Enhancing microservices architectures using data-driven service discovery and QoS guarantees

Zeina Houmani, Daniel Balouek-Thomert, Eddy Caron, Manish Parashar

Enhancing microservices architectures using data-driven service discovery and QoS guarantees

Zeina Houmani*†, Daniel Balouek-Thomert †, Eddy Caron*, Manish Parashar†
*Inria Avalon team, LIP Laboratory, UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668
University of Lyon, France
†Rutgers Discovery Informatics Institute
Rutgers University, NJ, USA

***Abstract*—Microservices promise the benefits of services with an efficient granularity using dynamically allocated resources. In the current evolving architectures, data producers and consumers are created as decoupled components that support different data objects and quality of service. Actual implementations of service meshes lack support for data-driven paradigms, and focus on goal-based approaches designed to fulfill the general system goal. This diversity of available components demands the integration of users requirements and data products into the discovery mechanism. This paper proposes a data-driven service discovery framework based on profile matching using data-centric service descriptions. We have designed and evaluated a microservices architecture for providing service meshes with a standalone set of components that manages data profiles and resources allocations over multiple geographical zones. Moreover, we demonstrated an adaptation scheme to provide quality of service guarantees. Evaluation of the implementation on a real life testbed shows effectiveness of this approach with stable and fluctuating request incoming rates.**

***Index Terms*—Microservices, data-driven, quality of service, software architecture, load shedding.**

I. Introduction

MicroServices Architectures (MSA) has gained great popularity in the recent years, exploring benefits of modular, self-contained components for highly dynamic applications. Each microservice operates as an independent function, offering access to its internal logic and data through network interfaces and defined APIs. MSA are widely used in the industry for applications where scalability, resiliency, and availability are required, as is the case for Netflix[1] and Uber[2] (among others). Recently, Internet of Things (IoT) [1]–[3] and network virtualization [4] applications started adopting this paradigm.

MSA are driving the continued evolution of Service Discovery (SD) [5]. Service discovery has a growing importance as the services providers increase in number and complexity. In particular, services may have different costs along different metric dimensions, and can adapt themselves and their interactions based on requirements and execution contexts. In current practice, SD implementations are goal-based, designed to fulfill the general system goal depending on the user's required functionalities. They often consist in letting the client discover the location of a provider for the requested service using its identifiers.

Additionally, emerging Cloud and Edge computing systems are data-driven. Producers and consumers of data are decoupled, using different data formats, resolution and expected Quality of Service (QoS). This highlights several limitations of goal-based approaches for service discovery. First, using approaches built on service identifiers, prevents the discovery of newly created microservices as well as those that do not have explicit identifiers. In addition, describing services using data models based primarily on network information and identifiers prevents client programs from using microservices that meet their specific needs. In this context, this paper addresses the challenges of discovering microservices based on client's data object with regards to a guaranteed QoS.

This paper proposes a data-driven SD framework to tackle these issues. This framework enables the creation of context-aware microservices architecture, able to allocate resources and replicate services in a federated environment. On the client side, it allows the discovery of services according to objects and data products. On the provider side, it allows for the integration of third party services with context-aware features.

Additionally, we put an emphasis on implementation and integration into an existing service mesh project. That requires the creation of new components to overcome the limitations of the chosen service mesh.

Our approach builds on a data-driven model for microservices and a peer-to-peer architecture to ensure performance, resiliency and scalability of application and management services. This paper makes the following contributions:

- A data-centric model of microservice.
- A service discovery process based on the client's data products and integrated in a data-driven architecture.
- An adaptation scheme to manage processioning based on given services demands.
- An implementation into the Istio service mesh project [6].

The rest of this paper is organized as follows: Section II presents targeted applications. Section III presents related work. Section IV describes our data-driven microservice model. Section V presents the architecture, while Section VI presents the algorithms for resource adaptation. Validation is performed using Istio service mesh in Section VII. Section VIII concludes the paper and discusses future work.

[1]Adopting Microservices at Netflix: Lessons for Architectural Design: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[2]Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow:https://eng.uber.com/soa/

II. Motivation

In traditional MSA, the service discovery mechanism consists of discovering the list of instances that can provide a functionality already known to the client. In the current computing landscape, Edge computing and Internet of things services present temporal, spatial and always-on features. This implies the ability to provide context-aware services answering the requests of a massive numbers of objects, sensors and devices connected to the network, using data products of different nature along with changing demands.

This work is motivated by applications built in dynamic environments where developers frequently publish new microservices but they are not discovered by the users. In addition, we target systems that deal with heterogeneous services offering same functionality but with different resolution and QoS guarantees. These use cases highlight the need for data-driven discovery approach capable of discovering relevant microservices dynamically based on client's data. Such flexibility is not currently offered in state-of-the-art microservices and service mesh. Major systems, such as [6] or [7], do not offer data-driven service discovery support to provide and guarantee the performance of services. Therefore, the ability for developers to compare multiple services versions is limited.

In all existing SD software [5], the client that produce the data and use the discovery process is designed by the same entity that has developed the consumer services. However in Edge computing, for example, data and functions are not in the same location and do not come from the same entities [8]. In these systems, service discovery is not a guaranteed process. For this reason, a service discovery independent from data consumers and data producers became a necessity.

The first use case is an edge-based video analysis system. This system relies on a deep-learning pipeline distributed between edge, in-transit and cloud resources [9]. At each stage of the pipeline, the required service is deployed on one or more resources according to the number of data products appearing in each frame. In this system, producers and consumers are often geographically distributed. Also, it exists many implementations for the same functionalities, for example, object detection algorithms have more than 40 possible implementations, such as Yolo detection or exhaustive search. Each of these implementations makes a different design choice and consume different resources. Thus, in order to choose the best implementations for each stage of this deep learning pipeline, it is necessary to allow for the use of data-products to discover services and a guarantee of quality to ensure the timely completion of analyzes.

The second use case is a data streaming framework that utilizes heterogeneous geo-distributed resources to maintain application's quality of service. In this work, adjusting the service in quality is on the side of the client. With the existing mapping solution, finding the optimal path may take too much time if the number of resources increases significantly. In order to effectively map the workflow stages to the appropriate nodes in a dynamic architectures, a data-driven SD can be used.

These use cases highlight the need to discover services not only based on their identifiers, but also with regards to context and IoT-like features.

III. Related work

Assigning network addresses manually to services is no longer possible in the current dynamic microservices architectures. Therefore, three different services discovery strategies are currently adopted: DNS-based service discovery [10], specialized service discovery solutions such as Netflix Eureka [11] and consistent datastores such as Apache Zookeeper [12], etcd [13] or MySQL [14]. In our approach, we assume users do not know in advance what microservices are available. In addition, we aim at using a specific data-centric service description model to enable a data-driven discovery process. Relational/non-relational datastores provide the necessary strategy for such implementation.

As the scale of applications rises, the effort required to manage inter-microservices interactions increases, and becomes an extremely complex task in large-scale architectures. As Microservices architectures are characterized by their technological heterogeneity, the integration of libraries into the technological architecture stack requires additional modifications depending on each microservice. Service Mesh [15] provides a viable solution to these challenges. Service Mesh creates a dedicated infrastructure layer for decoupling inter-service communication management from the deployed microservices. This additional layer forms a distributed network of interconnected proxies within the microservices network. Using this pattern, communications between microservices pass through the proxy servers deployed in *sidecars* next to the microservices without the need for any code modification. These proxies are responsible for translating, transmitting, and observing each network packet that flows to and from a microservice instance. Since 2016, several major open source platforms have been created such as Linkerd [16], Conduit [17], Istio [6] and recently the AWS App mesh project [18]. We chose the Istio open source project for this work. Istio uses the Envoy proxy [19] as its data plane which presents more functionalities than other proxies at this time. Other studies highlight the performance and extensible features associated to this service mesh [20], [21].

The quality of service delivered by service-based systems represents a major concern that needs to be managed. Since these systems are characterized by their dynamicity and their continuous changes in requirements, adaptation capabilities must be implemented in order to maintain system's objectives during executions [22]. Different paradigms to achieve adaptation have been developed. In this work, we are interested in self-organizing adaptive approaches SOAS to creates autonomous systems [23]. Self-adaptation systems, introduced by IBM [24], are able to automatically adapt themselves by modifying their behavior in response to changes in their operating environment without any external control [25], [26]. Adaptive systems can usually be implemented in centralized

and decentralized way [27]. In our actual architecture, the centralized approach was used. This approach consist of having central components that receive data from other components and take appropriate actions to maintain QoS.

IV. Data-driven service discovery process

Service discovery in microservices architectures usually implements two patterns: client-side service discovery [28] and server-side service discovery [29]. When implementing client-side discovery, the client is responsible for initiating the service discovery process and selecting the desired instances. However, with service-side discovery, an intermediate component act as middle-man to intercept client requests and complete the discovery process, while abstracting the discovery details from the client.

Our approach aims at designing a data-driven service discovery process that allows the matching between data products and services while using a hybrid service discovery pattern. We consider two design goals for this approach. First, a data model that enables the discovery and classification of microservices deployed in the platform. Second, a communication protocol that combines the two service discovery patterns. This gives the client full control over the discovery process while ensuring an authorized access to registered data. Additionally, it avoids bottlenecks that could occur in the intermediate components when an important number of requests enter the system due to the growing demands for services. This protocol consists of interactions between the client, the service registry and the API gateway.

A. Data Model

The main goal of service discovery is to show the clients the available microservices deployed in the platform. To this end, each microservice registers in the platform using a data model to declare its availability for the discovery clients. This data model usually contains at least a service name and information about the network location of the service provider. Consul project [30], for example, use a service description model that contains network configurations as well as service identification data such as service name, ID and tag. Some of these information can be removed but no additional properties can be added.

However, in a data-driven service discovery, additional information related to the service performance and the functionality provided should be specified. Listing 1 presents a microservice description. Input type, input parameters and measured performance are considered as the main properties in the microservices profiles that are examined in the data-driven service discovery process. When a client program initiates the discovery process, information concerning the properties of the client's data must be specified such as the data type, the data format or the size.

Listing 1: Data model describes functionalities and network features to match data producers and third-party microservices. The following sample presents a `crop` microservice associated to the type *Image*.

```
<ApplicationID>Crop</ApplicationID>
<Description>This service is to remove the unwanted
      areas of an image</Description>
<InstanceVersion>v1.2</InstanceVersion>
<Hostname>service.com</Hostname>
<AddressIP>10.96.0.11</AddressIP>
<Port>9500</Port>
<SecurePort>443</SecurePort>
<Protocol>http</Protocol>
<Interface>/crop</Interface>
<Status>UP</Status>
<InputType>image</InputType>
<OutputType>image</OutputType>
<MaxInput>12</MaxInput>
<MinInput>1</MinInput>
<MaxInputSize>50MB</MaxInputSize>
<MaxPixelDimension>2000x15000</MaxPixelDimension>
<InputFormat>
  <format>PNG</format>
</InputFormat>
<RPS>100</RPS>
<Uptime>1567889</Uptime>
<HealthCheck>/health</HealthCheck>
<Parameters>
  <Parameter id="areaWidth">
    <ValueType>int</ValueType>
    <Description>Width of the image area to
        extract</Description>
    <required>true</required>
  </Parameter>
  <Parameter id="file">
    <ValueType>string</ValueType>
    <Description>The location of the local/distant
        input object </Description>
    <required>true</required>
  </Parameter>
  ...
</Parameters>
```

If these properties match exactly the characteristics of the object supported by a microservice, its data model is considered as a matching profile. This profile is then added to a list and returned to the client as one of the microservices he can use. However, if a strict matching is considered and one of these properties differs, the profile is not matched. This model is extensible, with anticipation to an increasing number of properties and data types existing within a service mesh.

B. Client-registry communication strategy

The hybrid discovery pattern used in this platform involves two components during the discovery process: the service registry and the API Gateway.

The **service registry** represents a database cluster that contains the data model of available microservices deployed in the platform. This database must be highly available in order to discover existing microservices at any time. As in our architecture, new instances can be created and destroyed dynamically, this component must be continuously updated. When a new service instance is deployed, its data model is registered in the service registry to declare its availability. This service description is removed when the microservice is no longer available. Two different patterns exist to handle the registration and unregistration of microservices within the service registry. This process can be done directly (self-registration pattern) or via an intermediate component called "Registrar" (third-party registration pattern). In this platform, we use

the second pattern since it decouples existing microservices from the registration process. This helps us deploy platform-independent microservices that does not need to implement any registration logic to participate in our platform. More details about the "Registrar" is provided in Section VI-B. During service discovery, the service registry is queried by the discovery clients to find a matching profile with their data objects. The interaction between the client and the registry during the discovery process is intercepted by an API gateway.

The **API Gateway** provides a customized API to apply our communication strategy. It receives the discovery requests and contacts the dedicated registry to lookup for functionalities and microservices according to the client's objects.

The communication strategy describes two types of requests: "discovery requests" used to lookup services in the registry and "access requests" sent to benefit from discovered microservices. Any interaction between the client and the chosen microservices is direct.

The client initiates the service discovery by sending a request to the *service registry* ①. Second, the service registry filters the set of stored microservices and returns to the client a list containing the names of all the available functionalities that can be applied to this type of object ②. The client is in charge of selecting the most suited functionality according to his own objectives. He specify to the registry the chosen functionality as well as the details of his data object and quality requirements ③. These characteristics allow the registry to create a new list containing the full descriptions of all the existing microservices in the platform that can offer the desired functionality and support the client's object ④. At this point, when the client program receives the new list, it has discovered all the existing microservices and can contact the instance that seems most appropriate in terms of machine performance, network performance, requested parameters, etc. ⑤. During the discovery, microservices instances are replicated on run-time to meet the clients needs. Resources allocation are discussed later in Section VI.

C. Service discovery implementation

Service mesh tools address inter-services communication problems. They are fundamentally similar but differ in the implementation choices. For this reason, we aim to create a standalone communication strategy that is independent of the service mesh in use. Our implementation relies on REST APIs [31] to allow HTTP access. Fig. 1 presents the different steps of the discovery process.

As a client initiates the service discovery, the interactions in the system are as follows:

1) The client send a discovery request describing the object's characteristics within the query parameters.
   i. If no matching functionalities are present in the system, an HTTP response of status code `204` containing an empty response payload body is returned, and the discovery process stops.
   ii. If a match is found for this specific data object, an HTTP response of status code `200 OK` is returned to indicate that the request has been successfully completed. This response contains the list of existing functionalities, their names and descriptions.
2) After receiving the list of functionalities, the client need to select the desired one. This decision is made on the client side based on the specific objectives of each client.
3) A second query is sent to the service registry to discover the available microservices offering the chosen functionality. Within the query parameters, additional information concerning the client's data and his quality requirements are specified.
   i. If none of the available microservices offers the desired functionality or support the client's data, an HTTP response with status code `204` containing an empty list is returned.
   ii. If a match is found, the system returns the full registered data models of these microservices instances. The format of the response is presented in the Listing 2. Additional information might be added to the response, depending on the data type of the client's object. Among the instances details, the client receives the URL format to use in order to access these microservices.
4) On the client side, the discovery client will consider the response time of the returned microservices and other quality metrics to choose the most appropriate instance that meets the client's objectives.
5) Finally, using the returned self-linking URL, the client interacts directly with the chosen microservice instance.

In the next Section, we integrate this discovery process in a global data-driven architecture.

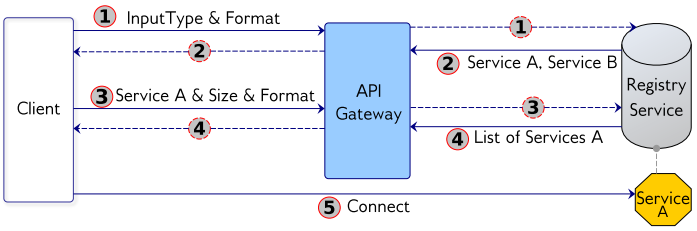

Fig. 1: Workflow of the data-driven service discovery initiated by the client to the API Gateway and service registry.

Listing 2: Response message associated to the discovery of a match. The properties describes details of the chosen functionality.

```
{
 "service": "microservice A",
 "description": "description of microservice A",
 "inputType": "type",
 "inputFormat": "format",
 "maxInput": "max objects by request",
 "maxInputSize": "max size in MB",
 "port": "port",
```

```
  "ip_address": "IP",
  "links":
  [ {
    "rel": "self",
    "href": http://ip_address:port/{URI}?params
  } ]
  "Response Time" : "",
  "User_defined": "",
}
```

V. Data-Driven microservices architecture

The service discovery process and its integration in a service mesh, relies on the interaction of several system components to manage the creation of microservices and the allocation of resources. As the complexity of service and infrastructure grows, there is a need to reduce the number of management services implicated in the discovery process and prevent them from causing potential degradation of system performance.

To this end, we propose a data-driven architecture with the following design goals: (i) A single purpose API Gateway specific for each type of data supported by existing microservices. This implementation design allows the management of services based on data. (ii) A zone management to allow clients to discover services in a specific geographical area, and balance loads between areas. (iii) A peer-to-peer model that creates an overlay network between the zones. This provides the ability to discover the resources deployed on several sites.

Communication between clients and microservices is implemented using two main models: General purpose API backend and Backend For Frontend (BFF). General purpose API backend provides a single entry point to backend services, while BFF introduces several entry points for each type of client. Using this model, the incoming load is shared among multiple customized gateways tailored to the needs of each client. This also reduce the possibility of bottleneck within these entry points.

Our architecture implements a customized BFF model. This model consists of creating entry points dedicated to each category of microservice. Each *BFF Gateway* is linked to a cluster of *service registry* to manage the descriptions of microservices that belong to the same data category. This cluster is only responsible for storing data models for microservices managed by this *BFF Gateway*, such as a cluster of *service registry image* managed by the *BFF API Gateway image* and independent of the registries of the other categories.

Wide distribution of data consumers and producers in architectures such as IoT systems, can lead to an increase in the system latency which affects the user experience. For this reason, we use in our system the concept of *Regions and Availability Zones* adopted by Amazon EC2 [32]. The Regions are designed to be completely isolated from each other to ensure the stability of our system, but the Availability Zones within a Region are connected together. The resources that belongs to the same geographical area are linked to the same *Availability Zone* within a *Region*. Each Zone has its own BFF backends and service registries. It contains a Zone Manager (ZM) component to manage incoming requests. This component, which represents the entry point of our architecture in each Zone, receives requests from clients located in its zone and determines to which *BFF Gateway* these requests should be forwarded. Once the chosen *BFF Gateway* has received the list of available microservices from its dedicated registry, it sends the results back to the ZM, which in turn passes them to the clients.

This architecture, presented in the Fig. 2, allows a data-driven management of deployed microservices. It reduces the number of transmitted messages in the system by forwarding the clients requests directly to the appropriate *BFF Gateway* and service registry cluster. Moreover, the BFF model adopted in this architecture aims at reducing the number of requests to be processed by each *API Gateway* that has become responsible for a single category instead of all deployed microservices. This reduces the bottlenecks within these components and as a result improves the system performance. In addition, this architecture avoids the continuous replication of microservices data models in the entire system by creating separate service registry clusters dedicated to each data type in every Zone of the Regions. Besides, this peer-to-peer (P2P) architecture between the different Zone Managers, creates an evolutionary system that allows to connect the entry points of all the Zones by forming a robust system for query processing. The next section addresses QoS guarantees, a prime concern to ensure service availability.

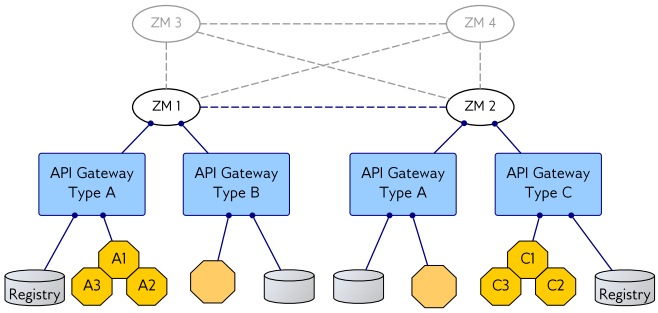

Fig. 2: The peer-to-peer model between the Zone Managers of the same Region for inter-zone connection.

VI. QoS approach

Linking Zones via a P2P network allow client programs to discover microservices located in any Zone of the Region. The support of various data types imposes management challenges to provide QoS while preventing any data type form taking over Zones resources. We propose an adaptation scheme that controls deployed microservices by data type as a Data-driven Microservices Groups (DMG). *DMG Image*, for example, represents all the microservices in the *Image* data category controlled by the *BFF API Gateway Image*.

This section presents components designed to support quality of service and their implementation in an existing service mesh, Istio.

A. Istio service mesh

Istio is an open service mesh built as a platform-independent software [6]. Istio offers a set of key features that allows to secure, observe, connect and control microservices across the service mesh network. It exposes deployed microservices to outside the platform via an Ingress Gateway component located at the Edge of the cluster.

Istio works natively with Kubernetes (k8s) [20]. The nodes of a k8s cluster run a set of application pods that encapsulate the microservices containers. Each microservice can have one or more instances called *replicas* accessible via the network using an abstraction called *service*. All existing services are distributed across one or multiple virtual clusters, called *Namespaces*, that create logically isolated environments.

In this work, we aim to create an enhancing microservices architecture using data-driven SD and QoS guarantees. With regard to this goal, Istio presents three major limitations.

First, microservices deployed in Istio are not immediately visible to the client programs outside the platform. To expose them, there is a need to manually create a set of routing rules linked to the Ingress Gateway. However, in our dynamic architecture where new microservices come and go, this is not efficient. Exposing available microservices to the external clients and hiding them when they are deleted, must be done automatically without any external intervention.

Second, when microservices have many *replicas*, Istio is responsible for balancing the incoming traffic between them within the same *Namespace*. Our architecture manages microservices by data type as DMG representing namespaces dedicated to each type. When multiple replicas of a DMG exist, we need to balance requests between the microservices instances replicated in different namespaces to ensure control over the load balancing feature, and prevent inconsistent workload distribution.

Last, Istio maintains an internal service registry containing the descriptions of deployed microservices running in the service mesh that have a specific service entry format. In addition, Istio offers a Consul adapter as an integrated registry. As described in Section IV, Consul provides its own service description format that does not support any additional properties. The two service registry presented above, use DNS for SD. This creates the need to modify Istio, and integrate an alternative service registry that supports our proposed data model without affecting the data-driven discovery process.

The realisation of these approaches to overcome the limitations of Istio, requires the integration of new components: *IngressController*, designed to automate the creation and deletion of routing rules ①. *LoadBalancer* to control sharing incoming requests among DMG replicas ②. And, the *ServiceRegistry* that accepts our data model and allow profiles matching for our data-driven discovery process ③.

B. QoS architecture

Figure 3 represents the new service mesh architecture that provides the missing functionalities and creates an adaptive system that take appropriate measures to maintain QoS.

This architecture utilizes the following components:

Registar Service is a customized third-party registration component that receives microservices data model at startup, registers them in the database and then deregisters the microservices at shutdown. This Registrar is the only component in the platform able to notify the management services when a new microservice is deployed or removed.

Many third-party registration components already exist to automate the registration process in containerized platforms such as Registrator [33] and Joyent [34]. However, they weren't useful in our architecture for the reason that they create services description based on the environment and does not support predefined data models. Also, they are designed to work with specific registries such as Consul and etcd [13] and do not support customized databases.

IngressController Service is a standalone service able to automatically update Istio's routing rules. It exposes two http endpoints to receive notifications from the *Registrar Service* when microservices are deployed or removed. It is able to add/delete routes or destinations for existing routes while considering Istio's weighted routing feature.

Loadbalancer Service is a standalone data-driven service designed to apply load balancing algorithms to share requests between DMG replicas of a specific data type. It uses by default a Round Robin algorithm and additional algorithms can be added as well. It exposes an http endpoint to receive incoming traffic from the Ingress Gateway and another to receive notifications from the *Registrar Service* when new microservices are added or deleted.

BFF API Gateway Service is a discovery service created for each data type supported in the platform. This component described in Section IV-B, sends the client's data type of each discovery request to the *DMGContoller Service* to verify if the request must be rejected or not.

DMGContoller Service is designed to receive the client's data type and check its dedicated DMG to decide whether any new request for this data type is allowed to proceed the discovery process. If this data type was overloaded, it triggers the *ScaleUP Service* to deal with the overload and demands the *BFF API Gateway Service* to reject the request. Otherwise, the discovery process proceeds normally.

ScaleUP Service is responsible for the "*ScaleUP*" algorithm, presented later in Section VI-C. It allows the system to adapt to the incoming requests by creating new DMG replica for overloaded data types while considering available resources. It exposes one http endpoint for receiving overload notifications and interacts with the service registries to add newly created DMG.

ScaleDOWN Service contains the "*ScaleDown*" algorithm, presented in Section VI-C. It is triggered on a given frequency to check if some DMG replicas are no more needed. If so, it removes deployed instances and free allocated resources.

C. QoS algorithms

The load in our system is always dynamically changing. This fickleness can give rise to excessive load that significantly

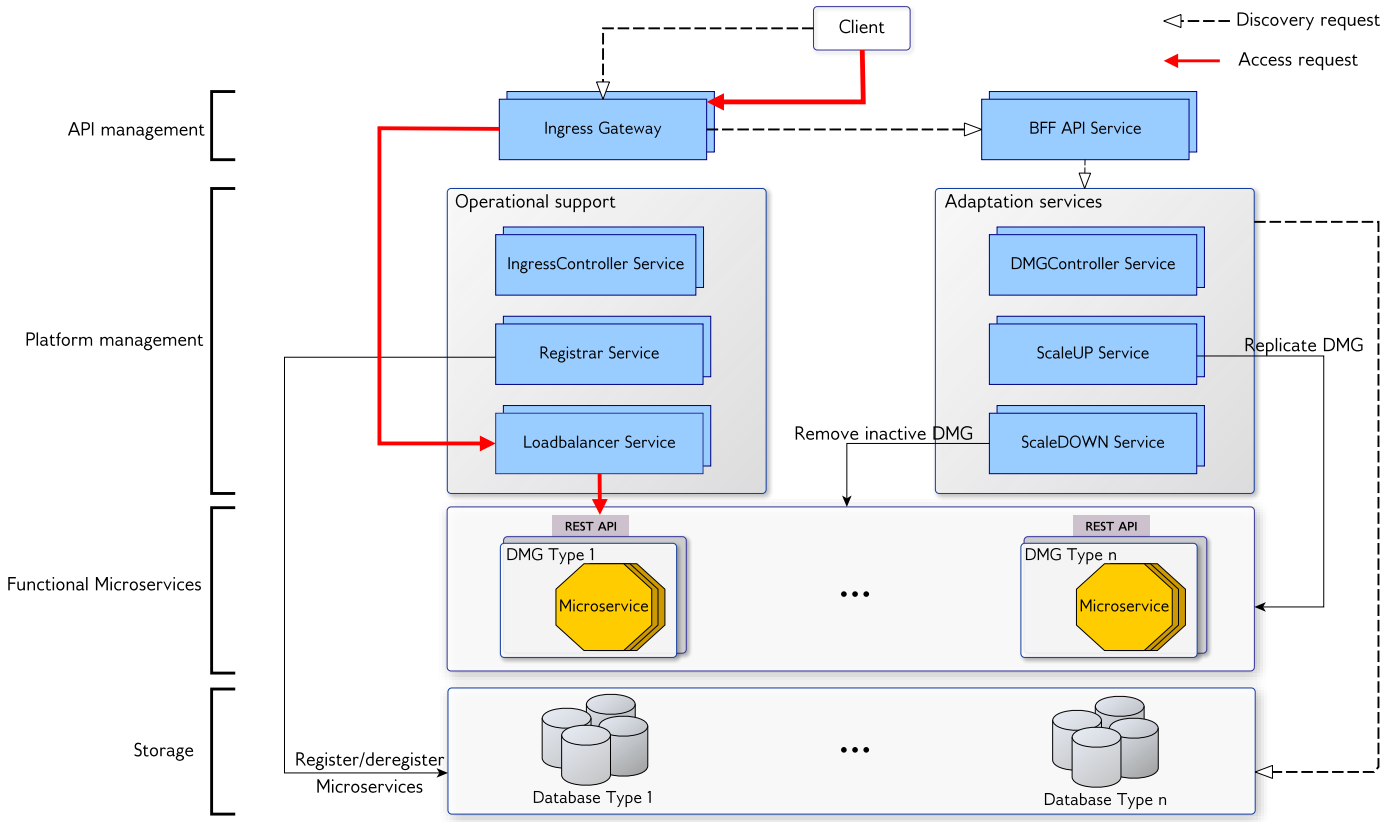

Fig. 3: Overview of the data-driven QoS architecture. It provides operational and adaptation support to control the discovery and access requests initiated by the clients.

slow down our processing system due to the extreme use of resources. Therefore, one way to avoid running out of resources is to set a limit to the maximum number of concurrent requests `maxreq` that microservices can process. However, our system contains microservices of different data types that share the Zone's resources. Thus, microservices of a specific DMG can sometimes receive a considerable number of requests. This can lead the system to reach its `maxreq` without allowing other DMG of different data types to process additional requests.

In this data-driven architecture, avoiding a specific data type from taking over Zone's resources is done by defining a rate limiter for each data type and applying resource quota (CPU, RAM) to every one of these types. On the other hand, the number and size of data arriving at the microservices is unknown in advance, the use of a requests rate limiter only can sometimes lead to a misuse of system resources. For example, with a limit rate equal to 10 concurrent requests, an application that accepts only images with dimension less than 2560x1600, will needs less resources to processes 10 HD grayscale images with dimension 1280x720, than 10 Widescreen RGB images with the maximum dimension.

As a consequence, we designed an adaptation scheme based on three QoS algorithms: ScaleUp, ScaleDown and Load Shedding. This scheme helps create a system able to handle traffic spikes with minimal performance degradation. In addition, it prevents rejecting requests when the rate limiter is reached but there are still enough resources in the platform. The details of the algorithms are presented below:

ScaleUp algorithm: it is responsible for adjusting the capacity of the system when an overload in detected. As described in Section VI-B, the component responsible for detecting the overload is the *DMGController Service*. For each discovery request, this service is notified to check whether an action should be done to prevent overload.

When a notification is received, *DMGController Service* queries the appropriate service registry (Algorithm 1). It verifies if the total number of running requests `RunnigReq` in all the DMG of this data type exceeds 85 percent of its rate limiter `maxreq`. If this threshold is not yet reached, the discovery request will be processed normally and the client will discover the existing functionalities. However, if it is reached, 2 options exist:① if the resource quota specified for this data type allows the reservation of additional system resources, the *DMGController Service* triggers the *ScaleUP Service* presented in Section VI-B to create a new DMG *replica*. At each creation of a new *replica*, the `maxreq` will be incremented. That increases the capacity of the data type to process more requests. ② However, if there are no more free resources for this data type sufficient to replicate a DMG, the algorithm hides first the overloaded DMG while ensuring

Algorithm 1: Adaptive system for processing requests

```
Data: dataType, dataFormat, quantity
Result: Discover functionalities depending on the data
Microservice discovery(dataType,
 dataFormat, quantity)
    begin
        HandleREQ ← NotifyDMG(dataType);
        if HandleREQ == TRUE then
            MSlist ← FindServices(dataType,
              dataFormat, quantity);
            return (MSlist);
        else
            return(NULL);
```

```
Data: dataType
Result: Modifying system capacity based on the load
boolean NotifyDMG(dataType)
    begin
        nbDMGtype ← countDMG(dataType);
        RunningReq ← countREQ(dataType);
        maxreq ← countMAX(dataType);
        if RunningReq > 85%× maxreq then
            if resourceavailable()==TRUE then
                DeployNewDMG(dataType);
            else
                if nbDMGtype > 1 then
                    DMGname ←
                     FindOveloadedDMG(dataType);
                    UpdateDMGstate(DMGname,
                     "OFF")
            return(FALSE);
        else
            return(TRUE);
```

Algorithm 2: Scale down underutilized DMG

```
Data: maxreq, DMGtype
Result: Remove the useless DMG
Microservice ScaleDownDMG(DMGtype)
    begin
        nbDMGtype ← countDMG(DMGtype);
        RunningReq ← countREQ(DMGtype);
        if RunningReq ≤ 50%× maxreq then
            if nbDMGtype > 1 then
                DMGmin ← selectDMGmin();
                UpdateDMGstate(DMGmin,"OFF");
        RemoveInactiveDMG(DMGtype);
```

that one *replica* remains, and then triggers the *LoadShedding* algorithm. Hiding a DMG *replica* prevents it from appearing to the next discovery requests and decreases the `maxreq` of the data type.

ScaleDown algorithm: hiding a DMG simply means switching its status to "OFF", but without actually removing the DMG and releasing the allocated resources. Thus, when the system hides multiple DMG due to excessive overload, the allocated resources become all reserved, which prevents the system from using these resources to deploy new DMG.

This algorithm, applied by the *ScaleDOWN Service* presented in Section VI-B, has two responsibilities: ① releasing resources reserved by the hidden DMG after they end their requests. ② removing the deployed DMG *replicas* that are no longer needed. As Algorithm 2 shows, hiding a DMG for a specific data type requires that the total number of running requests `RunnigReq` in all the DMG of this type be less than 50 percent of the total data type capacity. If this is the case, the DMG which has the fewest number of requests, at the time when the algorithm is triggered, will be hidden.

LoadShedding algorithm: this algorithm represents a rate limiting technique. It aims to shed some of the incoming load so that the system can at least continue to operate and provide services for a subset of requests, rather than crashing completely. In our QoS approach, we shed load in two cases: ① when the number of running requests for a specific data type exceeds the `maxreq` but there is no enough free resources (CPU, RAM) in the platform to deploy a new DMG *replica*. ② when a new request arrives to the system and the new deployed DMG is not yet ready to receive requests. In this case, the load shedding will continue to reject the incoming requests until the status of the newly deployed DMG switches to "ON" or the already existing DMG are no longer overloaded. The component responsible of applying this algorithm is the *DMGController Service*.

The use of these algorithms in the service mesh offers our system the ability to adapt itself to dynamic and heterogeneous load while effectively using the physical resources of the platform. In the following section, we present a couple of tests realised on our platform to show the behavior of the system to unexpected excessive load.

VII. Evaluation

In this section, we present the methodology and the results of a set of tests realised to evaluate the performance of our data-driven service discovery approach.

A. Methodology

The evaluation of the platform is performed on the large-scale platform Grid'5000 [35]. It represents a distributed testbed designed to support experimental-driven research in parallel and distributed systems. Our experimental setup contains 27 compute nodes of the dahu cluster. Each node is equipped with 2 x Intel Xeon Gold 6130 processors, 16 cores per CPU and 192 GiB memory.

We implemented and deploy the architecture presented in Fig. 3 that includes a service registry cluster with 3 nodes, 7 customized management services as well as Istio's components. The architecture contains one DMG *Image* containing

a set of microservices dedicated to graphics and image processing. This DMG reserves a limit of 13 CPU and 23 GB memory. It has a rate limiter equal to 80 concurrent requests to control incoming load.

First, we present the variation of the system Response Time (RT) and the percentage of accepted requests while sending a stable rate equals to 600 Requests per Second (RPS) during 20 min. Note that the percentage of accepted requests presents the percentage of discovery requests that were allowed to enter the platform after the system verified it's ability to process them.

Second we evaluate the system scalability by presenting the variation of the number of DMG *replicas* while sending the following load distribution: 50 RPS, then 100 RPS and lastly another 50 RPS. Each rate last for 5 min. The results of these tests are presented in the following subsection.

B. Results

Fig. 4 shows the variation of the average RT and the percentage of accepted requests for a stable incoming load. We can observe 3 different phases.

The first phase shows the baseline of 1 request per second. It presents a low average response time and a percentage of accepted request equal to 100%.

We aim to saturate the existing DMG by creating an increase of the incoming rate to 600 RPS in order to show the system behavior. When we switch from the baseline to this new rate, the second phase of the graph shows that the average RT begins to increase respectively until it reaches a peak of 25 seconds. On parallel, as the existing DMG alone is unable to process this incoming rate, the system starts shedding load. As it is shown in the graph, the percentage of accepted requests fall off rapidly until only 60% of incoming requests are allowed to enter the platform.

The drop of system performance, triggers the *DMGController Service* that detects a system overload and notifies the *ScaleUP Service* which in turn runs our ScaleUp algorithm. This algorithm replicates the targeted DMG to increase the system capacity and improve the performance. On parallel, the *DMGController Service* runs the LoadShedding algorithm to prevent requests from blocking the saturated DMG.

As requests continue to arrive, the final phase in the graph shows a decrease in the average RT. This decrease continues until it stabilizes around a value close to our baseline. At the same time, due to the creation of new DMG *replica*, the number of accepted requests rises respectively until it maintains its peak of 100%.

In the second experiment, we switch to a dynamic distribution of load rather than a stable incoming rate to focus on the number of DMG *replicas*.

Fig. 5 shows the variation in the number of DMG *replicas* while the incoming rate varies over time following a specific distribution of the load. We aim with this distribution to show the system capacity to adjust the number of DMG *replicas* according to a load that goes up and down respectively.

At first, since the unique DMG *replicas* targeted in this experiment is unable to process more than 80 concurrent

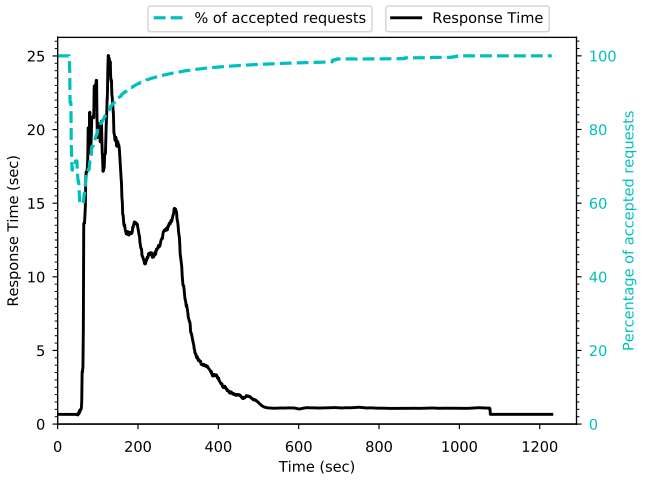

Fig. 4: With a stable incoming rate, the system's response time and the percentage of accepted requests stabilize around values close to the baseline due to DMG replications

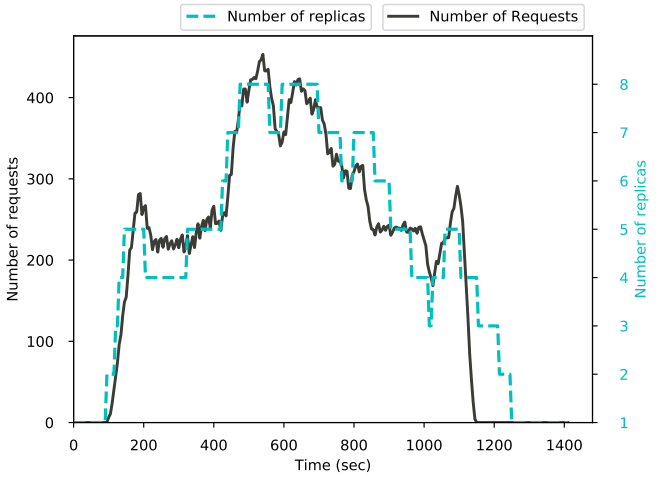

Fig. 5: With a dynamic incoming rate, the system tune the number of replicas according to the load.

requests, sending 50 requests each second leads to a system overload. This triggers the *DMGController Service* and then the *ScaleUP Service*, responsible for the ScaleUp algorithm, to create multiple new DMG *replicas*. As the graph shows, the number of *replicas* increased rapidly from 1 to 5 DMG *replicas*. Similarly, when the number of requests goes from 50 to 100 RPS, the number of *replicas* continues to increase slowly while following the variation of incoming rate.

Later, to show how the system reacts if the incoming rate suddenly falls off, we switched back the number of incoming requests from 100 to 50 RPS. As the incoming rate is lower than the capacity of the total number of *replicas*, the *ScaleDOWN Service* triggers the ScaleDown algorithm in order to free unnecessary DMG. As the graph shows, following this diminution, the number of *replicas* decreases respectively with a variation similar to the incoming rate.

VIII. Conclusion

Producers and consumers of data are growing continuously with different QoS requirements and data supported. Keep using Goal-based service discovery approaches that builds on specific identifiers to discover services locations, prevent the discovery of newly published services.

In this paper we presents a standalone data-driven service discovery framework that allow client programs to discover the available functionalities and microservices depending on their data objects, while providing a QoS guarantees. It builds on a data-centric models to allow matching between data products requirements and services. In addition, it uses a data-driven microservices architecture with a peer to peer network that enables a scalable service discovery with a possibility of integrating geographical features. This microservices architecture uses a service mesh Istio for inter-service communication management. It introduces new components to overcome the service mesh limitations that prevents the creation of a dynamic data-driven system able to take appropriate measures to maintain a QoS.

We have deployed this service mesh data-driven architecture on a real life testbed. Results show that the platform is able to adapt and maintain QoS in term of response time and percentage of accepted requests when receiving incoming rates that exceed system capacity. In addition, the system is effectively adapting itself to the incoming load by replicating a sufficient number of DMG, enough to process the incoming requests and removing allocated resources when they are no more needed. As part of our future research, we aim to extend our data-driven platform to introduce Edge-core placement of replicas based on data products requirements, and integrate programmable network services to the data model.

Acknowledgements

This research is supported in part by the NSF under grants numbers OAC 1640834, OAC 1835661, OAC 1835692 and OCE 1745246 and in other part by LIP Laboratory and ENS grants.

References

[1] Kleanthis Thramboulidis, Danai C. Vachtsevanou, and Alexandros Solanos. Cyber-physical microservices: An iot-based framework for manufacturing systems. *CoRR*, abs/1801.10340, 2018.

[2] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6. IEEE, 2016.

[3] Juan-Manuel Fernandez, Ivan Vidal, and Francisco Valera. Enabling the orchestration of iot slices through edge and cloud microservice platforms. *Sensors*, 19(13):2980, 2019.

[4] Darren Gallipeau and Sara Kudrle. Microservices: Building blocks to new workflows and virtualization. *SMPTE Motion Imaging Journal*, 127(4):21–31, 2018.

[5] Best service discovery software. https://www.g2.com/categories/service-discovery. Accessed 03/12/2020.

[6] Istio. https://istio.io/. Accessed 03/12/2020.

[7] Hashicorp consul. https://www.consul.io/mesh.html. Accessed 03/12/2020.

[8] Eduard Gibert Renart, Daniel Balouek-Thomert, and Manish Parashar. An edge-based framework for enabling data-driven pipelines for iot systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 885–894. IEEE, 2019.

[9] M. Ali, A. Anjum, M. U. Yaseen, A. R. Zamani, D. Balouek-Thomert, O. Rana, and M. Parashar. Edge enhanced deep learning system for large-scale video stream analytics. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10, May 2018.

[10] M. Krochmal S. Cheshire. Dns-based service discovery. https://tools.ietf.org/html/rfc6763, February 2013. Accessed 03/12/2020.

[11] Netflix. Eureka at a glance. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance, December 2014. Accessed 03/12/2020.

[12] Welcome to apache zookeeper™. https://zookeeper.apache.org/. Accessed 03/12/2020.

[13] etcd key-value store. https://etcd.io/. Accessed 03/12/2020.

[14] Mysql. https://www.mysql.com/. Accessed 03/12/2020.

[15] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225. IEEE, 2019.

[16] Linkerd 1 overview. https://linkerd.io/1/overview/. Accessed 03/12/2020.

[17] Linkerd 2 overview. https://linkerd.io/2/overview/. Accessed 03/12/2020.

[18] App mesh: Application-level networking for all your services. https://aws.amazon.com/app-mesh/. Accessed 03/12/2020.

[19] Envoy proxy. https://www.envoyproxy.io/. Accessed 03/12/2020.

[20] Sachin Manpathak. Kubernetes service mesh: A comparison of istio, linkerd and consul. https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/, October 2019. Accessed 03/12/2020.

[21] Christine Hall. What service meshes are, and why istio leads the pack. https://www.datacenterknowledge.com/open-source/what-service-meshes-are-and-why-istio-leads-pack, October 2019. Accessed 03/12/2020.

[22] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, May 2011.

[23] Angelika Musil, Juergen Musil, Danny Weyns, Tomas Bures, Henry Muccini, and Mohammad Sharaf. Patterns for self-adaptation in cyber-physical systems. In *Multi-disciplinary engineering for cyber-physical production systems*, pages 331–368. Springer, 2017.

[24] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36:41–50, 2003.

[25] Andreas Niederquell. Self-adaptive systems in organic computing: Strategies for self-improvement. *arXiv preprint arXiv:1808.03519*, 2018.

[26] Danny Weyns. Software engineering of self-adaptive systems: an organised tour and future challenges.

[27] Ahmad A Masoud. Decentralized self-organizing potential field-based control for individually motivated mobile agents in a cluttered environment: A vector-harmonic potential field approach. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 37(3):372–390, 2007.

[28] Chris Richardson. Pattern: Client-side service discovery. https://microservices.io/patterns/client-side-discovery.html. Accessed 03/12/2020.

[29] Chris Richardson. Pattern: Server-side service discovery. https://microservices.io/patterns/server-side-discovery.html. Accessed 03/12/2020.

[30] Consul data model. https://www.consul.io/docs/agent/services.html. Accessed 03/12/2020.

[31] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[32] What is amazon ec2? https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. Accessed 03/12/2020.

[33] Registrator. https://github.com/gliderlabs/registrator. Accessed 03/12/2020.

[34] Joyent. https://github.com/joyent/containerpilot. Accessed 03/12/2020.

[35] Desprez and al. Adding virtualization capabilities to the grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, pages 3–20, Cham, 2013. Springer International Publishing.