A Service Migration Strategy for Resilient Multi-Domain Networks in Outage Scenarios

Federico Giarré, Yassine Hadjadj-Aoul, Soraya Aït-Chellouche

A Service Migration Strategy for Resilient Multi-Domain Networks in Outage Scenarios

Federico Giarré*, Yassine Hadjadj-Aoul*, Soraya Aït-Chellouche*
*Univ Rennes, Inria, CNRS, IRISA, France

***Abstract*—Network slicing has emerged as a crucial technology in 5G and post-5G networks, enabling the creation of customized logical networks on a shared physical infrastructure. However, the problem of service placement, which entails determining the optimal deployment of network functions within each slice, becomes particularly challenging during outage scenarios. Service migration serves as a critical technique for reconfiguring and relocating network functions, thereby ensuring the resilience and availability of the slices. This paper presents a service migration strategy specifically designed for resilient multi-orchestrator multi-domain networks. Our approach considers multi-domain services composed of components deployed across different network domains using Kubernetes clusters. The abstract nature of this migration strategy allows for its adaptation to different orchestrators within a cluster, facilitating the integration of more diverse and heterogeneous systems.**

***Index Terms*—Network function virtualization, Resiliency, Service Migration.**

I. Introduction

Network slicing is a key technology for 5G and post-5G networks, as it allows the creation of multiple logical networks on top of a shared physical infrastructure, each tailored to a specific service or application. In this context, service placement is a crucial problem, as it determines how and where to deploy the network functions composing each slice.

Service placement becomes even more complex in outage scenarios, where some network elements may fail or become unreachable, degrading the performance and availability of the slices. To cope with such situations, service migration is essential, as it allows for a dynamic reconfiguration and relocation of network functions to ensure the resilience and availability of the slices.

In this paper, we propose a service migration strategy for resilient multi-orchestrator multi-domain networks. We consider multi-domain services that consist of components deployed in Kubernetes clusters across different network domains, but the strategy is abstract enough to be implemented in other systems. While there have been some advancements in the field of service migration with geo-distributed storage [1], our current study aims to explore the unique challenges posed by multi-domain networks. To ensure a comprehensive analysis, we will specifically focus on stateless services, allowing us to delve deeper into the core strategy itself.

Existing work on this scenario mostly involves the migration of monolithic applications [2]. An interesting concept can be found in web migration, where a web service can be migrated according to the requested or preferred resources that the service requests. In our work, we migrate the service containers by killing them and creating a new service on another cluster. This method results in better performance overall for stateless containers, even if the new container needs to warm up again to build a cache [3]. For stateful migration, this is true only if the time taken to compute a certain request is not too large.

The main contributions of this paper include the following: i) The description of a strategy for zero-touch, transparent migration of lightweight virtualized services over multiple domains. ii) The description of the proposed framework to implement the previous points. iii) The description of a testbed for testing different strategies and infrastructures.

In this paper, we provide a detailed description of the proposed migration strategy, in Section II. Subsequently, we present the experimental framework utilized and discuss the achieved results, in Section III. Finally, we summarize the key findings of this paper and outline future envisioned research, in Section IV.

II. Multi-orchestrator Multi-domain Service Migration

In this section, we analyze in depth the envisioned architecture for service migration in multi-orchestrator multi-domain networks. The main components of the envisioned infrastructure are: *1)* the Meta-Orchestrator and *2)* the Clusters. These components will be deployed on different networks. However, assuming an environment where all resources can contact each other throughout the internet, we can ensure services' continuity during migration and complete transparency for services/users trying to use the migrated service. Indeed, end-users will always connect to a single endpoint for all the time spent using the service.

A. Meta-orchestrator

A Meta-Orchestrator (MO) plays a crucial role as an entity equipped with comprehensive knowledge of all available network resources. Its primary responsibility is to determine the optimal placement for specific resources within the network. Initially, the MO is deployed in a centralized manner to ensure persistence and replication

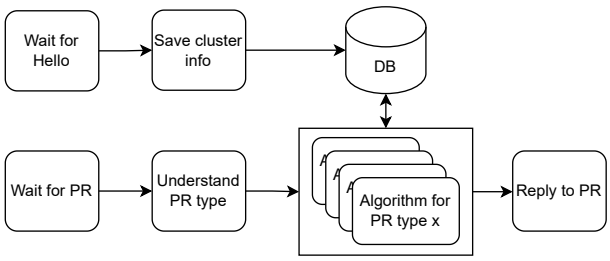

Figure 1. Lifecycle of the Meta-Orchestrator

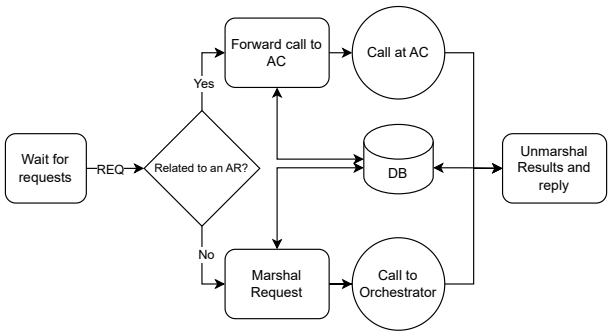

Figure 2. Flowchart of Controller handling API requests

capabilities. The MO operates by processing two distinct types of messages: *1)* Hello messages and *2)* Placement requests.

1) Hello messages: Clusters periodically send Hello messages (HMs) to the MO to indicate their active participation in the architecture. Along with this notification, clusters also transmit essential parameters that can aid in making informed decisions during resource migration. These parameters include the current load of the cluster, available resources within the cluster, and other relevant information. The periodicity of the HMs keeps the MO up to date with the various domains' conditions, leading to virtually optimal placement decisions.

2) Placement requests: Clusters generate Placement requests (PRs) when they require a resource to be migrated. The Cluster sends a descriptor of the resource in order for the MO to understand the type of issued request (e.g., new placement due to lack of resources, new placement due to changes in the network conditions, etc.). The MO eventually replies to the PRs with a valid destination for the resource to be placed.

Figure 1 illustrates a flow chart depicting the lifecycle of the Meta-Orchestrator (MO). The functions presented in the chart are essentially based on message-passing paradigms, indicating that alternative implementations of the system can be tailored to suit specific use cases (i.e., centralized, decentralized, flat, hierarchical, etc.).

B. Clusters

We consider as a Cluster all the Clusters that have an integrated orchestrator capable of acting as an "entry point" to which we can submit services for deployment. We define Home Cluster (HC) as the cluster that originally hosts a service, and Abroad Cluster (AC) as the cluster that hosts a migrated resource ("Abroad" with respect to the HC).

At the infrastructure level, we assume that the Clusters can directly reach each other without accounting for multi-hop topologies. Each Cluster will also need to configure a *Controller* to interact with the Cluster orchestrator and a *Load Balancer* in order to redirect traffic.

1) Controller: The Controller deploys some APIs, that will enable external actors (i.e., Users, other Clusters, etc...) to interface with the Cluster's orchestrator and commit operations and needs to marshal/unmarshal commands received via the APIs exposed into the Clusters' orchestrator's language. If an API call wants to act on a migrated resource, the Controller will forward the request to the Cluster deploying such a service using the same API call. The latter is an advantage when building heterogeneous systems, since the APIs would provide enough abstraction to implement this strategy regardless of the type of Cluster deployed. Additionally, every Cluster must keep entries for tracing resources that have been deployed:

- Local Resources (LR): Services that are deployed on the Cluster that owns them;
- Abroad Resources (AR): Services that are deployed in an AC. The HC will keep track of where the resources are;
- Foreign Resources (FR): Services that are deployed in the Cluster, but are owned by another Cluster. The Cluster will keep track of the HC of the resources.

2) Load Balancer: To achieve the desired level of resiliency in our work, we find it impractical to rely on the time required to update DNS records. In order to address this challenge, we employ a Load Balancer (LB) to redirect traffic. Much like Mobile IP [4], users and other services will point to the HC of the migrated service, where a LB will just redirect the traffic to the AC deploying the AR.

C. Migrating Resources

Although migration can be triggered by a large number of factors, it always take place in five stages: *1)* triggering, *2)* placement request, *3)* replication, *4)* rerouting and *5)* cleanup.

A visualization of the following steps can be seen in Fig. 3.

1) Triggering: In this phase, the Controller recognizes that a resource needs to be migrated. This could happen for environmental factors (i.e., bandwidth, energy consumption, etc.) or specifically triggered by an API call.

2) Placement Request: The Controller issues a PR to the MO. In detail, the Controller sends a descriptor of the service to be moved so that the MO can take the decision accordingly.

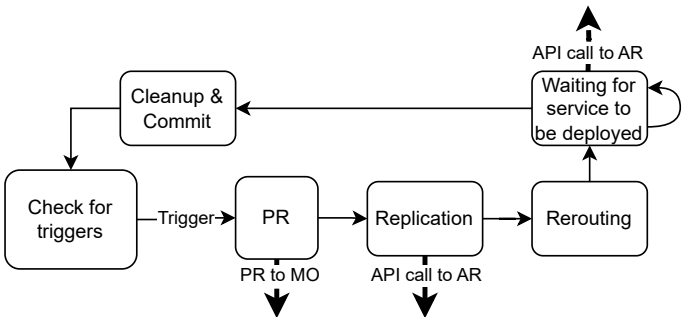

Figure 3. Flowchart of Controller handling API requests

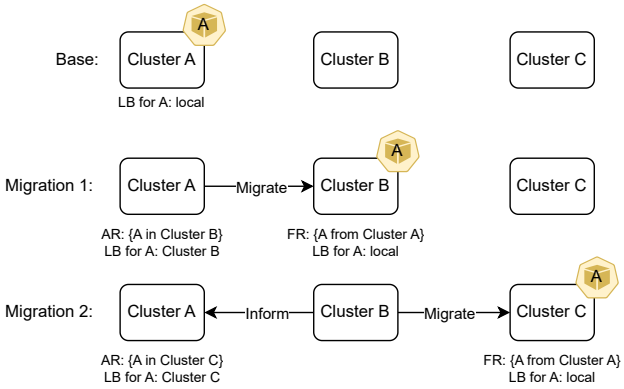

Figure 4. Migration of resources

3) Replication: The Controller issues an API call to the AC returned by the PR, requesting the deployment of the service. The Controller also attaches information about the HC to the call, so that the AC is aware that the service deployed is the result of a migration. The service is now considered as an AR from the HC, and as a FR from the AC. In the case where the Controller issuing the call is not the HC of the service, the procedure remains mostly unchanged. In fact, the Controller issuing the migration in the latter case updates the HC of the FR about the further movement, making the HC update its knowledge with the new AC for the AR. A more accurate description of this process can be seen in Fig. 4.

4) Rerouting: The HC's Controller[1] prepares the remapping of the AR, changing the LB configuration (but not committing it).

5) Cleanup and Commit: Only when the Controller has verified that the resource has been migrated successfully (performing API calls to the AC), it will proceed with deleting the service from the cluster and committing the LB configuration. This step ensures a transparent transition and virtually zero downtime for the application

D. Example of migration

In Fig. 4, we summarized an over-simplification on how a resource will be moved between Clusters, and how the Controllers will modify their configuration accordingly.

1) Base case: In this case, service A is deployed on cluster A, its own home cluster. The LB will point to the cluster normally.

2) Migration 1: Cluster A sends a PR to the MO, and receives as a reply that Cluster B would be suitable for the migration. Cluster A follows the upper cited steps in order to successfully migrate service A from Cluster A to Cluster B, at the end of it the LB of Cluster A will point to B. Moreover, Cluster A will insert service A in the records of ARs. Cluster B will have the LB pointing locally for service A, but will register the service as a FR whose HC is Cluster A.

[1]Recall that all traffic passes from the HC's LB, so only that configuration has to be modified

3) Migration 2: Migration takes place again, Cluster B informs Cluster A of the further movement, and Cluster C of the HC of service A. Doing so, Cluster A will modify its AR record and LB to make them point to Cluster C, while Cluster C will record a new FR coming from A and have the LB to point locally to it. After the migration has been completed, Cluster B will remove any record regarding the migrated service (since it is not the service's HC or AC, there is no need for further tracking).

Even if not shown in Fig. 4, we also implemented a fail-safe mechanism, to be able to re-deploy services deployed in ACs in case of network partitioning. Conversely from the migration described above, this fail-safe mechanism is visible to the end users, who would experience an unavoidable downtime due to the re-deployment.

III. Experimental Framework and Results

In this section of the paper, we will analyze the use-case scenario and the framework we used in order to emulate it and test the strategy.

A. Testbed and setup

In order to test the use case, we created the following setup: Flask [2] for creating REST APIs, RabbitMQ for AMQP communication [3], Containernet for network emulation [5], Minikube which is a Kubernetes deploy using docker [6], and Sysbox which is a runtime environment for complete isolation of docker in docker [4].

What we successfully created thanks to these software is a testbed that allows us to spawn different fully functioning Kubernetes clusters in a network emulator, and then deploy, remove, and migrate services between them. The advantages of creating such an environment is to have a fast, highly reproducible sandbox where we easily spawn as many clusters as we need, make them interact, and test the strategy while also having a complete control over the underlying network infrastructure.

[2]https://flask.palletsprojects.com/
[3]https://github.com/rabbitmq
[4]https://github.com/nestybox/sysbox

Table I
EXPERIMENTAL RESULTS

PR	Replication	Reroute	Wait	Cleanup	Deploy
$3ms$	$21ms.$	$47ms$	$\sim 10s$	$\sim 1.5s$	$\sim 9.5s$

B. Chosen use case

As a use case, we decided to aim for network bandwidth variations in outage scenarios. In this scenario, two or more clusters are present and are labeled by their position as deep-edge, edge, and cloud clusters. The idea is to have a streaming service (from cameras, sensors, ...) in the deep-edge area and a classification service (i.e., a computer vision service) whose position can vary depending on the network availability. For instance, with bad network conditions (or in the case of network partitioning) the classification service can be placed on the same cluster as the streaming service, although working poorly due to the low computing power of the deep-edge. In the case of good bandwidth, the classification service can be moved to the edge, since there would be more computing power, or even to the cloud if the bandwidth is good enough.

C. Tests and Results

In Table I the experimental results of the tests performed on the testbed described above are presented. In particular, the results refer to the migration phases of an nginx deployment over more experiments and the time taken to deploy one pod of the deployment and the service to access it. Experimentally, the migration is indeed transparent to the users, who don't experience loss of requests issued to the services deployed. In the case of connection failure between an HC and a AC, the ARs on that AC get redeployed on the HC as soon as the HC triggers the fail-safe mechanism. Thanks to the results in Table I, we can draw a series of conclusion: *i)* Optimal positioning of the MO (as in these tests) allows for a seamless, almost instantaneous PR phase. *ii)* Since the HC is sending only descriptors of the services to migrate, the Replication is going to be light no matter the service, and mostly dependant on the connection between Controllers.*iii)* The Reroute phase consist in modifying the local configuration of the LB, thus the time cost should remain constant in every scenario. *iv)* The Waiting time is highly dependant on the Deploy time, optimal combinations of hardware and orchestrator settings can reduce the Deploy time, impacting the Waiting time *v)* In case of failure, the downtime expected for the user is solely made of the Deploy time and the time used to sense the failure. Optimal sensing for the specific use-case analyzed will lead to a reduced downtime.

IV. CONCLUSION

This paper presented a functioning strategy for migration that can be adapted at will for more specific use-cases. The focus has been on how to make a zero-touch migration

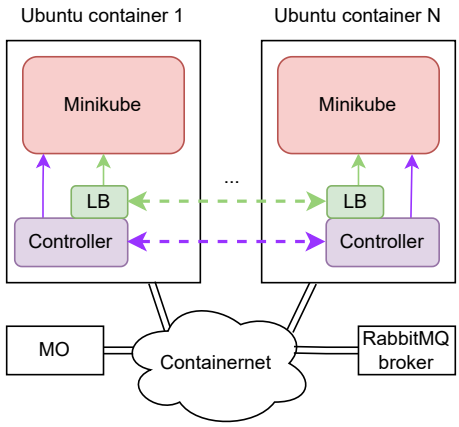

Figure 5. Testbed with containernet

feel transparent and resilient in case of different events (variation of bandwidth, down-links, etc.) that are likely to happen in an outage scenario. This paradigm is simple, effective, and highly scalable, enabling other paradigms such as multi-tenancy. This paradigm is also abstract enough to be adaptable to any Cluster's orchestrator, opening the path for more heterogeneous systems.

One possible extension to the proposed architecture is the decentralization of the MO. Although a centralized MO implementation is simple, effective, and scalable with additional replicas for fault tolerance, it faces challenges in network partitioning scenarios. Excluded clusters lack the ability to migrate resources. By decentralizing the MO and integrating it as a functionality of the Controller, this problem can be addressed.

V. ACKNOWLEDGMENT

This research work is conducted as part of the NaviDEC project, funded by the Brittany Region, France, under grant agreement #21004179.

REFERENCES

[1] Jemal H. Abawajy and Mustafa Mat Deris. Data replication approach with consistency guarantee for data grid. *IEEE Transactions on Computers*, 63(12):2975–2987, 2014.

[2] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. Live migration of containerized microservices between remote Kubernetes Clusters. working paper or preprint, March 2023.

[3] Y.C. Tay, Kumar Gaurav, and Pavan Karkun. A performance comparison of containers and virtual machines in workload migration context. In *2017 IEEE 37th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, pages 61–66, 2017.

[4] C.E. Perkins. Mobile ip. *IEEE Communications Magazine*, 35(5):84–99, 1997.

[5] M. Peuster, H. Karl, and S. van Rossem. Medicine: Rapid prototyping of production-ready network services in multi-pop environments. In *2016 IEEE Conf. on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 148–153, Nov 2016.

[6] Ruchika Muddinagiri, Shubham Ambavane, and Simran Bayas. Self-hosted kubernetes: Deploying docker containers locally with minikube. In *2019 Int. Conf. on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, pages 239–243, 2019.