Thinking out of replication for geo-distributing applications: the sharding case

Geo Johns Antony, Marie Delavergne, Adrien Lebre, Matthieu Rakotojaona Rainimangavelo

▶ To cite this version:

Geo Johns Antony, Marie Delavergne, Adrien Lebre, Matthieu Rakotojaona Rainimangavelo. Thinking out of replication for geo-distributing applications: the sharding case. ICFEC 2024 - 8th IEEE International Conference on Fog and Edge Computing, May 2024, Philadelphia, United States. pp.1-8. hal-04522961

HAL Id: hal-04522961
https://inria.hal.science/hal-04522961

Submitted on 27 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Thinking out of replication for geo-distributing applications: the sharding case

Geo Johns Antony
STACK research team
Inria
Nantes, France
0000-0002-9129-8281

Marie Delavergne
STACK research team
IMT Atlantique
Nantes, France
0000-0001-8020-959X

Adrien Lebre
STACK research team
Inria
Nantes, France
0000-0002-0305-4130

Matthieu Rakotojaona Rainimangavelo
STACK research team
IMT Atlantique
Nantes, France
<firstname.last-name>@imt-atlantique.fr

***Abstract*—To promote the adoption of the edge paradigm, our community needs innovative approaches for geo-distributing cloud applications across multiple locations without modifying existing business logic. While recent efforts propose using external services to orchestrate REST operations and achieve geo-distribution, relying solely on resource sharing and replication has limitations in finely distributing manipulated resources. This paper introduces a novel collaboration method that extends resources across multiple instances, going beyond simple replication. Our approach employs a shard-like strategy, enabling the creation of a distributed resource with a unified state view while mitigating a synchronization overhead. The effectiveness of our mechanism is demonstrated through a proof-of-concept implemented on top of the Kubernetes ecosystem.**

***Index Terms*—Sharding, Replication, Geo-distributed system, Edge application, Cloud application, Micro-services**

I. Introduction

Edge computing enables the processing of data closer to the source, reducing latency that is critical in future applications envisioned for augmented reality, industry 4.0, etc.

To favor the adoption of this new paradigm, our community is looking for new abstractions to shift applications from the cloud to the edge as easily as possible [1]. However, the literature have mainly proposed ad-hoc approaches where existing services composing an application are either revised in an intrusive manner or replaced by new ones.

In 2021, we introduced a preliminary study to present a new approach to geo-distribute microservice based applications without meddling in their code [2]. Considering that a cloud application consists of multiple services in charge of managing resources through CRUD/REST APIs, we proposed to execute one instance of each service on every location of the edge infrastructure and rely on a generic service to allow collaborations between the different instances each time it is required. A DSL (Domain Specific Language), called scope-lang, allows the configuration of the service on demand and on a per request basis, enabling the manipulation of resources at different levels: locally to one instance (by default), across distinct (sharing) and multiple (replication) instances.

While these collaborations enable geo-distribution of a cloud application, allowing to create the appearance of a unified resource across multiple instances when needed, they encounter significant challenges related to synchronization overhead. Indeed, using replication collaborations as proposed, results in a full copy on the state of resources even by using advanced CRDT approach [3]. Geo-distributing a Sharelatex project that is composed of multiple files using the replication collaboration suggested by the authors [1] will results in having all files replicated across each involved instance, leading to significant overheads and possibly non relevant for files that are purely local to one site. Even worse, for some cases, not being able to specify how a resource can be spread over multiple sites is an important limitation: Replicating a Kubernetes deployment resource results in pods running on every site, potentially deviating from the anticipated behavior where DevOps must specify the location of each pod composing the deployment, a topic we will delve into later in this article.

In light of these challenges, we propose in this paper to explore an alternative collaboration that rely on sharding techniques. Sharding involves dividing the resource into smaller, more manageable pieces called shards, and distributing these shards across different sites. The first challenge is to be able to shard a resource without making intrusive changes at the code level with the opportunity to mitigate the overhead caused by replication. In our case, a shard is not replicated on every site, hence another challenge will be, to make an independent shard that can serve some requests locally without relying on any external communication to the other sites. The approach we propose in this article differs from the sharding method generally used in database. Instead of assigning a subset of shards to one particular site based on range, directory or key distribution [4], we propose to reify the sharding strategy at the creation of each resource and leverage aforementioned *scope-lang* DSL [2] to define the sharding distribution.

However, another challenge we face is dealing with operations during network partitions. We ensure that a fragment of a resource is present at each site to satisfy the local requests and for remote operations we guarantee an eventual propagation of operations.

The contributions of this paper are:

- A new collaboration, *Cross*, that enables to extend a local resource into a geo-distributed one, irrespective of the underlying application.
- An extension routine that defines how a resource is geo-distributed in a non-intrusive manner to define shards.

- An aggregation routine that collects shards from geo-distributed sites in order to give the illusion of a complete resource available locally at each involved site.
- A generic routine to process shardable states within a resource at each site.
- A solution to handle network disconnections during an operation for a geo-distributed resource.
- A validation of our proposal through experiments that aim to geo-distribute the Kubernetes orchestrator.

The remaining of the paper is as follows: Section II presents the background on which our proposal relies. Section III introduces the *Cross* model and takes a deep dive into aggregation and extension operations. Section IV discusses the validation of the approach and Section V the related works. We conclude our paper by presenting the conclusion, in Section VI.

II. Background

The edge infrastructure we consider can be seen as a set of small data centers spread over multiple locations. Connectivity between these sites relies on WAN links with round trip times ranging from a few to hundreds of milliseconds together with probable disconnections.

Cloud applications pushed to the Edge should thus be able to cope with these specifics and include means to manage the geo-distribution of their resources across the infrastructure. In that regard, we suggested in [2] that geo-distributed applications should follow two principles: **local-first** to minimize communications between sites and to serve requests locally at least in case of network partitioning issues, and **collaborative-then** to leverage the resources across the infrastructure when needed. In order to do that, we introduced a twofold approach: a DSL called *scope-lang* to allow DevOps to choose where and how their resources will be located, and *Cheops*, a service that enables the external and generic management of the collaborations between different instances of the same application.

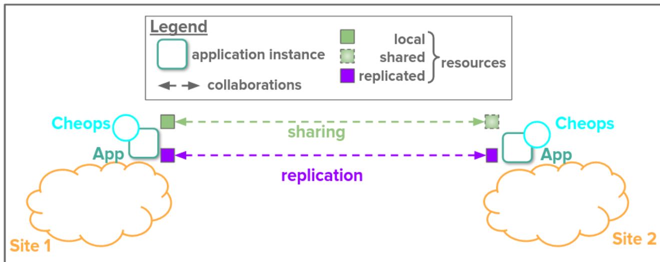

Figure 1. Collaborations offered by scope-lang and Cheops, along with Cross

Two collaborations were introduced, as depicted in Figure 1: *sharing*, which explicitly allows to forward a CRUD request to a remote service (and so interact with a remote resource) and *replication*, which replicates CRUD operations between multiple instances of the applications, allowing to manage multiple replicas of a resource. Sharing is useful to leverage the entire infrastructure without having to put all the resources on every site, and replication conveniently reduces the latency as well as increasing robustness in case of network partition by allowing to use the closest resources.

To demonstrate the relevance of both collaborations, we geo-distributed OpenStack[1], putting every services (the entire application) on each location. In sharing, the resources do not need themselves to be everywhere; if Nova (the compute service) needs a specific image from Glance (the image service) located only in Chicago to boot a VM in New York, the user can request to use sharing to get the image from Chicago, thus making a collaboration between two services that usually collaborate only when they are on the same site. Another example can be with the widely used Sharelatex application, where replication [5] can be used to push projects closer to users. If users are working in New York and Chicago, they might want to replicate their projects on two locations to use the closest in case of network partition, allowing them to continue working on those projects. They only need to request for their projects to be replicated on two services, and the two instances will manage them as they were only one.

These collaborations are instantiated on demand and on a per request basis using the DSL. Scope-lang reifies the location aspects thanks to a grammar that defines the execution location of requests. For more details about the scope-lang grammar, we invite the reader to refer to our previous paper [2].

Finally, Cheops [6] enacts these collaborations outside of the application by running as an external service co-located with an application instance which propagates CRUD operations to the appropriate sites, like a simple proxy.

Under the hood, Cheops consists of two major components: *Core* and *Glue*. ***Cheops Core*** consists of the generic collaboration management, considering resources as a black box and forwarding to the appropriate sites. ***Cheops Glue*** consists of all the information required to translate the request from the Core to the application, making it dependent on the application.

In summary, users specify in their requests through scope-lang, the collaborations to be implemented by Cheops. Regarding the two existing collaborations, sharing allows the separation of resources on different sites to avoid redundancy, and replication allows this redundancy to be defined only when needed by the users for specific resources.

III. Extending a geo-distributed resource

Cross consists of an extension process for a resource, referring to its ability to be distributed over multiple sites, allowing CRUD operations to be performed as local ones wherever the resource is effectively located. This section presents the purpose of Cross and our proposal on how to implement it.

A. General idea

Figure 2 (a) depicts the general idea of an extension of a resource R available at *Site 1, Site 2, ..., Site n*. Figure 2 (b) depicts our Cross concept based on a sharding strategy. The resource R is divided into chunks (referred as R', R'' and R''' in the figure) spread over the different sites.

[1] https://www.openstack.org/

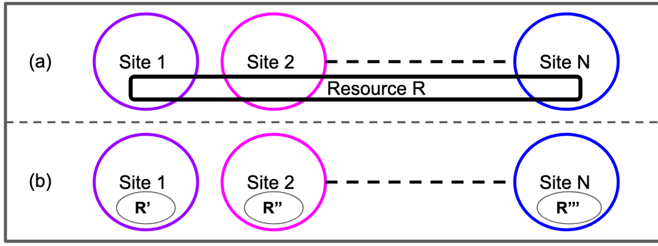

Figure 2. (a) Extending a resource (b) Replication pattern (c) Sharded pattern

As a resource is defined by attributes/elements, sharding a resource consists of assigning these elements to a specific site. However, as we want to give the illusion of having a single resource across multiple sites, the resource R should exist somehow on each site. The use of a proxy may help us deliver such an illusion [7], but it implies establishing a reliable communication to the site that actually contains the resource. The approach we propose consists of creating shards, that are resources of the same kind on each requested site and managing them as a single resource.

For instance, sharding a set $R = \{1, 2, 3, 4\}$ can be seen as splitting it into two sets $R' = \{1, 2\}$ and $R'' = \{3, 4\}$ instantiated respectively on $Site$ 1 and $Site$ 2, ensuring that the logical view can be reconstructed when needed. The elements inside the set i.e., $\{1, 2, 3, 4\}$ are split into fragments and deployed as independent resources on multiple sites. These are called *shardable elements*. On the flip side, the set R is rather straightforward as it does not contain any *non-shardable elements* defining it (such as single integer, a name, etc): if a name attribute (e.g., foo) is added to the set R, this name should be available on each site that owns a part of the resource. This type of elements requires replication across multiple sites and propagation of local changes to involved sites if necessary. For example, while creating independent fragments of the resource called as sub-resources R', R'', etc., we replicate the foo name and shard the set elements as mentioned earlier. We believe this is an acceptable trade-off in comparison to a full replica approach.

We generalize this concept as follows: sharding a resource R results in creating sub-resources in order to have R', R'', etc. non-shardable elements are replicated over all sub-resources, while shardable elements are spread according to the scope-lang specifications. Finally, every operations related to the non-shardable elements should be propagated to every sub-resources whereas others (i.e., operations on shardable elements) can be executed only on the impacted sites.

The challenge of sharding a resource R, while only relying on the API of the application, consists in answering the two following questions:

1) How do we shard a single resource into multiple sub-resources and identify which of these are shardable and non-shardable without changing the original code?
2) How can a logical view be guaranteed and reconstructed each time needed?

We propose an answer through two principles:

1) ***Extension*** allows the sharding of a resource into independent smaller ones, each existing on one site.
2) ***Aggregation*** maintains the illusion of a single resource across multiple sites. It hides the distribution from the application and gathers independent resources into an aggregated one when needed.

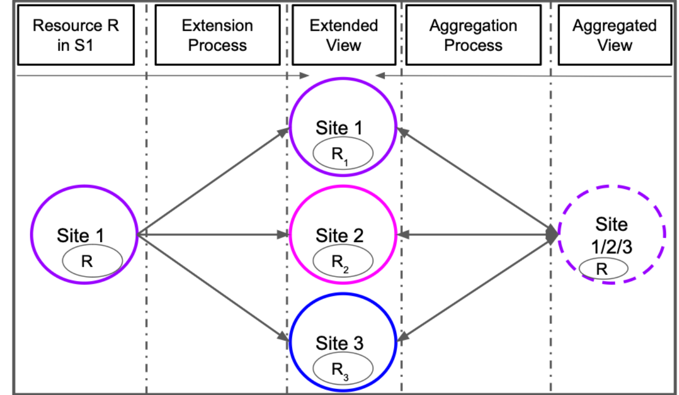

Figure 3. Different processes in the Cross model

Figure 3 illustrates the two principles over a resource R. Initially, R exists only as a single resource on $Site$ 1. Later, devops requests to extend R over $Sites$ 1, 2, 3. The extension process will lead to the creation of several independent resources R_1, R_2 and R_3 that will be gathered each time it is needed to give the illusion of interacting with the Cross resource R.

B. Extending/Mapping a Cross Resource

The information to specify the collaboration is given at the creation of a resource and stored in a data structure called Cross database (DB) model. $local_identifier$ identifies the local elements or sub-resources based on user definition for a site, which are then mapped to its corresponding site: $site \rightarrow local_identifiers$. Cross DB model maps the resource with these elements or sub-resources instantiated locally on each site through a global identifier that is called a *meta identifier*. These identifiers are then mapped themselves onto the global reference as $meta_identifier : [site_n : local_identifier_n, ...]$. The information about the $local_identifier$ is exclusive to sites where the local elements are deployed and $meta_identifier$ is stored at all involved sites as a copy. Such a mapping approach allows us to offer the cross extension externally without being intrusive to the native application.

However, if such a mapping looks appropriate, it is not sufficient to capture all sharding possibilities. Consider an ordered list $R = [a, b, c, d]$ that has been split into $R_1 = [a, b]$ on *Site 1* and $R_2 = [c, d]$ on *Site 2*. The current model does not allow the addition of an element e at the tail of the list while sharding it on $Site$ 1 (in this case, a simple concatenation of $R_1 + R_2$ does not allow to obtain R).

Consequently, the $meta_identifier$ can include additional information to be able to gather sub-resources to build an aggregated view. This information corresponds to values linked

to a logic defined once and for all through a configuration file for each kind of resources manipulated by any application to be geo-distributed. It is called the *resource_logic*. It consists of the extension and aggregation functions for a resource In other words, this logic defines how a resource can be sharded and what is the process (in form of code) to execute at each site, whereas the information saved in the Cross DB model corresponds to the effective value required for this logic. The effective value is stored as the *local_logic* at each site. The final Cross DB model corresponds to a mapping between the $meta_identifier$ to a $local_identifier$ associated with a $local_logic$ each time. Hence, the model will become $meta_identifier(resource_logic)$: $[site : local_identifier(local_logic), ...]$.

Code 1 shows a $Resource_logic$ for a list resource. We solve the problem mentioned above with the list by defining an extension and aggregation function. Extension function describes that the list elements are to be sharded and if a new operation happens, this logic is followed to keep the behaviour of the resource as expected for any operation. The aggregation function will give the illusion that the list is still maintained as a single resource for any operations. It follows a class-object like abstraction, where the class is the $resource_logic$ itself that consists of extension and aggregation functions. $local_logic$ is the object that instantiates the class with values associated with each user request and calls the functions for creating a resource.

The sharding process relies on a site based distribution provided explicitly through scope-lang. We extend the DSL to allow users to collaborate explicitly, by adding the specifications for Cross (% as the identifier). In a request, we would then specify in the scope part a request like that: $\{\{Loc\%Loc'\}; local_logic = \{Loc : \{shardLoc\}; Loc' : \{shardLoc'\}\}\}$ where Loc is the local site where the operation is performed by the user and Loc' is the new site where the resource is extended from Loc. The $local_logic$ specifies exactly where the shards will be located, by associating parts of the resource to Loc/Loc'. For example, in the previous list example, to split $R = [a, b, c, d, e]$ such that e is placed in $Site1$, the actual scope in the request would be : $\{Site\ 1\%Site\ 2, local_logic = \{Site\ 1 : \{\{a, b, e\}, \{1, 2, 5\}\}; Site\ 2 : \{\{c, d\}, \{3, 4\}\}\}\}$. This specifies that the local site is $Site\ 1$ and it will execute the operation to shard the list on itself and $Site\ 2$, following the $local_logic$ added onto $Site\ 1$, there will be elements a, b and e and $Site\ 2$ will contain c and d. The numbers correspond to the position in the list when the aggregation function is applied. The logic for this scope is illustrated in Code 1.

Figure 4 gives a concrete overview of the workflow for any Cross resource. It contains the user input that gives the resource configuration, scope-lang expression, $resource_logic$ and $local_logic$ for each site. $Resource_logic$ defines what elements need to be sharded and it describes how these elements are performed during any CRUD operations and $local_logic$ are objects defined with values for these sharded elements at each site. At each site, a compilation happens

```
// Resource_logic:

Aggregation(site S, position P):
Initialize empty list L
FOR{i=1 TO len(S)}
  Get element from Site S_i based on position P_i
  Add retrieved element to list L
// reordering based on the positions obtained from each site
ENDFOR
RETURN L

Extension(list L, position P):
Get sites S from scope-lang
Divide L into elements based on P
Distribute elements onto Site S
// creates the list elements at each site based on the request

// Local_logic:
// These values below instatiates resource_logic
List {Foo, {({a,b,e},{1,2,5})}}
// Values local to Site 1
List {Foo, {({c,d},{3,4})}}
// Values local to Site 2
```

Code 1. Cross resource_logic definition for a List

for the resource configuration from the user operation and the $resource_logic$ with the values from $local_logic$ for the local site. This results in creating individual sharded resources at each site. The same process contains non sharded elements, which by default is not specified in the $resource_logic$ and they are replicated at each site.

C. Aggregation and partial errors

Aggregation acts as an interface between CRUD operations and the effective resource(s). More precisely, this interface relies on the $meta_identifier$ previously introduced, in order to (i) propagate the request to the relevant sites and (ii) aggregate the responses back. This interface hides the distribution details from the users, so that they do not need to mention the sites for an operation (updates/reads/delete).

Many times the request propagation may not be successful and can fail due to various issues (network disconnections, server crashes, etc.), a first challenge is to deliver a meaningful answer from all gathered responses. Initially, our objective was to deliver a global answer that would be close to the one returned by the native API. Unfortunately, such an approach does not allow to deal with errors that can be encountered: if only one site is not reachable, would a global error be the appropriate response? We chose to aggregate responses delivered by different sites into a collection and raise a partial error if needed. This collection of responses can inform whether a request was success or not at each site.

In the presented example (Figure 5), the aggregated response includes the HTTP status codes of the request at each site. Success response of 200 and failure responses such as $404, 408, 503$ and the message obtained at each site. User can take appropriate actions based on this information from each site.

D. Consistency

While the propagation of the requests thanks to the abstract layer may seem trivial, it is important to consider

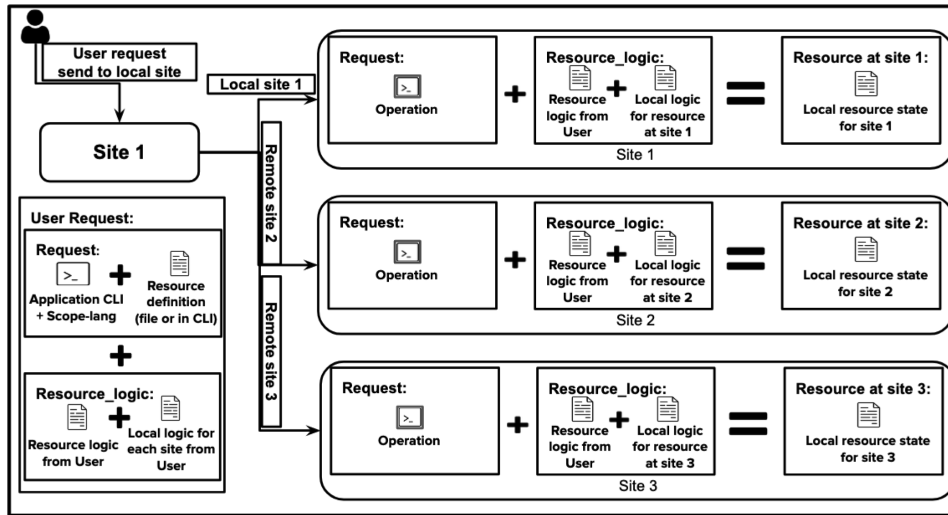

Figure 4. Cross resource workflow for an operation across 3 sites

create/update/delete requests might be performed concurrently on distinct sites or when facing network partitions. Even though all operations are executed directly as per the partial error strategy discussed previously, some cases might arise which require an eventual propagation of operations to remote sites. For example, when the name of a list is updated; this attribute is replicated in the resource due to its non-shardable nature. Such operations can lead to consistency issues, in particular for updates involving non-shardable elements.

We propose a consistency mechanism based on a distributed grow-only set which can be seen similar to a log. On each site, a new CRUD operation is tagged by incrementing a counter by 1, where the counter is the number of operations. Each site has a local log which is incremented upon receiving an operation from remote site or by the user. Operations are applied locally before propagating to the remote sites. These operations along with the tags are then send to remote sites. This idea is inspired from the Vector clock [8]. All operations are tagged locally as they are received; if the tags are the same, this implies the operations are concurrent, i.e., of the same order. In order to solve such a conflict, we look at the lexicographical order of the operations such that we obtain an eventually consistent order for operations. After obtaining the order, the concurrency is solved by applying the operations in this order.

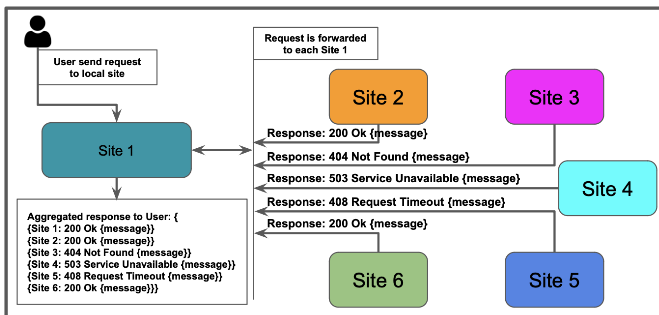

Figure 5. The aggregation with partial errors

IV. Validation

The aim of this section is to present our solution that is flexible enough to handle any resource distribution for an application based on the user needs and we create individual resources at each Site based on a combination of sharded and non-sharded elements. Our validation approach motivates the relevance of Cross and how we can achieve such as system for any application.

We validate our proposal on the Kubernetes application (k8s) due to extensive research on this topic and ease of managing API's for the entire application with a single service (Kube-API). We present an approach to extend our solution to different applications such as Sharelatex (now Overleaf). Sharelatex is an application consisting of multiple services which coordinates together to manage LaTeX files.

A. Experiments - Kubernetes

We introduce our experimental setup and detail our experiments to answer the following questions:

1) How a non-shardable element is distributed & managed.
2) How a shardable element is distributed & managed.
3) What happens to Cross under network partition and how do we manage consistency for non-shardable elements.

1) Experimental setup: Kubernetes is a software of around 4 Millions lines of code, designed to manage the lifecycle of containerized applications within a cluster. Our experiment setup consists of eight k8s instances, located at different locations and is emulated on top of Grid'5000 testbed [9]. We integrate our proposal in the available version of Cheops[2]. A Cheops agent is deployed on each cluster, and each agent is connected with the others. Our basic test cases consisted of 700 lines of code for *Cheops Core* and 500 lines of code for

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

Cheops Glue, both written in Go, in which we added the logic for *Cross*. Scope-lang used for the user interaction is another part of the code in *Cheops Core*. We introduced CouchDB into Cheops and built our consistency model for non-sharded elements.

CouchDB features align with our requirements to allow users to target specific Cheops agents for each resource and to make sure an operation for a resource eventually reaches all agents involved with that resource. It can register and queue all operations locally, selectively push them only to agents that need to receive it, survive network partition or even an agent being shut down, and frugally use the network by only activating when a new operation is known locally (whether it is started by the user or it has been replicated from another agent). It pre-checks whether an operation to be replicated already exists or not, thus avoiding duplication. CouchDB allows us to build a layer that gives us a selectively replicated set of objects; our consistency model is built on it.

2) Non-Sharded elements on k8s Namespace: A *Namespace* (abbreviated as NS) is an abstraction to isolate a set of resources from others. In this experiment, a foo NS is created with a pod a in $Site$ 1. foo is extended from $Site$ 2 to $Site$ 8. We use the default k8s CLI kubectl along with scope-lang to define such a distribution from $Site$ 1 as `kubectl create ns foo --scope {Site 1 % Site 2 % Site 3 % Site 4 % Site 5 % Site 6 % Site 7 % Site 8}`. The notation % represents the *Cross* collaboration

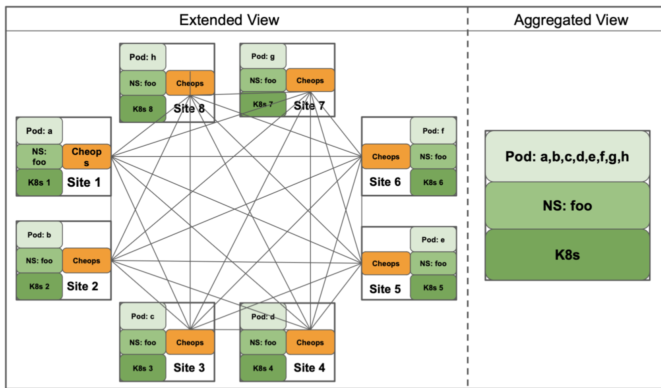

Figure 6. Cross resource: Extended & Aggregated View

for Cheops operations. Pods b to h are deployed on each Site respectively, and after the extension process from $Site$ 1 with individual command for a pod such as `kubectl create pod b --scope {Site 2 }`, we use scope-lang again to deploy a remote resource to a Cross resource. In this scenario all the elements from the configuration such as name, labels, etc. are non-shardable elements and hence, replicated. We create pod a at $Site$ 1 directly using scope-lang like `kubectl create pod a --namespace foo --scope {Site 1}`, similarly for all pods to their corresponding site. The first part of Figure 6 represents an extended view of the resource foo distributed across multiple sites. The latter part presents an aggregated view of the resource foo to give the illusion of a single Namespace foo at each resource Site. No extra code is

```
// Resource_logic:

Aggregation(sites S, resource R, configuration C):
// Concatenate non-sharded elements from all sites
FOR{i=1 TO len(S)}
        R_S = R_S + replica value at i
ENDFOR
Replace replica field with sum_of_replicas in C
RETURN (C)

Extension(resource R, value V):
Retrieve Configuration C from R and Sites S from Cross_DB
C_RS = Change replica attribute of C based on V
Apply C_RS onto Site S
// Apply each local logic based configuration (C_RS)

// Local_logic:
Deployment {bar, {replica = 2}}// Values local to Site 1
Deployment {bar, {replica = 3}}// Values local to Site 2
```

Code 2. Cross resource_logic definition for a Deployment

required to be written to geo-distribute Namespace i.e., no extension function, as there is no division of elements but for aggregation we need to aggregate the responses from multiple sites which is the default behaviour of Cross: combining the request from all of the involved site and return it to the user.

3) Sharded resource on k8s Deployment: A *Deployment* is another k8s resource that manages pods with a single source of truth. Deployments are a local cluster resource by default which contains a number of replicas defined by the configuration file. We try to geo-distribute this resource by sharding the *replica* attribute of the deployment configuration such that the user can choose how many replicas are to be instantiated at each site. Note that, all other attributes related to deployment such as name, labels, container names, etc. are non-sharded for our scenario. By default Cross replicates all elements (attributes in the configuration) which are not specified in the *resource_logic*. For a deployment *bar* requesting 5 $pods$ across $Site$ 1 and $Site$ 2, an operation is performed to create 2 $pods$ on $Site$ 1 and 3 $pods$ on $Site$ 2 with command (`kubectl create deployment --scope { Site 1%Site 2, local_logic = {Site 1:{replica = 2},Site 2: {replica = 3}}}`).

Code 2 depicts $resource_logic$ and $local_logic$ for a deployment resource. The only sharded attribute *replica* from the configuration file of the deployment given by the user is added onto the $resource_logic$. It can be called by the $local_logic$ by $Deployment\{bar, \{replica = 3\}\}$ at site 1 and $Deployment\{bar, \{replica = 2\}\}$ at site, indicating the total number of replicas=5. The aggregation and extension function defines how to handle the distribution of the resource and the $local_logic$ instantiates $resource_logic$ with values for sharded resource at each site.

4) Network partition and Consistency for Cross: We consider both sharded and non-sharded elements individually to show how they behave under network partition. For a Deployment resource in which we consider shardable and non-shardable elements, if an operation is performed during a network partition on a shardable element, it needs to be propagated to all the sites eventually. There is no scope for synchronization, as the elements are not replicated. In case of a

non-sharded element, i.e.,, for a resource such as a Namespace, if an operation is performed during a network partition, the operations are applied locally and when network is regained they are eventually applied onto the remote site. In such a case, we need to perform a synchronization of the resource and need to resolve any conflict.

To test our solution under network partition scenario, we simulated it by disconnecting a CouchDB instance at $Site3$. We tried to update a *label* (which is an element in Namespace consisting of a key-value pair) with an initial value *colour:brown* for all sites before disconnection. After disconnection, at $Site1$, we set the value as *colour:blue* and at $Site3$, *colour:red*. The *colour:blue* update returned with a partial error with REST code 200 from $Site1$, $Site2$ (success) and 404 from $Site3$ (No connection). For the update from $Site3$, 200 was returned for $Site3$ and 404 from $Site1$ and 2. Local log was updated on sites as per the local operation. When network came back, there was a conflict as the two operation were concurrent. This proved that eventually the operations are sent to remote sites after a network disconnection. The order was determined lexicographically with $Site1$ operation getting a precedence and the conflict was eventually resolved. We tried for all CRUD operations, for cases that required synchronization, we got the expected results for these cases.

B. Use-case: Extending Cross to Sharelatex

This analysis focuses on an Sharelatex application, we did not do an experiment, this is a future work we are proposing and here we demonstrate how our geo-distribution approach can be extended to multiple applications. Sharelatex (now part of Overleaf), is a web-based collaborative platform for working with LaTeX, a typesetting system commonly used for scientific and mathematical documents. It allows users to create LaTeX documents, enabling real-time online collaborations and rich text editing. It is composed of multiple services in a single Site as explained in the research [1], which introduces a proxy approach towards distributing these services in order to geo-distribute them. The proposed approach is too intrusive to the application and it can be prone to errors during network disconnections as it is not based on a local-first and generic approach.

We follow a local-first approach for our application such that an entire instance of Sharelatex is deployed at each site. The major resource for a Sharelatex application is the LaTeX project, which is a resource that isolates all the files related to a single research document. We consider all the configuration elements related to a project such as name, labels etc as non-shardable resource, since sharding these identifiers with local values does not bring any added value and can create problem for referring the resource. The sub-resources (files) inside a project which are the elements that can be sharded with respect to each location such that the overall project size can be reduced at a Site and they can be local to a site.

For example: A LaTeX project foo which has different files *main.tex, image1.png, image2.png, intro.tex, conclusion.tex* deployed at $site$ 1. Extending and geo-distributing foo with Cross to $site$ 1 and $site$ 2 creates sub-resources foo' and foo'' respectively. foo' consists of *main.tex, image1.png, image2.png* sub-resource and foo'' consist of *intro.tex, conclusion.tex*. Such a division is facilitated by the extension principle and the sharded sub-resources are mapped onto *local_logic* as an object to *resource_logic*. All CRUD operations from the user or from other services will get the illusion of foo being a single resource present at the local site. The resource behaves like the k8s Namespace resource, but presents more challenge with respect to consistency aspect and for a compile operation in Sharelatex, which combines all the files to create a single file. The approach demonstrates portability to multiple applications, offering advantages like localizing processing to a single site or ensuring data locality and offers flexibility.

V. Related work

DIMINET [10] is an early research on distributed inter-site networking services, focusing on local site networking and communication with remote modules. The concept highlighted in this research of collaboration between independent systems has been extended in our research on a much generalized method to include more resources.

Single system Image [11] was another research that hides the heterogeneous and distributed nature of available resources, presenting them as a single unified computing resource. We extend this concept to all the CRUD operations for the application layer as we also deal with the splitting and placement of the resource at sites while handling network disconnections. Volunteer computing [12] involves participants downloading client software to receive and process tasks from a central server. It follows a *Multiple program Multiple data* parallelism, on contrast *Cross* follows a *Single Program Multiple Data* (SPMD), involves multiple individual sub-resources working a resource but with different sharded elements.

Techniques such as RAFT [13] and CRDT [3] are frequently employed for the geo-distribution of resources. RAFT is a consensus algorithm, hence it fails to hold a local-first approach such that executing operations at local sites is not possible without the consensus, a challenge addressed in our research. On the other hand, while CRDT offers a local-first approach for geo-distributing resources, it often necessitates modifications to the existing application, making it excessively intrusive to the application code — a concern we aim to mitigate in our approach. Our proposed methodology is characterized by its generality and independence from application code modifications.

Shard Manager [14] from Meta and Slicer [15] from Google provides resource management systems that partitions tasks across the application. These solutions offer a specific sharding approach, requiring users to design a micro-service capable of sharding in a prescribed manner. Our model presents a general programming model by identifying the elements that can be replicated or sharded within a single micro-service by the user to any existing application.

Solutions such as Kube-edge [16], Mck8s [17], Picasso [18], etc are either too intrusive or ad-hoc to the application, hence,

we utilize Cheops [6] framework to handle geo-distribution in s a generic manner. They present a full replication approach towards geo-distributing a resource which we try to avoid in our model. A decentralised Kubefed [19] provides a similar functionality specific to the platform. The framework proposes a distribution model for deployment resource in Kubernetes which is tightly bound to the configuration of the resource, we try to separate distribution from the buisness logic such that we can provide it with an external service.

VI. Conclusion

The distribution of cloud legacy applications across geographical locations has been a longstanding focus within our community for several years. A prominent suggestion advocates the utilization of CRDT but it re-implements a significant portion of these applications. Recent initiatives have commenced investigating the feasibility of achieving geo-distribution without introducing intrusive modifications. Drawing upon prior research that substantiates this methodology, we demonstrate that a comprehensive replication strategy remains not entirely pertinent.

In our solution, we present a collaborative approach aimed at extending a resource across multiple sites while creating the illusion of a unified entity. Our proposal introduces an alternative method for geo-distributing a resource without relying on a complete replication strategy, incorporating a sharding technique.

The implementation of such collaboration encounters diverse challenges, including synchronization overheads and potential network partitions. To address these issues, the solution strategically handles requests partially. For resources that require a synchronization, we proposed a method to synchronize them in an agnostic manner. We are aware about the drawbacks of our consistency model for some scenarios, to which we are working on by creating a more specific method that can accommodate more data types and applications while being non-intrusive to the application. We substantiate the applicability of our approach through illustrative examples involving different resource types and experimentation within a Kubernetes application.

Drawing parallels with the concepts of sharing and replication previously proposed in Cheops, we posit that this new collaborative method, named Cross, holds the potential for further adaptation to any CRUD application. This paper contributes to advancing the goal of decoupling the intricate business logic of contemporary applications from the common challenges associated with geo-distribution, offering an effective solution through an external service.

References

[1] G. Tato, M. Bertier, E. Rivière, and C. Tedeschi, "Sharelatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application," in *Proceedings of the 3rd Workshop on Middleware for Edge Clouds & Cloudlets*, 2018, pp. 8–15.

[2] R.-A. Cherrueau, M. Delavergne, and A. Lèbre, "Geo-distribute cloud applications at the edge," in *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*. Springer, 2021, pp. 301–316.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 2011, pp. 386–400.

[4] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter, "Nosql database systems: a survey and decision guidance," *Computer Science-Research and Development*, vol. 32, pp. 353–365, 2017.

[5] M. Delavergne, R.-A. Cherrueau, and A. Lebre, "A service mesh for collaboration between geo-distributed services: the replication case," in *Agile Processes in Software Engineering and Extreme Programming–Workshops: XP 2021 Workshops, Virtual Event, June 14–18, 2021, Revised Selected Papers 22*. Springer International Publishing, 2021, pp. 176–185.

[6] M. Delavergne, G. J. Antony, and A. Lebre, "Cheops, a service to blow away cloud applications to the edge," in *International Conference on Service-Oriented Computing*. Springer, 2022, pp. 530–539.

[7] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, "Shard scheduler: Object placement and migration in sharded account-based blockchains," in *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, ser. AFT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 43–56. [Online]. Available: https://doi.org/10.1145/3479722.3480989

[8] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," 1987.

[9] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[10] D. E. Sarmiento, A. Lebre, L. Nussbaum, and A. Chari, "Multi-site connectivity for edge infrastructures: Diminet: Distributed module for inter-site networking," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 121–130.

[11] R. Buyya, T. Cortes, and H. Jin, "Single system image," *The International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 124–135, 2001.

[12] L. F. Sarmenta and S. Hirano, "Bayanihan: Building and studying web-based volunteer computing systems using java," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 675–686, 1999.

[13] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014.

[14] S. Lee, Z. Guo, O. Sunercan, J. Ying, T. Kooburat, S. Biswal, J. Chen, K. Huang, Y. Cheung, Y. Zhou *et al.*, "Shard manager: A generic shard management framework for geo-distributed applications," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 553–569.

[15] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter *et al.*, "Slicer:{Auto-Sharding} for datacenter applications," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 739–753.

[16] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium On Edge Computing (SEC)*. IEEE, 2018, pp. 373–377.

[17] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021, pp. 1–10.

[18] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, "Picasso: A lightweight edge computing platform," in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. IEEE, 2017, pp. 1–7.

[19] L. Larsson, H. Gustafsson, C. Klein, and E. Elmroth, "Decentralized kubernetes federation control plane," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2020, pp. 354–359.