1 creates a replica with the identifier 42 and t_2 6, and our meta-identifier was generated as 72, the mapping is: $\{72 : [App_1 : 42, App_2 : 6]\}$. Mappings are stored in an independent database alongside each application instance.

The replication process is as follows:

1. A request for replication is addressed to the endpoint of a service of one application instance. For example in Fig. 9: $\bullet \xrightarrow{t:App_1 \& App_2} t.g$.
2. Similarly to the sharing, the $\mathcal{R}$ function is used to resolve the endpoints that will store replicas. $\mathcal{R}[\![s : Loc \& Loc']\!] = \mathcal{R}[\![s : Loc]\!]$ and $\mathcal{R}[\![s : Loc']\!]$. In Fig. 9: $\mathcal{R}[\![t : App_1 \& App_2]\!]$ is equivalent to $\mathcal{R}[\![t : App_1]\!]$ and $\mathcal{R}[\![t : App_2]\!]$. Consequently, t_1 and t_2.
3. The meta-identifier is generated along with the mapping and added in the database. In Fig. 9: $\{\ 72 : [App_1 : none, App_2 : none]\}$.
4. Each request is forwarded to the corresponding endpoints on involved DCs and a copy of the mapping is stored in those DCs' database simultaneously. In Fig. 9: geo_t forwards the request to $t_1.g$ and $t_2.g$ and stores the mapping $\{72 : [App_1 : none, App_2 : none]\}$ in App_1 and App_2 databases.
5. Each contacted service instance executes the request and returns the results (including the local identifier) to the service mesh. In Fig. 9: t_1 and t_2 returns respectively the local identifier 42 and 6.
6. The service mesh completes the mapping and populates the involved DCs' databases. In Fig. 9: the mapping now is $\{72 : [App_1 : 6, App_2 : 42]\}$ and added to databases of App_1 and App_2.

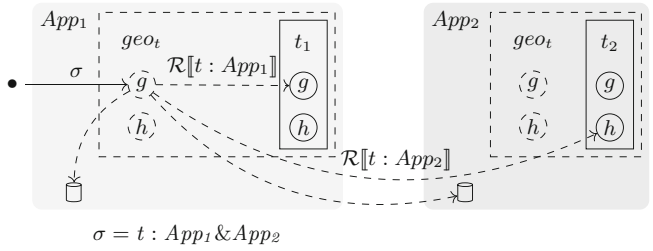


Fig. 9. Replication by forwarding on multiple instances

7. By default, the meta identifier is returned as the final response. If a response other than an identifier is expected, the first received response is transferred (since others are replicas with similar values).

This process ensures that only interactions with the involved DCs occur, avoiding wasteful communications. Each operation that would later modify or delete one of the replicas will be applied to every others using the mapping available on each site. To prevent any direct manipulation, local identifiers of replicas are hidden.

This replication control perfectly suits our collaborative-then principle. It allows a client to choose "when" and "where" to replicate. Regarding the "how", our current process for forwarding replica requests, maintaining mappings and ensuring that operations done on one replica are applied on others, is naive. Implementing advanced strategies is left as future work. However, we underline that it does not change the foundations of our proposal. Ultimately, choosing the strategy should be made possible at the scope-lang level (e.g., weak, eventual or strong consistency).

3.5 Towards a Generalized Control System

The code of scope-lang is independent of cloud applications and can easily be extended with new features in it. Scope-lang is thus a great place to implement additional operators that would give more control during the manipulation of resources although it has been initially designed for resources sharing and replication. Choosing between different levels of consistency, as highlighted in the previous section, is one example of new operators. Here, we give two other ones to stress the generality of our approach.

The new *otherwise* operator ("$Loc_1; Loc_2$" in Fig. 10a) informally tells to use the first location or fallback on the second one if there is a problem. This operator comes in handy when a client wants to deal with DCs' disconnections. Adding it to the service mesh implies to implement in the reverse proxy what to do when it interprets a scope with a (;). The implementation is straightforward: Make the reverse proxy forward the request to the first location and proceed if it succeeds, or forward the request to the second location otherwise.

Ultimately, we can built new operators upon existing ones. This is the case of the `around` function that considers all locations reachable in a certain amount of time, e.g., `around(`App_1`, 10ms)`. To achieve this, the function combines the available locations with the otherwise operator (;), as shown in Fig. 10b. Thus it does not require to change the code of the interpreter in the service mesh.

$$Loc ::= \ldots \quad \text{see Fig. 6a}$$
$$\mid Loc; Loc \quad \text{otherwise location}$$

$$\mathcal{R}[\![s : Loc_1; Loc_2]\!] = \mathcal{R}[\![s : Loc_1]\!] \text{ otherwise } \mathcal{R}[\![s : Loc_2]\!]$$

(a) The otherwise (;) operator. It requires to update the code of the interpreter in the service mesh.

```
def around(loc: Loc, radius: timedelta) -> Loc:
  # Find all Locs in the `radius` of `loc`
  # >  locs = [App1, App2, ..., Appn]
  locs = _find_locs(loc, radius)

  # Combine all `locs` with `;`
  # >  App1;App2;...;Appn
  return foldr(;, locs, loc)
```

(b) The `around` operator build upon (;). It does not need to update the code of the interpreter in the service mesh.

Fig. 10. New operators for scope-lang

4 Validation on OpenStack

We demonstrate the feasibility of our approach with a prototype for OpenStack.[6] In this proof of concept, we set up an HAProxy[7] in front of OpenStack services. HAProxy is a reverse proxy that intercepts HTTP requests and forwards them to specific backends. In particular, HAProxy enables to dynamically choose a backend using a dedicated Lua code that reads information from HTTP headers. In our proof of concept, we have developed a specific Lua code that extracts the scope from HTTP headers and interprets it as described in Sect. 3.2 to 3.4.

In a normal OpenStack, the booting of a VM with a Debian image (as presented in Fig. 1) is done by issuing the following command:

```
$ openstack server create my-vm --image debian          (Cmd. 1)
```

We have extended the OpenStack command-line client with an extra argument called `--scope`. This argument takes the scope and adds it as a specific header on the HTTP request. Thus it can latter be interpreted by our HAProxy. With it, a DevOps can execute the previous Cmd. 1 in a specific location by adding a `--scope` that, for instance, specifies to use the `compute` and `image` service instance of DC 1:

```
$ openstack server create my-vm --image debian\
            --scope { compute: DC 1, image: DC 1 }      (Cmd. 2)
```

In the case where the DevOps is in DC 1, the request is entirely local and thus will be satisfied even during network disconnections between DCs. Actually, all locally scoped requests are always satisfied because each DC executes one particular instance of OpenStack. This ensures the local-first principle.

The next command (Cmd. 3) mixes locations to do a resource sharing as explained in Sect. 3.3. It should be read as "boot a VM with a Debian image using the compute service instance of DC 1 and the image service instance of DC 2":

[6] https://github.com/BeyondTheClouds/openstackoid/tree/stable/rocky. Accessed 2021-02-15.

[7] https://www.haproxy.org/. Accessed 2021-02-15.

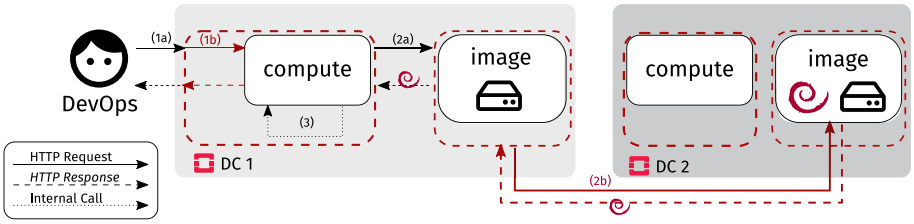


Fig. 11. Boot of a VM at DC 1 with a BLOB in DC 2 using scope-lang

```
$ openstack server create my-vm --image debian\
            --scope { compute: DC 1, image: DC 2 }      (Cmd. 3)
```

Figure 11 shows the execution. Dotted red squares represent HAProxy instances. Step 1b and 2b correspond to the forwarding of the request according to the scope. The execution results in the sharing of the Debian image at DC 2 to boot a VM at DC 1. That collaboration is taken over by the service mesh. No change to the OpenStack code has been required.

Mixing locations in the scope makes it explicit to the DevOps that the request may be impacted by the latency. If DC 1 and DC 2 are far for from each other, then it should be clear to the DevOps that Cmd. 3 is going to last a certain amount of time. To mitigate this, the DevOps may choose to replicate the image:

```
$ openstack image create debian --file ./debian.qcow2\
            --scope { image: DC 1 & DC 2 }              (Cmd. 4)
```

This command (Cmd. 4) creates an identical image on both DC 1 and DC 2 using the protocol seen in Sect. 3.4. It reduces the cost of fetching the image each time it is needed (as in Cmd. 3). Moreover, in case of partitioning, it is still possible to create a VM from this image on each site where the replicas are located. This fine-grained control ensures to replicate (and pay the cost of replication) only when and where it is needed to provide a collaborative-then application.

Our proof of concept has been presented twice at the OpenStack Summit and is mentioned as an interesting approach to geo-distribute OpenStack in the second white paper published by the Edge Computing Working Group of the OpenStack foundation in 2020.[8]

5 Conclusion

We propose a new approach to geo-distribute microservices applications without meddling in their code. By default, one instance of the application is deployed on each edge location and collaborations between the different instances are achieved through a generic service mesh. A DSL, called scope-lang, allows

[8] https://www.openstack.org/use-cases/edge-computing/edge-computing-next-steps-in-architecture-design-and-testing Accessed 2021-02-18.

the configuration of the service mesh on demand and on a per request basis, enabling the manipulation of resources at different levels: locally to one instance (by default) and across distinct instances (sharing) and multiple (replication). Expliciting the location of services in each request makes the user aware of the number of sites that are involved (to control the scalability), the distance to these sites (to control the network latency/disconnections), and the number of replicas to maintain (to control the availability/network partitions).

We demonstrated the relevance of our approach with a proof of concept that enables to geo-distribute OpenStack, the defacto cloud manager. Thanks to it, DevOps can make multiple independent instances of OpenStack collaborative. This proof of concept is among the first concrete solutions to manage a geo-distributed edge infrastructure as an usual IaaS platform. Interestingly, by using our proposal over our proof of concept, DevOps can now envision to geo-distribute cloud applications.

This separation between the business logic and the geo-distribution concern is a major change with respect to the state of the art. It is important however to underline that our proposal is built on the modularity property of cloud applications. In other words, an application that does not respect this property cannot benefit from our service mesh. Regarding our future work, we have already identified additional collaboration mechanisms that can be relevant. For instance, we are investigating how a resource that has been created on one instance can be extended or even reassigned to another one.

We believe that a generic and non invasive approach for geo-distributing cloud applications, such as the one we propose, is an interesting direction that should be investigated by our community.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. Computer **45**(2), 37–42 (2012)
2. Alvaro, P., et al.: Consistency analysis in bloom: a CALM and collected approach. In: 5th Biennial Conference on Innovative Data Systems Research, CIDR 2011, Online Proceedings, Asilomar, CA, USA, 9–12 January 2011, pp. 249–260 (2011)
3. Cherrueau, R., et al.: Edge computing resource management system: a critical building block! initiating the debate via openstack. In: USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2018, Boston, MA, 10 July. USENIX Association (2018)
4. Corbett, J.C., et al.: Spanner: Google's globally-distributed database. In: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, 8–10 October 2012, pp. 261–264 (2012)
5. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
6. Herbst, N.R., et al.: Elasticity in cloud computing: what it is, and what it is not. In: 10th International Conference on Autonomic Computing, ICAC 2013, San Jose, CA, June 2013, pp. 23–27. USENIX Association (2013)
7. Jamshidi, P., et al.: Microservices: the journey so far and challenges ahead. IEEE Softw. **35**(3), 24–35 (2018)

8. 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)
9. Liskov, B.: A design methodology for reliable software systems. In: Proceedings of the AFIPS 1972 Fall Joint Computer Conference, USA, 5–7 December, pp. 191–199. American Federation of Information Processing Societies (1972)
10. Markopoulou, A., et al.: Characterization of failures in an operational IP backbone network. IEEE/ACM Trans. Netw. **16**(4), 749–762 (2008)
11. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
12. Satyanarayanan, M.: The emergence of edge computing. Computer **50**(1), 30–39 (2017)
13. Shapiro, M., et al.: Just-right consistency: reconciling availability and safety. CoRR abs/1801.06340 (2018). http://arxiv.org/abs/1801.06340
14. Tato, G., 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, 17–19 December, Neuchâtel, Switzerland. LIPIcs, vol. 153, pp. 9:1–9:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)