Politecnico di Torino

Master's Degree in Computer Engineering

Delivering Software Services In An Open Multi-Cloud Environment

Supervisors
Prof. Fulvio Risso
Alessandro Cannarella
Lorenzo Moro

Candidate
Francesco Pio Barletta

Academic Year 2022/2023

“Newton’s third law – the only way humans
have ever figured out of getting somewhere
is to leave something behind.”
- Cristopher Nolan, Interstellar

Abstract

Edge computing is a rapidly evolving field that leverages the vast and fragmented processing capacity at the network's edge to create a seamless and scalable computing continuum. The FLUIDOS project, a European initiative, aims to change how computer resources are used by developing an adaptable, expandable, secure, and decentralized operating system.

This thesis focuses on essential components within the FLUIDOS Node ecosystem, emphasizing their role in efficient communication and resource management. These components, including the Local Resource Manager, Discovery Manager, Available Resources, REAR Manager, Contract Manager, and Peering Candidates, form the core of FLUIDOS Nodes, operating collectively on Kubernetes clusters.

Additionally, the thesis delves into the development of the REAR Protocol, designed for secure data exchange of resources and capabilities among different cloud providers. It serves as a means to advertise resources, such as virtual machines with CPU and RAM specifications, capabilities like Kubernetes clusters, and, in the future, services such as database-as-a-service, to third parties.

This thesis contributes to the FLUIDOS project while providing valuable experience in edge computing. The insights and advancements obtained during this thesis aim to explore the potential of decentralized systems in the field of edge computing and innovative technology.

Contents

Listings

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Edge computing is a rapidly evolving field that capitalizes on the distributed processing capacity at the edge of the network, creating a seamless and scalable computing continuum. The **FLUIDOS** project, a European initiative, aims to revolutionize computing resource utilization by establishing homogeneous fabrics from edge processing capacity. This endeavor is rooted in the firm belief that a fluid cloud should be the community's response to counterbalancing the influence of hyperscalers, addressing concerns such as neutrality, lock-in, and flexibility. As part of this journey, TOP-IX, as an Internet Exchange Point (IX), represents the networking component that empowers this fluid cloud.

1.2 Research Objectives

This research centers on the development of pivotal components within the **FLUIDOS Node** ecosystem. The emphasis is on their pivotal role in facilitating effective communication and resource management. These components include the Local Resource Manager, Discovery Manager, Available Resources, REAR Manager, Contract Manager, and Peering Candidates. These components collectively constitute the foundation of **FLUIDOS Nodes**, operating seamlessly within Kubernetes clusters.

1.3 Scope and Methodology

The scope of this research encompasses the comprehensive exploration and implementation of the **REAR Protocol**. This protocol's primary goal is to establish secure data exchange between disparate cloud providers. It enables the advertisement of resources such as virtual machines, with specifications such as CPU and RAM, as well as capabilities like Kubernetes clusters. Furthermore, the protocol lays the groundwork for potential future inclusion of services, such as databases.

The methodology employed in this research entails leveraging cutting-edge technologies, prominently **Kubernetes**, to contribute to the advancement of the **FLUIDOS** project. This includes the development of key components, protocols, and systems. The research outcomes will play a pivotal role in shaping the trajectory of decentralized systems and offer an intriguing opportunity for individuals keen on exploring the realm of edge computing and innovative technology.

Chapter 2

Technologies

This chapter provides an overview of the pivotal technologies employed throughout the course of this research. The selection of appropriate technologies played a fundamental role in achieving the objectives and crafting effective solutions. The chapter delves into the utilization of Kubernetes, kubebuilder, and NATS, highlighting their significance in different phases of the project.

The realm of edge computing demands an arsenal of powerful tools that can orchestrate, communicate, and deliver seamless experiences. The subsequent sections provide insights into how each of these technologies was harnessed to tackle unique challenges and drive innovation in the FLUIDOS project.

2.1 Kubernetes

Kubernetes stands as a versatile, extensible, and open-source platform designed to manage containerized workloads and services, providing seamless support for both declarative configuration and automated processes. Its ecosystem is expansive and rapidly evolving, offering a wide array of Kubernetes services, tools, and support options.
Originating from Google's 2014 open-sourcing of the project, Kubernetes brings together more than 15 years of Google's expertise in handling large-scale production workloads with insights and best practices contributed by the community. The subsequent section offers a succinct introduction to Kubernetes, drawing inspiration from its official documentation. [1]

Although the FLUIDOS project assumes Kubernetes at its technological foundation, the overall architecture and most of the choices and PoC components developed in this project aim at having a more general breadth, hence potentially enabling their reuse with other technological substrates.

2.1.1 Architecture

At the core of Kubernetes lies the concept of a cluster, which serves as a user's initial encounter upon deploying the platform. A Kubernetes cluster encompasses a collection of worker machines known as nodes, responsible for executing containerized applications. Within every cluster, there exists at least one node designated as the master, fulfilling the role of hosting the control plane. This control plane effectively manages the worker nodes and the pods within the cluster. In production environments, the control plane often operates across multiple machines, while a cluster generally consists of multiple nodes, ensuring resilience and uninterrupted availability. On the other hand, worker nodes play host to the pods that collectively compose the application's workload.

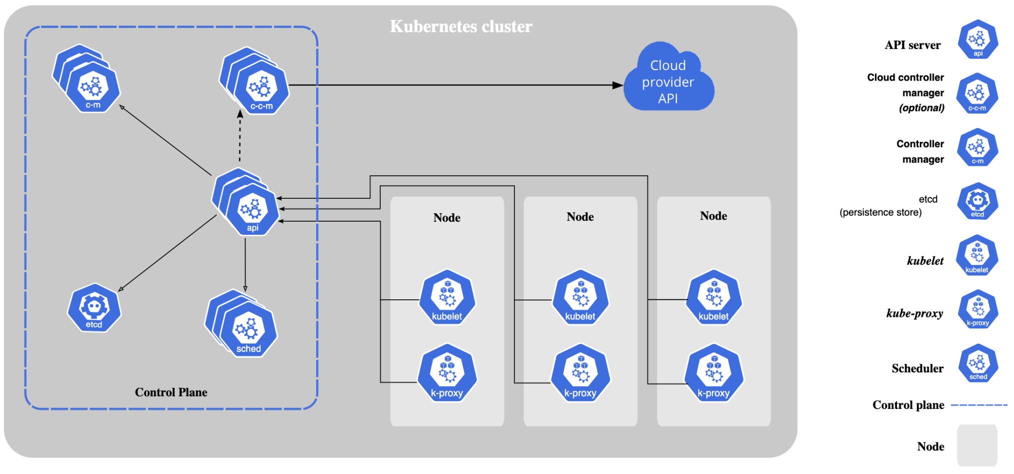

Figure 2.1: Kubernetes Architecture

2.1.2 Control Plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example,starting up a new pod). They can be run on any machine in the cluster. However, for simplicity, they are typically executed all together on the same machine, which do not run user containers.

API Server

The API server is a pivotal component within the Kubernetes control plane, serving as the gateway to the Kubernetes REST API. It functions as the interface that intercepts REST requests, validating and processing them. The primary implementation of the Kubernetes API server is known as kube-apiserver. Engineered for horizontal scalability, it grows by deploying additional instances, allowing straightforward redundancy by distributing traffic across multiple instances.

etcd

etcd emerges as a distributed, consistent, and highly available key-value store, acting as the foundational storage system for all cluster data in Kubernetes. Built on the Raft consensus algorithm [17], etcd empowers multiple machines to function cohesively as a unified group, even in the face of individual member failures. etcd can be embedded within the master node or positioned externally on dedicated hosts. It's crucial to note that solely the API server maintains communication with etcd.

Scheduler

A core control plane component, the scheduler orchestrates the instantiation of pods. Kubernetes includes a default scheduler, kube-scheduler, but customization is possible by adding new schedulers and specifying their use within pods. kube-scheduler monitors newly created pods without node assignments and assigns them to nodes. In decision-making, it factors in both individual and collective resource requisites, hardware/software/policy constraints, affinity and anti-affinity stipulations, data locality, inter-workload interference, and deadlines.

kube-controller-manager

The kube-controller-manager manages controller processes, continuously comparing the cluster's desired state (as specified by objects) with the current state (fetched from etcd). Conceptually, each controller operates as a distinct process. However, for simplicity, all controllers are amalgamated into a single binary, executing within a singular process. Included controllers consist of:

- **Node Controller**: Observes and reacts to node failures.
- **Replication Controller**: Ensures correct pod counts for replica objects.
- **Endpoints Controller**: Populates Endpoint objects linking Services and Pods.
- **Service Account and Token Controllers**: Generates default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component runs controllers that interact with the underlying cloud providers. The cloud-controller-manager binary is a beta feature introduced in Kubernetes 1.6. It only runs cloud-provider-specific controller loops. You can disable these controller loops in the kube-controller-manager. cloud-controller-manager allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to cloud-controller-manager while running Kubernetes. Some examples of controllers with cloud provider dependencies are:

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.
- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.1.3 Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

The kubelet is the primary node agent, responsible for instantiating pods and ensuring they maintain the desired state. It communicates with the API server to receive pod specifications and report the status of running pods. The kubelet also communicates with the container runtime to execute pod containers and handle pod lifecycle events. The kubelet runs on every node in the cluster.

kube-proxy

The kube-proxy is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes. These network rules allow network communication to the pods from network sessions inside or outside of the cluster. The kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

The container runtime is the software responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

2.1.4 Kubernetes Objects

Kubernetes objects are persistent entities representing the state of the cluster. They can be created, updated, and destroyed using the Kubernetes API. Kubernetes provides a number of object types that can be created by users or automatically created when handling certain tasks. The subsequent sections provide a brief overview of the most important Kubernetes objects.

Pods

A pod is the smallest and simplest Kubernetes object. A pod represents a running process on the cluster. A pod encapsulates an application's container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources. Pods are ephemeral, with a life expectancy equivalent to that of the application instance they represent. If a pod ceases to function, Kubernetes can create a replacement pod to maintain the desired state of the system. Here there is an example of the .yaml file used to create a pod:

Listing 2.1: Pod creation yaml file

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
    containers:
   - name: nginx
```

```
        image: nginx:1.14.2
        ports:
       - containerPort: 80
```

Services

A service is an abstraction that defines a logical set of pods and a policy by which to access them. Services enable a loose coupling between dependent pods. A service acts as a stable address for a set of pods, allowing them to be decoupled from the ephemeral nature of pods. Services are the abstraction that enables pods to die and replicate in Kubernetes without impacting your application. A service is a named load balancer that proxies traffic to one or more pods. Services enable a loose coupling between dependent pods. A service acts as a stable address for a set of pods, allowing them to be decoupled from the ephemeral nature of pods. Services are the abstraction that enables pods to die and replicate in Kubernetes without impacting your application. We can have different types of services:

- **ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default ServiceType.
- **NodePort**: Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- **LoadBalancer**: Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.
- **ExternalName**: Maps the service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

Here there is an example of the .yaml file used to create a service:

Listing 2.2: Service creation yaml file

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
```

```
spec:
    type: NodePort
    ports:
   - port: 80
        targetPort: 80
        nodePort: 30080
    selector:
        app: nginx
```

Deployments

A deployment is a Kubernetes object that manages a replicated application. A deployment ensures that a specified number of pod replicas are running at any given time. If a pod fails or is deleted, the deployment replaces it with a new pod. Deployments are the recommended way to manage the creation and scaling of pods. Deployments are the recommended way to manage the creation and scaling of pods. Deployments can be scaled up and down, and they can be updated with zero downtime. A deployment is a logical grouping of pods that are managed by Kubernetes. A deployment is similar to a pod, but it is more powerful and flexible. A deployment is a logical grouping of pods that are managed by Kubernetes. A deployment is similar to a pod, but it is more powerful and flexible. Here there is an example of the .yaml file used to create a deployment:

Listing 2.3: Deployment creation yaml file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
    replicas: 3
    selector:
        matchLabels:
        app: nginx
    template:
        metadata:
        labels:
            app: nginx
        spec:
        containers:
       - name: nginx
```

```
        image: nginx:1.14.2
        ports:
       - containerPort: 80
```

StatefulSets

A StatefulSet is a Kubernetes object that manages stateful applications. A StatefulSet maintains a sticky identity for each of its pods. These identities are useful for applications that require one or more of the following:

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.
- Ordered, automated rollbacks.

DaemonSets

A DaemonSet is a Kubernetes object that ensures that all (or some) nodes run a copy of a pod. As nodes are added to the cluster, pods are added to them. As nodes are removed from the cluster, those pods are garbage collected. Deleting a DaemonSet will clean up the pods it created. Deleting a DaemonSet will not delete previously created pods.

2.2 kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using custom resource definitions (CRDs). It provides tooling to simplify CRUD operations, webhook handling, and other tasks. [2]
Similar to web development frameworks such as Ruby on Rails and SpringBoot, Kubebuilder increases velocity and reduces the complexity managed by developers for rapidly building and publishing Kubernetes APIs in Go. It builds on top of the canonical techniques used to build the core Kubernetes APIs to provide simple abstractions that reduce boilerplate and toil. Kubebuilder does not exist as an example to copy-paste, but instead provides powerful libraries and tools to simplify building and publishing Kubernetes APIs from scratch. It provides a plugin architecture allowing users to take advantage of optional helpers and features.

Kubebuilder is developed on top of the **controller-runtime** and **controller-tools** libraries. Kubebuilder helps a developer in defining his Custom Resource, taking auto- matically basic decisions and writing a lot of boilerplate code. These are the main actions operated by Kubebuilder:

- **Create a new project**: Kubebuilder provides a command to create a new project, with a basic structure and a Makefile to build it.
- **Create a new API**: Kubebuilder provides a command to create a new API, with a basic structure and a Makefile to build it.
- **Create a new controller**: Kubebuilder provides a command to create a new controller, with a basic structure and a Makefile to build it.
- **Create a new webhook**: Kubebuilder provides a command to create a new webhook, with a basic structure and a Makefile to build it.

2.2.1 controller-runtime

The controller-runtime library provides a set of tools to simplify building Kubernetes controllers. It provides a set of commonly needed implementations of the **controller-runtime/pkg/reconcile.Reconciler** interface, as well as utilities to make building your own Reconcilers simpler. It also provides a set of utilities for testing your controllers in a Kubernetes cluster-independent way. [3]

2.2.2 controller-tools

The controller-tools library provides a set of utilities for working with Kubernetes-style API objects. It provides a set of utilities for generating code to work with Kubernetes-style API objects, as well as a set of utilities for converting between different API versions of the same object.

2.3 Docker

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and are thus more lightweight than virtual machines. Containers are created from images that specify their precise contents. Images are often created by combining and modifying standard images downloaded from public repositories. [4]

2.3.1 Dockerfile

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession. This page describes the commands you can use in a Dockerfile. [5]

Here there is an example of a Dockerfile:

Listing 2.4: Dockerfile example

```
FROM python:3.8-slim
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r req.txt
EXPOSE 80
ENV NAME World
CMD ["python", "app.py"]
```

2.3.2 Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. [6]

2.4 Liqo

Liqo is an open-source project that supports the vision of liquid computing described above. It extends Kubernetes to enable the creation of dynamic and seam-less multi-cluster topologies, regardless of underlying infrastructure limitations.Liqo achieves this without introducing modifications to standard Kubernetes APIsfor application deployment, making it compatible with a wide range of common in-frastructures and cluster types, with no restrictions on networking configurations. [7]

2.5 NATS

This technology was used just in the **first release** of the project, to implement the communication between the different components of the FLUIDOS Node. NATS is a lightweight, high-performance cloud native messaging system that implements a publish-subscribe model. It is written in the Go programming language and is built on top of the NATS platform. NATS is a Cloud Native Computing Foundation (CNCF) incubating project. [8]

2.5.1 NATS Client Applications

Developers use one of the NATS client libraries in their application code to allow them to publish, subscribe, request and reply between instances of the application or between completely separate applications. Those applications are generally referred to as 'client applications' or sometimes just as 'clients' throughout this manual (since from the point of view of the NATS server, they are clients).

2.5.2 NATS Service Infrastructure

The NATS services are provided by one or more NATS server processes that are configured to interconnect with each other and provide a NATS service infrastructure. The NATS service infrastructure can scale from a single NATS server process running on an end device (the nats-server process is less than 20 MB in size!) all the way to a public global super-cluster of many clusters spanning all major cloud providers and all regions of the world such as Synadia's NGS.

2.5.3 Connecting NATS Client applications to the NATS servers

To connect a NATS client application with a NATS service, and then subscribe or publish messages to subjects, it only needs to be configured with:

- URL(s): A 'NATS URL'. This is a string (in a URL format) that specifies the IP address and port where the NATS server(s) can be reached, and what kind of connection to establish (plain TCP, TLS, or Websocket).
- Authentication (if needed): Authentication details for the application to identify itself with the NATS server(s). NATS supports multiple authentication schemes (username/password, decentralized JWT, token, TLS certificates and Nkey with challenge).
- Subject(s): A 'NATS subject'. This is a string that identifies the subject that the application wants to publish messages to or subscribe messages from.

2.5.4 Simple messaging design

NATS makes it easy for applications to communicate by sending and receiving messages. These messages are addressed and identified by subject strings, and do not depend on network location. Data is encoded and framed as a message and sent by a publisher. The message is received, decoded, and processed by one or more subscribers.

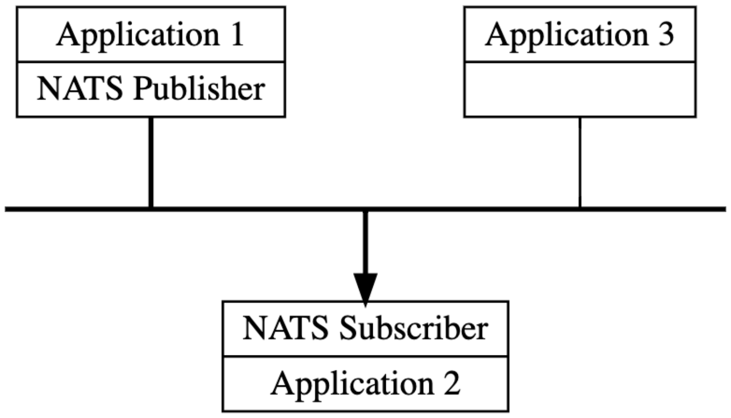

Figure 2.2: NATS Messaging Design

With this simple design, NATS lets programs share common message-handling code, isolate resources and interdependencies, and scale by easily handling an increase in message volume, whether those are service requests or stream data.

Chapter 3

State Of the Art for Resources and Services Exchange Protocols

3.1 Introduction

The exchange of resources and services between clients and providers plays an important role in various domains, ranging from cloud computing to edge computing and beyond. To ensure efficient resource utilization, it is important to establish robust protocols that facilitate the interaction between clients seeking specific resources and providers offering them. This chapter presents a comprehensive review of the state of the art in resource exchange protocols, focusing on the context of customer-provider interactions.

3.2 Communication Protocols

In this section, we will explore the commonly used communication protocols for exchanging information between clients and providers. Effective communication is essential in various domains, ranging from web applications to distributed systems. Understanding the strengths, limitations, and use cases of different communication protocols is crucial for designing efficient and reliable systems.

We will examine several widely adopted protocols, including HTTP, REST, gRPC, and MQTT, and analyze their key features, architectural styles, message formats, and integration possibilities.

3.2.1 Hypertext Transfer Protocol (HTTP)

HTTP is a widely used protocol for communication over the web. It operates on a client-server model, where clients send requests to servers and receive responses.

HTTP is designed to enable the exchange of various resources, including HTML pages, images, documents, and data. Its stateless nature, request-response structure, and support for methods like GET, POST, PUT, and DELETE make it suitable for resource retrieval, submission, modification, and deletion.

In a resource exchange workflow using HTTP, clients can send requests to servers to retrieve or upload resources. For example, a client can use an HTTP GET request to retrieve data from a server, while an HTTP POST request can be used to upload new resources. This workflow is commonly employed in web applications, file transfers, and RESTful APIs.

3.2.2 gRPC

gRPC is a high-performance, open-source framework developed by Google. It enables efficient communication and data exchange between distributed systems. gRPC is based on the Remote Procedure Call (RPC) paradigm, where clients can invoke methods on remote servers as if they were local. It uses the Protocol Buffers (protobuf) language for defining services and message formats.

In a resource exchange workflow using gRPC, clients can define service methods that specify the desired resource interactions, such as retrieving or updating data. These methods are exposed by servers, allowing clients to make remote procedure calls. gRPC's efficient binary serialization and support for streaming make it suitable for scenarios requiring high-performance communication, such as microservices architectures and inter-service communication in cloud environments.

3.2.3 MQTT

MQTT is a lightweight publish-subscribe messaging protocol designed for constrained devices and low-bandwidth networks. It provides a simple, efficient, and reliable communication mechanism. MQTT follows a publish-subscribe pattern, where clients can publish messages to topics, and other clients can subscribe to those topics to receive the messages.

In a resource exchange workflow using MQTT, clients can publish resource updates or notifications to specific topics. Other interested clients can subscribe to these topics to receive the published messages. This pattern is useful in scenarios where resource status updates or event-driven communication is required, such as IoT (Internet of Things) applications, real-time data streams, and sensor networks.

3.2.4 REST

REST is an architectural style that provides a set of constraints for designing networked applications. It emphasizes a resource-centric approach and leverages existing web standards, including HTTP, to enable communication between clients and servers. RESTful APIs use standard HTTP methods and adhere to principles like statelessness, uniform resource identification, and self-descriptive messages.

In a resource exchange workflow using REST, clients interact with resources exposed by servers through HTTP methods. For example, clients can use an HTTP GET request to retrieve resource representations, an HTTP POST request to create new resources, and HTTP PUT or DELETE requests to update or delete existing resources. RESTful APIs are widely used in web services, mobile applications, and distributed systems for resource exchange and integration.

3.3 Real Use Cases: Resource Acquisition Workflows

In this chapter, we will explore real use cases that demonstrate the resource acquisition workflows in action. By examining examples from various industries, such as Booking.com, we can gain insights into how these organizations manage the process of acquiring resources effectively and efficiently. Through these real-world scenarios, we will uncover the underlying workflows and understand how different platforms and frameworks facilitate the resource acquisition process.

3.3.1 Booking.com Connectivity APIs

The Booking.com Connectivity APIs enable to send and retrieve data for properties listed on Booking.com. It is possibile to manage room availability, reservations, prices, and many other things [9].

The Booking.com Connectivity APIs offer a number of specialised functions, divided into these categories:

- **Content**: Create properties, rooms, rates, and policies, and link this information together for the Booking.com website.
- **Rates and Availability**: Load inventory counts, rates, and price availability restrictions (for specific room-rate combinations), per date and/or date range combination.
- **Reservations**: Retrieve reservations, modifications, and cancellations made on Booking.com.
- **Promotions**: Create special promotions for certain date ranges and booker types.

- **Reporting**: Report credit card problems, changes to reservations after check-in, and no-shows.

In addition to the specialised APIs, we also have a set of supporting APIs for retrieving general Booking.com system information, such as accepted currency codes and room names.

Reservations APIs

A **reservation** represents the booking of one or more room nights at a property. Each reservation is a unique booking created by a guest using the Booking.com channels. Reservations API keeps you updated on your bookings by sending a sequence of messages, also known as reservation messages.

The messages are classified as new booking confirmation, modification to an existing booking, or cancellation. Regardless of the category, the reservations API provides the data in a common format. A reservation may include several units of rooms, apartments or villas. Each reservation or booking is specific to exactly one property.

To process reservations, Booking.com provides two sets of endpoints using the following two specifications:

- **OTA XML specifications** (OTA_HotelResNotif e OTA_HotelResModifyNotif): A complete and fault-tolerant reservations processing solution following the specification from the OpenTravel Alliance (OTA). Use this solution to retrieve and acknowledge processing the reservations.
- **B.XML specifications** (/reservations): A simple and light-weight solution to retrieve reservations following Booking.com's XML specifications. Use this solution to retrieve the property reservations. Acknowledging that you successfully processed the reservation is currently not supported with this solution.

Here, two different examples using B.XML specifications:

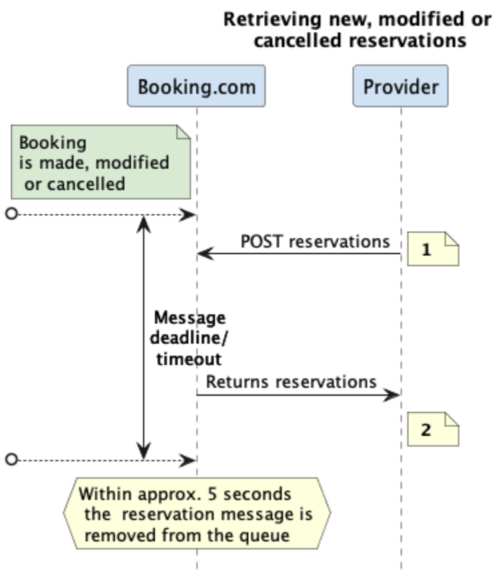
Retrieving new, modified or cancelled reservations
Booking.com
Provider
Booking is made, modified or cancelled
POST reservations
1
Message deadline/ timeout
Returns reservations
2
Within approx. 5 seconds the reservation message is removed from the queue

Figure 3.1: Retrieveing new, modified or cancelled reservations

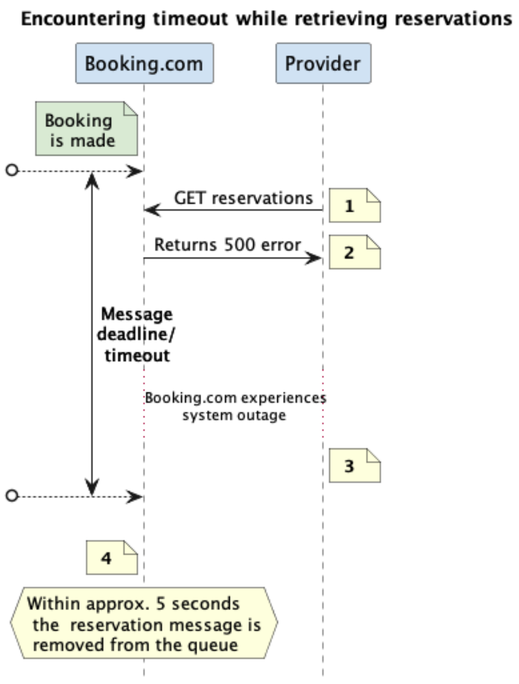
Encountering timeout while retrieving reservations
Booking.com
Provider
Booking is made
GET reservations
1
Returns 500 error
2
Message deadline/ timeout
Booking.com experiences system outage
3
4
Within approx. 5 seconds the reservation message is removed from the queue

Figure 3.2: Encountering timeout while retrieving reservations

3.3.2 Ticketmaster

Ticketmaster is a globally recognized ticketing platform that revolutionized the way people purchase tickets for various events, including concerts, sports games, and theatrical performances. With its user-friendly interface and extensive event catalog, Ticketmaster has become a go-to destination for millions of customers worldwide.

The ticket acquisition workflow on Ticketmaster follows several key steps to ensure a seamless and efficient ticket purchasing process for customers:

- **Event Discovery**: Customers begin by browsing Ticketmaster's website or mobile app to explore upcoming events in their area. They can search by event type, artist, venue, or date to find the desired event.
- **Ticket Selection**: Once customers find the event they are interested in, they can select the specific tickets they want to purchase. Ticketmaster offers various ticket options, including different seating sections, price ranges, and quantities.
- **Seat Allocation**: After selecting tickets, the system allocates seats based on the customer's preferences and availability. Ticketmaster's seat selection algorithm ensures that seats are assigned in the most optimal way to accommodate the customer's group and provide an enjoyable experience.
- **Checkout Process**: Customers proceed to the checkout page, where they review their ticket selection, enter their payment and billing information, and complete the transaction. Ticketmaster supports multiple payment methods, including credit cards, digital wallets, and other secure payment options.
- **Order Confirmation**: Once the purchase is completed, customers receive an order confirmation that includes details such as the event name, date, time, seating information, and a unique order ID. This confirmation serves as proof of purchase and is often sent via email or can be accessed through the customer's Ticketmaster account.

Partner APIs

The Ticketmaster Partner API lets clients reserve, purchase, and retreive ticket and event information [10].

If a user abandons a page/tab after a ticket reserve has been made, client applications should do their best to detect this and issue a DELETE /cart request to free up allocated resources on the ticketing server. This should also be done if client apps no longer want to wait through a long, continuing polling process.

This is necessary since ticket reserve requests that result in polling will eventually complete asynchronously and take up resources even if clients do not consume the next polling url.

It is possible to use the different APIs to define the workflow for searching and purchasing a ticket:

- `GET /discovery/v2/events`: find events and filter your search by location, date, availability, and much more.
- `POST /partners/v1/events/{event_id}/cart?apikey={apikey}`: reserves the specified tickets. For integrations requiring captcha, send the captcha solution token in the json body. A **hold time** will be returned in the cart response that will indicate, in seconds, how long the cart is available for. This value may increase if the user moves through the cart process.
- `GET /partners/v1/events/{event_id}/...`: get shipping options available for this event. Note: some API users will be pre-configured for certain shipping options and may not need to perform this. Specifying the "region" query parameter will return options available for users in the selected country. Using the value 'ALL' will return all options.
- `PUT /partners/v1/events/{event_id}/...`: add a shipping option to the event. Note: some API users will be pre-configured for certain shipping options and may not need to perform this.
- `PUT /partners/v1/events/{event_id}/cart/payment`: add customer and billing information to the order.
- `PUT /partners/v1/events/{event_id}/cart?apikey={apikey}`: finalize the purchase and commit the transaction.

Here, an example of workflow to purchase a ticket for a certain event:

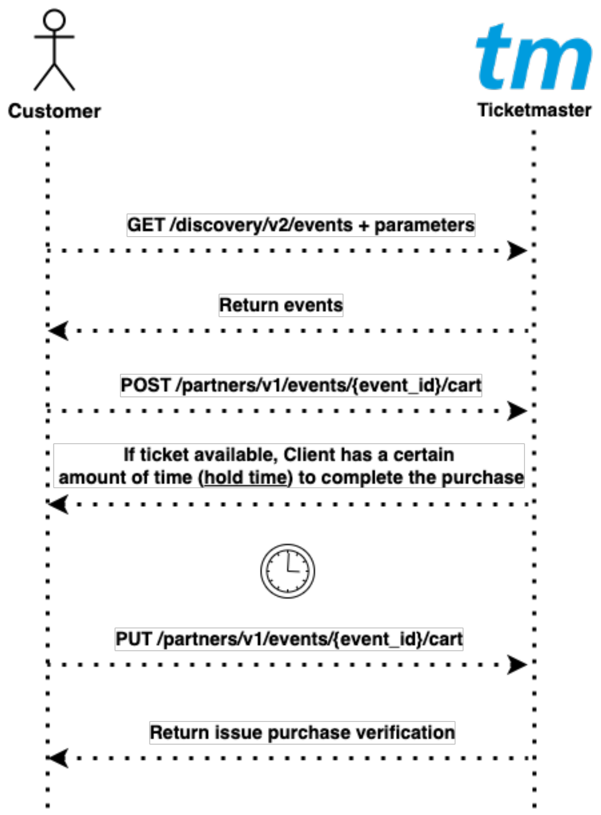
Customer
tm
Ticketmaster
GET /discovery/v2/events + parameters
Return events
POST /partners/v1/events/{event_id}/cart
If ticket available, Client has a certain amount of time (hold time) to complete the purchase
PUT /partners/v1/events/{event_id}/cart
Return issue purchase verification

Figure 3.3: Search for an event and buy a ticket

3.3.3 Research Solution

Reservation protocols are an essential communication mechanism in many areas, which ensure fair and efficient resource allocation in shared environments. These protocols are commonly used in distributed systems, networking, and multi-user applications to prevent conflicts and coordinate access to critical resources.

One of the most adopted reservation protocol in computer networks is RSVP (Resource Reservation Protocol [11]). Its primary goal is to establish and manage resource reservations for data transmission, and it is mainly used in Quality of Service (QoS) enabled networks to ensure the efficient and reliable delivery of data traffic. However, one of the main limitations of RSVP is its limited scalability, because, as the number of participants and the complexity of the network increase, managing and maintaining reservations can become challenging. This is because RSVP operates in a soft-state manner, which requires the continuous refreshing of reservations, preventing its adoption when a huge amount of (tiny) reservations are required and in case of mobile hosts, in which the reservation (which requires the detailed knowledge of the location of the host) is being made by mobile hosts. To overcome such limitation MRSVP [12] has been proposed, allowing mobile devices to perform reservations not only for the current location, but also for future locations. In addition, [16] extends the problem formulation, including the price of networking resources, so that the network service provider can communicate the availability of services and delivers price quotations and charging information to the user, and the user requests or re-negotiates services with desired specifications for one or more flows.

As an extension of RSVP, RSVP-TE (Resource Reservation Protocol - Traffic Engineering) is designed to support traffic engineering capabilities in computer networks. It enables the establishment of explicit paths for data traffic, allowing network administrators to control the flow of traffic and optimize network resources. RSVP-TE has thus been proposed in combination with MPLS to perform path signaling in a wide area network [13, 14].

In our perspective, RSVP and similar solutions target only network parameters, failing to include the multi-dimensionality of the computing resources (e.g., reserve CPU, RAM, etc.).

Authors in [15] present the Service Negotiation and Acquisition Protocol (SNAP) as a means to enable communication and negotiation between different entities in a distributed system, such as clients and servers. The protocol aims to establish agreements on the expected quality of service (QoS) that clients require and that servers can provide. In the attempt to extend the flexibility of the SLA negotiation mechanism, [17] proposes a bilateral protocol for SLA negotiation using the alternate offers mechanism wherein a party is able to respond to an offer by modifying some of its terms to generate a counteroffer. Finally, authors in [20] also describe a brokering architecture that is able to make advance resource reservations and create SLAs using the WS-Agreement standard [18], based on the Contract Net protocol for negotiating SLAs [19].

Recently, also telco Operators in the 5G era have a significant opportunity to monetize the capabilities of their networks. This paradigm change led to additional requirements for the Edge infrastructure [21], and to the definition of a suitable protocol to allow seamless application deployment across different Telco providers [22]. Specifically, this interface enables also the federation between Operator Platforms, sharing of edge nodes, and access to Platform capabilities while customers are roaming. The above technical capabilities are leveraged to provide the same software services associated with the customer also when it is connected to a foreign operator, thanks to the capability to deploy containerized application in the visited Operator Platform. Although promising, the current proposal (i) does not include a discovery mechanism to allow the members of the federation to share the price of computing resources or services, (ii) it does not support highly dynamic environments in which the roaming occurs with unforeseen operators (a previously established agreement must be already in place before the roaming), and (iii) is not able to guarantee the property of generality when describing the offered resources/services, but focuses only on containerized applications.

Chapter 4

FLUIDOS: Architecture and Components

FLUIDOS (Flexible, scaLable, secUre, and decentralIseD Operating System) aims to leverage the enormous, unused processing capacity at the edge, scattered across heterogeneous edge devices that struggle to integrate with each other and to coherently form a seamless computing continuum. [23]

FLUIDOS overcomes the above limitation by enabling the creation of a virtual computing space spanning across multiple physical domains, hence enabling a service (which has been started in the virtual space) to leverage all the resources belonging to the same virtual domain, independently from their physical location. Hence, in FLUIDOS a service can seamlessly scale based upon the availability of resources within the entire virtual infrastructure, e.g., ending up having one instance running in the telco edge, and another in the cloud datacenter, hence blurring the current rigid cluster boundaries.

4.1 Technology Substrate

FLUIDOS starts with a strong (and potentially controversial) assumption: the chosen reference technology is Kubernetes. This stems from several considerations:

- **Cloud-native**: Kubernetes is considered the most promising platform to provide cloud- native services, which provide unprecedented agility and efficiency compared to the previous world of VMs.
- **Scalability**: Kubernetes is designed to scale applications across many servers in a cluster, allowing them to handle large traffic loads without downtime.

- **Portability**: Kubernetes is platform-agnostic, meaning it can be used on-premises or in the cloud, and supports multiple cloud providers, such as AWS, Google Cloud Platform, and Azure.
- **Large-to-Small scale support**: multiple flavors of Kubernetes exist, with support for both large deployments (e.g., cloud datacenter), and small-scale deployments (e.g., even resource-constrained individual devices), hence becoming the natural candidate to build the META-OS concept upon.
- **Flexibility**: Kubernetes provides a range of features to manage containerized applications, such as scaling, deployment, updates, and monitoring. It also supports various container runtimes, including Docker and CRI-O.
- **Resource Optimization**: Kubernetes can optimize resource utilization by automatically allocating containers based on available resources and workload demands.
- **Community Support**: Kubernetes has a large and active open-source community that contributes to its development and maintains a vast ecosystem of add-ons and tools.

Nevertheless, although the FLUIDOS project assumes Kubernetes at its technological foundation, the overall architecture and most of the choices and proof-of-concept components developed in this project aim at having a more general breadth, hence potentially enabling their reuse with other technological substrates.

4.2 Main characteristics

Main characteristics of FLUIDOS are:

- **Intent-driven**: a consumer can assign to each workload the desired execution constraints through high-level policies, without knowing about the infrastructural details. Overall, liquid computing brings the cattle service model to a greater scale.
- **Decentralized architecture**: the resource continuum stems from a peer-to-peer approach, with no central point of control and management entities, as well as no intrinsically privileged members. Following a decentralized and peer-based model like the Internet, the liquid computing approach fosters the coexistence of multiple actors, including larger cloud providers, smaller, territory-linked enterprises, and even small office/homeowners

- **Multi-ownership**: each actor maintains full control of his own infrastructure while deciding at any time how many resources and services to share and with whom. Although single clusters are expected to be under the control of a single entity, the entire resource ocean would likely span across different administrative domains
- **Fluid topology**: members can join and leave the virtual continuum at any time, independently from their infrastructure size, from enterprise-grade data centers to IoT and personal devices.

4.3 Architecture

FLUIDOS architecture is made up of two main objects, very different from eachother still greatly coupled in several workflows: the **Node** and the **Catalog**. While the first is mantatory, as it's the base element of the ecosystem, the latter is optional, and jumps in whenever there's the need to go cross-domain or let the general public access the ecosystem through a user interface. [24]

4.3.1 Node

A **FLUIDOS Node** is a unique computing environment, under the control of a single administrative entity (although different Nodes can be under the control of different administrative entities), composed of one or more machines and modeled with a common, extensible set of primitives that hide the underlying details (e.g., the physical topology), while maintaining the possibility to export the most significant distinctive features (e.g., the availability of specific services; peculiar HW capabilities).

Overall, A FLUIDOS node is orchestrated by a single Kubernetes control plane, and it can be composed of either a single device or a set of devices (e.g., a datacenter). Device homogeneity is desired in order to simplify the management (physical servers can be considered all equals, since they feature a similar amount of hardware resources), but it is not requested within a FLUIDOS node. In other words, a FLUIDOS node corresponds to a Kubernetes cluster.

A FLUIDOS node includes a set of resources (e.g., computing, storage, networking, accelerators), software services (e.g., ready-to-go applications) that can be either leveraged locally or shared with other nodes. Furthermore, a FLUIDOS node features autonomous orchestration capabilities, i.e., (1) it accepts workload requests, (2) it runs the requested jobs on the administered resources (e.g., the participating servers), if application requirements and system security policies are satisfied, and (3) it features a homogeneous set of policies when interacting with other nodes.

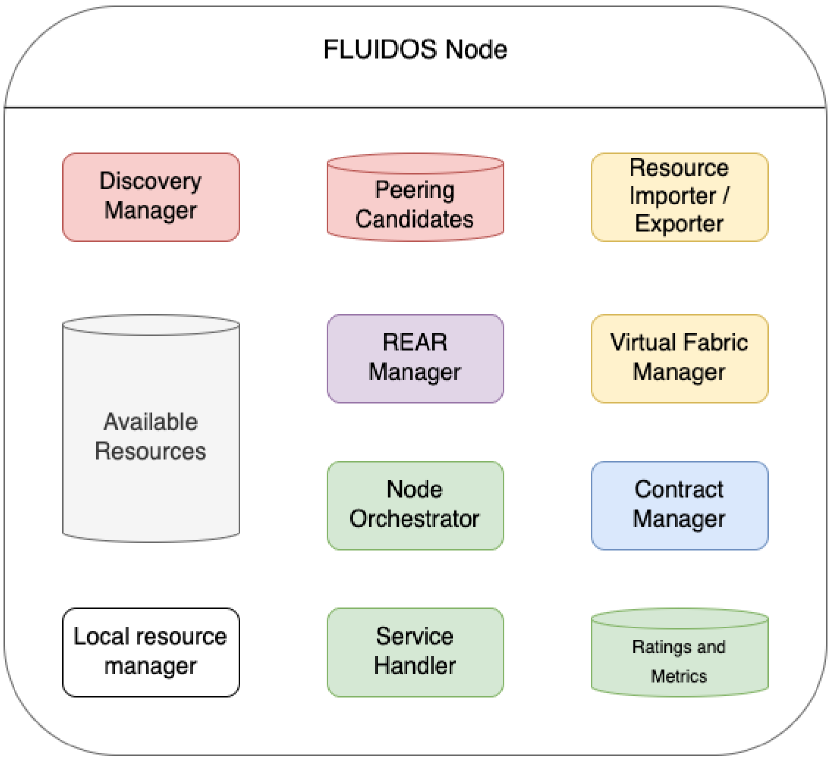

Figure 4.1: FLUIDOS Node architecture

4.3.2 Supernode

A **FLUIDOS Supernode** acts as a gateway for domain's nodes that do not have access to the Internet or they don't know other domains. Its behavior is mostly similar to the node's one, with the main difference being the knowledge of other domains or catalogs with wich it can interact. The interactions exploit the same protocols and are managed by the same components, but the sources and destinations of the informations differ.

Besides acting as a gateway, a Supernode is another aggregation point in the distribution chain of informations. The specific tasks are deepened in the descriptive document of each component, while the interactions are shown in the workflows involving multiple domains.

4.3.3 Catalog

A **FLUIDOS Catalog** is a set of services that can be used to interact with the FLUIDOS ecosystem. It is composed of a set of components that can be used to interact with the ecosystem, and it's the only way to interact with the ecosystem from the outside. The components are:

- Connector-UI
- Connector
- Broker

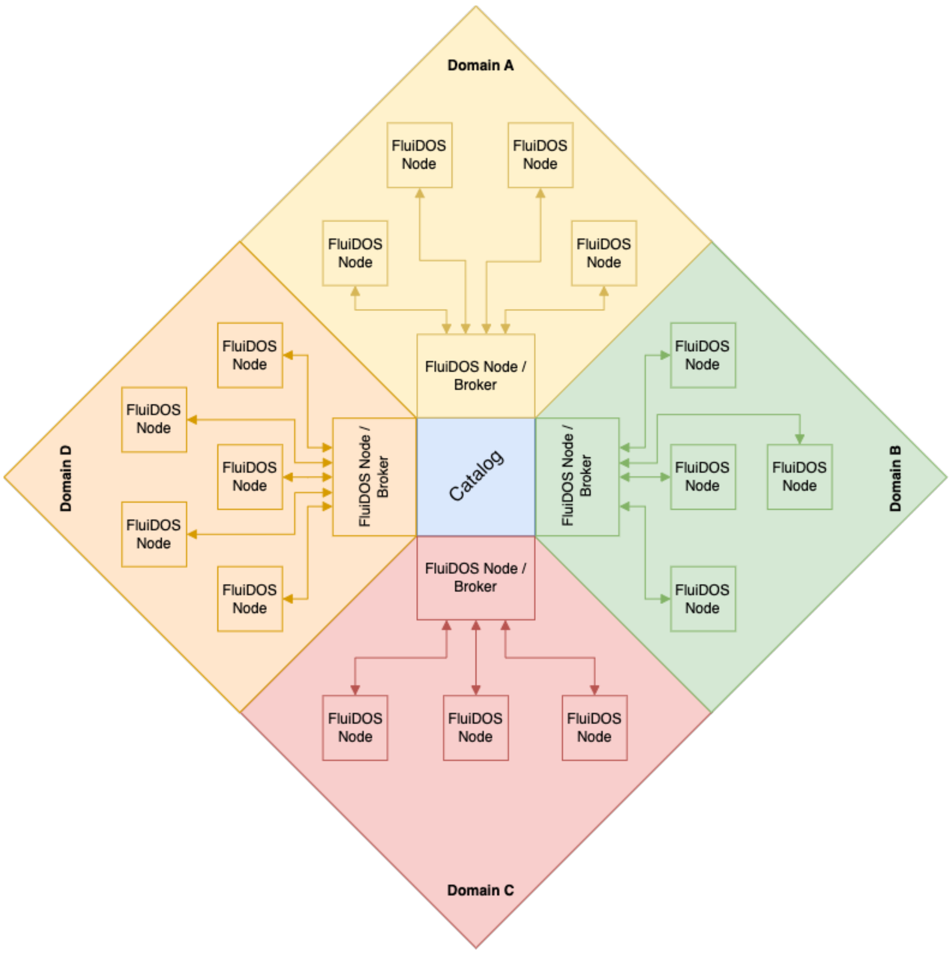

Figure 4.2: FLUIDOS Catalog interactions

4.4 Interactions

The interactions among Nodes, Nodes and Supernodes, Supernodes and Catalogs, and all software components belonging to these objects, can be grouped in five phases:

1. **Intent management**: this phase is carried out by the Service Handler. It consists in the translation and the reduction to lowest terms of a received intent so that the Node Orchestrator can trigger the primitive functions of the Node to fulfill that intent.

2. **Discovery**: this phase consists in the discovery, both solicited or advertised, of flavours provided by other Nodes. It is mainly carried out by the Discovery Manager with the scope of filling the Peering Candidates table and provide suitable flavours to the REAR Manager. It relies on first couple of messages of the REAR protocol.

3. **Reservation**: this phase involves many components, since it consists in the contract signing step, managed by the Contract Managers of both Nodes, and the actual resources reservation step, that consists in updating of the Available Resources tables. The phase is syncronously coordinated by the REAR Managers of both Nodes.

4. **Peering**: the Peering phase is carried out by the Virtual Fabric Managers of both Nodes: it basically relies on the primitives provided by LIQO to establish the peering between the two nodes and extend the continuum, exploiting virtual routers created by the Network Managers to ensure node-to-node reachability.

5. **Usage**: this phase is managed by the Node Orchestrator of the Consumer Node and consists in the actual deployment of the workload on the Kubernetes Virtual Nodes created as output of the Peering phase, so that the real workload is automatically moved to the Provider Node for execution. This is the phase where the two Telemetry Services are working actively to monitor both the Local and Remote performances to populate the Ratings and Metrics table.

6. **Tear down**: This is the phase where the process is sunsetted, the peering is teared down, and the resources are freed and made available again for future processes. Its triggered by the Node Orchestrator and involves all the components that previously have cooperated to take the whole process up.

The phases described make use of a set of inter-process communication methods and protocols, such as the **REAR Protocol**.

4.5 Workflows

In this section are described the workflows of the main interactions among the components of the FLUIDOS ecosystem.

4.5.1 Two nodes in the same domain

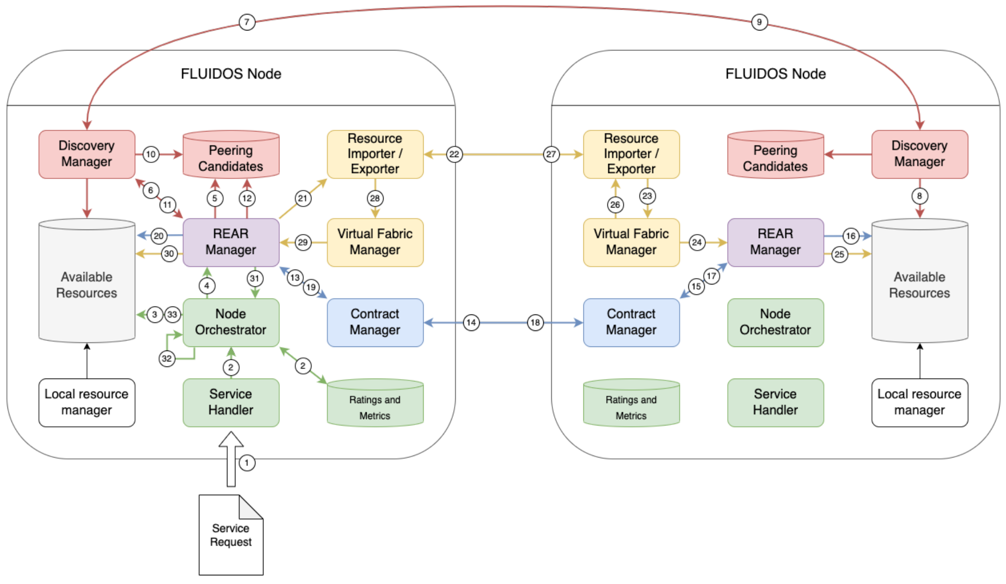

Figure 4.3: Worflow of two nodes in the same domain

1. A new **service request** is sent to the **Service Handler** in the form of an **intent**.
2. The **intent** is passed to the **Node Orchestrator**, which translates and decomposes it into simple **resources or services requests**.
3. The **Node Orchestrator** looks up for **flavours** matching the **request** in the **Available Resources** and the **Ratings and Metrics** tables.
4. In case there are no suitable available resources, the **request** is passed to the **REAR Manager** to start the Discovery, Reservation, and Peering phases.
5. The **REAR Manager** looks up in the **Peering Candidates** table for potential peering candidates (suitable **flavours**) matching the **request**.
6. In case there are no suitable peering candidates, the **REAR Manager** builds a **flavour selector** and passes it to the **Discovery Manager**, asking for a suitable peering candidate.
7. The **Discovery Manager** of the Consumer Node sends a **LIST_FLAVORS** message to all the endpoints it already knows, whether they are local Nodes, a Supernode, or a Catalog.

8. The **Discovery Manager** of the Provider Node looks-up for suitable local **flavours** matching the received **flavour selector** in its **Available Resources** table.

9. In case one or more suitable **flavours** are found, the **Discovery Manager** of the Provider Node sends back an OK message to the **Discovery Manager** of the Consumer Node, attaching the **flavours** list.

10. The **Discovery Manager** of the Consumer Node populates the **Peering Candidates** table with the newly discovered **flavours**.

11. It then informs the **REAR Manager** that the Discovery phase has completed successfully.

12. See Step 5.

13. Now that a suitable peering candidate is found, the **REAR Manager** asks the **Contract Manager** to start the Reservation phase, pointing to the Provider Node.

14. The **Contract Manager** of the Consumer Node sends a **RESERVE_FLAVOUR** message to the **Contract Manager** of the Provider Node.

15. The **Contract Manager** of the Provider Node creates a new **contract** and asks its **REAR Manager** to reserve resources for the incoming request.

16. The **REAR Manager** updates the **Available Resources** table by deleting the old advertised **flavour** and creating a new reduced **flavour** (suitable for other future requests) and by creating a new **allocation** of type "Node" in "inactive" status.

17. The **REAR Manager** informs the **Contract Manager** that the resources have been reserved.

18. The **Contract Manager** of the Provider Node answers back to the **Contract Manager** of the Consumer Node with an **OK** message. At the same time, an informer makes the **Network Manager** aware, so that it can deploy a virtual router to enable peering reachability.

    Following the steps defined in the REAR Protocol, the two **Contract Managers** may exchange a couple of other messages to confirm the reservation on both sides, but may also directly agree following the procedure described from Step 14 to Step 18.

19. The **Contract Manager** of the Consumer Node creates a new **contract** and informs its **REAR Manager** that the resources have been reserved. At the same time, an informer makes the **Network Manager** aware, so that it can deploy a virtual router to enable peering reachability.

20. The **REAR Manager** updates the **Available Resources** table by creating a new **allocation** of type "VirtualNode" in "inactive" status.

    Now the flow changes a little bit according to the release of Liqo. The current available release provides an "External Resource Monitor" to intercept incoming peering requests so to allocate resources based on the CustomerID available in the request. This mechanism has a clear limitation: only a single peering can be established per each couple of Nodes, since the discriminator is the CustomerID of the cluster. Instead, with the next release of Liqo, which will provide a declarative way of defining Virtual Nodes, this limitation will be overcome.

21. The **REAR Manager** solicits its **Virtual Fabric Manager** (LIQO) to set up the peering.

22. The **Virtual Fabric Manager** of the Consumer Node generates a ResourceRequest on the **Virtual Fabric Manager** of the Provider Node. Network reachability is provided by the **Network Managers**.

23. The **Virtual Fabric Manager** of the Provider Node looks up in its **Contract Manager** for an already signed contract embedding a **flavour** matching the CustomerID of the Consumer Node: once found, it completes the peering.

24. Once done, an informer makes the **REAR Manager** aware.

25. The **REAR Manager** updates the **Available Resources** table by putting the previously created **allocation** to "active" status.

26. The **Virtual Fabric Manager** of the Provider Node sends a **ResourceOffer** to the **Virtual Fabric Manager** of the Consumer Node.

27. See Step 24.

28. See Step 25.

29. The **REAR Manager** finally informs the **Node Orchestrator** that all the Discovery, Reservation, and Peering phases are over.

30. The **Node Orchestrator** is auto-solicited to continue with the Intent Management phase.

31. See Step 3.

Now that there are available resources that fulfill the request, the **Node Orchestrator** can finally deploy the workload described in the **intent** exploiting standard Kubernetes primitives and pointing to the **VirtualNode** retrieved in the **Available Resources** table.

4.5.2 Two nodes in different domains (w/o Catalog)

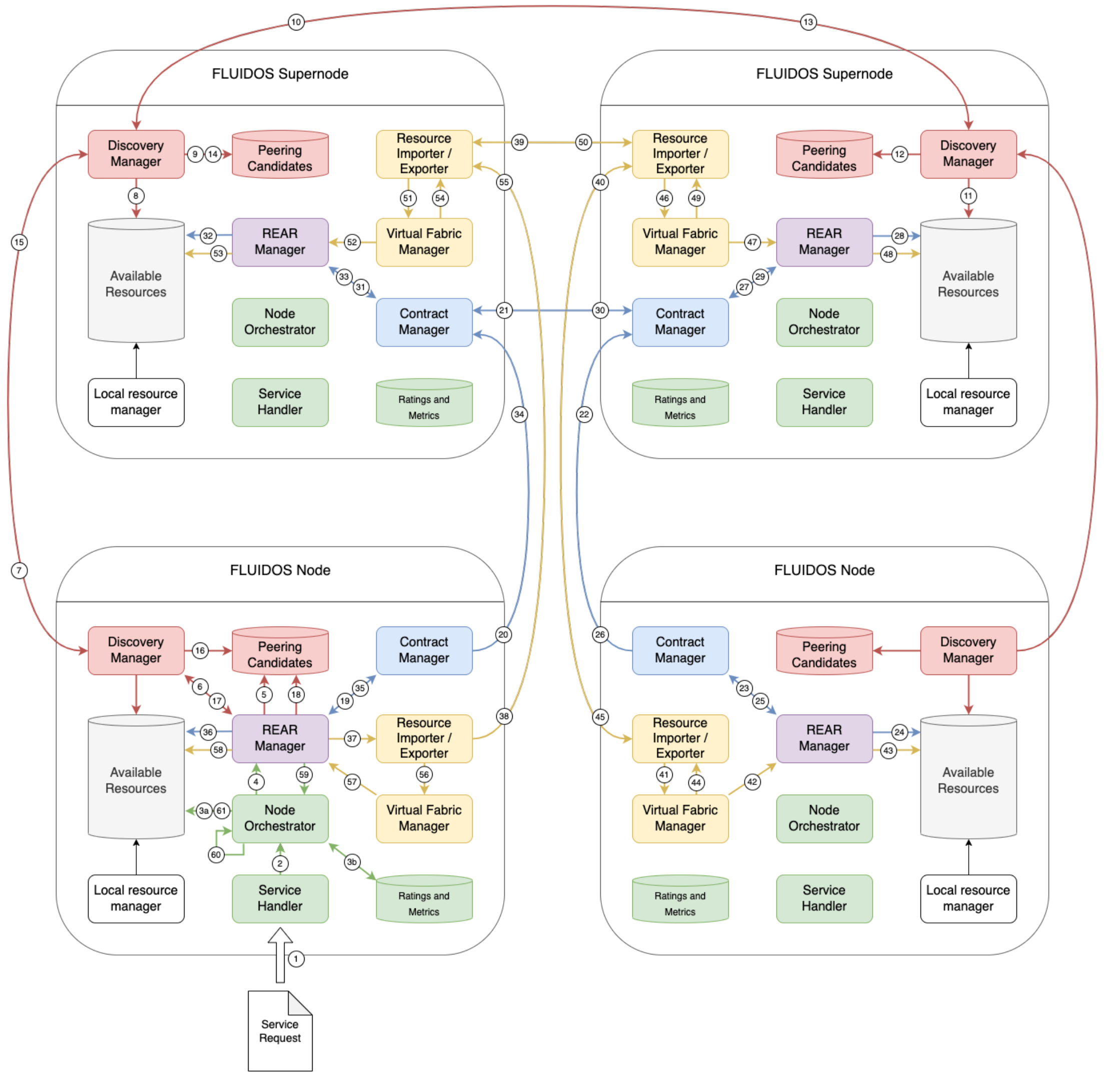

Figure 4.4: Worflow of two nodes in different domains (w/o Catalog)

1. A new **service request** is sent to the **Service Handler** in the form of an **intent**.
2. The **intent** is passed to the **Node Orchestrator**, which translates and decomposes it into simple **resources or services requests**.
3. The **Node Orchestrator** looks up for **flavours** matching the **request** in the **Available Resources** and the **Ratings and Metrics** tables.

4. In case there are no suitable available resources, the **request** is passed to the **REAR Manager** to start-up the Discovery, Reservation, and Peering phases.
5. The **REAR Manager** looks up in the **Peering Candidates** table for potential peering candidates (suitable **flavours**) matching the **request**.
6. In case there are no suitable peering candidates, the **REAR Manager** builds a **flavour selector** and passes it to the **Discovery Manager**, asking for a suitable peering candidate.
7. The **Discovery Manager** of the Node sends a **LIST_FLAVOURS** message to its Supernode, already known and directly reachable.
8. The **Discovery Manager** of the Supernode looks-up for suitable local **flavours** matching the received **flavour selector** in its **Available Resources** table.
9. In case no suitable **flavours** are found, the **Discovery Manager** looks up in the **Peering Candidates** table for potential peering candidates (suitable **flavours**) matching the **request**.
10. In case no suitable peering candidates are found, the **Discovery Manager** of the Consumer Supernode sends a **LIST_FLAVOURS** message to all the endpoints it already knows, whether they are other Supernodes or a Catalog, attaching the **flavour selector**.
11. The **Discovery Manager** of the Provider Supernode looks-up for suitable local **flavours** matching the received **flavour selector** in its **Available Resources** table.
12. See Step 9.
13. In case one or more suitable **flavours** are found, the **Discovery Manager** of the Provider Supernode sends back an OK message to the **Discovery Manager** of the Consumer Supernode, attaching the **flavours** list.
14. The **Discovery Manager** of the Consumer Supernode populates the **Peering Candidates** table with the newly discovered **flavours**.
15. The **Discovery Manager** of the Consumer Supernode sends back an OK message to the **Discovery Manager** of the Consumer Node attaching the **flavours** list.
16. The **Discovery Manager** of the Consumer Node populates the **Peering Candidates** table with the newly discovered **flavours**.

17. It then informs the **REAR Manager** that the Discovery phase has completed successfully.

18. See Step 5.

19. Now that a suitable peering candidate is found, the **REAR Manager** asks the **Contract Manager** to start the Reservation phase pointing to the Provider Node.

20. The **Contract Manager** of the Consumer Node sends a message **RESERVE_FLAVOUR** to the **Contract Manager** of the Consumer Supernode.

21. The **Contract Manager** of the Consumer Supernode sends a message **RESERVE_FLAVOUR** to the **Contract Manager** of the Provider Supernode.

22. The **Contract Manager** of the Provider Supernode sends a message **RESERVE_FLAVOUR** to the **Contract Manager** of the Provider Node.

23. The **Contract Manager** of the Provider Node creates a new **contract** and asks its **REAR Manager** to reserve resources for the incoming request.

24. The **REAR Manager** updates the **Available Resources** table by deleting the old advertised **flavour** and creating a new reduced **flavour** (suitable for other future requests) and by creating a new **allocation** of type "Node" in "inactive" status.

25. The **REAR Manager** informs the **Contract Manager** that the resources have been reserved.

26. The **Contract Manager** of the Provider Node answers back to the **Contract Manager** of the Provider Supernode with an **OK** message. At the same time, an informer makes the **Network Manager** aware, so that it can deploy a virtual router to enable peering reachability.

    Following the steps defined in the REAR Protocol, the couples of **Contract Managers** may exchange a couple of other messages to confirm the reservation on both sides, but may also directly agree following the procedure described from StepX to StepY.

27. The **Contract Manager** of the Provider Supernode creates a new **contract** and answers back to the **Contract Manager** of the Consumer Supernode with an **OK** message. At the same time, an informer makes the **Network**

Manager aware, so that it can deploy a virtual router to enable peering reachability.

28. The **Contract Manager** of the Consumer Supernode creates a new **contract** and answers back to the **Contract Manager** of the Consumer Node with an **OK** message. At the same time, an informer makes the **Network Manager** aware, so that it can deploy a virtual router to enable peering reachability.

29. The **Contract Manager** of the Consumer Node creates a new **contract** and informs its **REAR Manager** that the resources have been reserved. At the same time, an informer makes the **Network Manager** aware, so that it can deploy a virtual router to enable peering reachability.

30. The **REAR Manager** updates the **Available Resources** table by creating a new **allocation** of type "VirtualNode" in "inactive" status.

31. The **REAR Manager** solicits its **Virtual Fabric Manager** (LIQO) to set up the peering.

32. The **Virtual Fabric Manager** of the Consumer Node generates a ResourceRequest on the **Virtual Fabric Manager** of the Provider Node. Network reachability is provided by the **Network Managers**.

33. The **Virtual Fabric Manager** of the Provider Node looks up in its **Contract Manager** for an already signed contract embedding a **flavour** matching the CustomerID of the Consumer Node: once found, it completes the peering.

34. Once done, an informer makes the **REAR Manager** aware.

35. The **REAR Manager** updates the **Available Resources** table by putting the previously created **allocation** to "active" status.

36. The **Virtual Fabric Manager** of the Provider Node sends a **ResourceOffer** to the **Virtual Fabric Manager** of the Consumer Node.

37. See Step 34.

38. See Step 35.

39. The **REAR Manager** finally informs the **Node Orchestrator** that all the Discovery, Reservation, and Peering phases are over.

40. The **Node Orchestrator** is auto-solicited to continue with the Intent Management phase.

41. See Step 3.

42. Now that there are available resources that fulfill the request, the **Node Orchestrator** can finally deploy the workload described in the **intent** exploiting standard Kubernetes primitives and pointing to the **VirtualNode** retrieved in the **Available Resources** table.

Chapter 5

The REAR Protocol

The **REsource Advertisement and Reservation** (REAR) protocol aims at providing secure data exchange of resources and capabilities between different cloud providers. It can be used to advertise resources (e.g., virtual machines and their characteristics in terms of CPU, RAM), capabilities (e.g., Kubernetes clusters) and (in future) services (e.g., a database as a server) to any third party, enabling potential customers to know what is available in other clusters, and possibly (automatically) establish the technical steps that enables the customer to connect and consume the resources/services agreed in the negotiation phase. [25]

There are two main types of entity involved, which are providers and customers:

- **Providers** advertise their resources and services in a standardized format.
- **Customers** explore and find resources according to their specific criteria.

Overall, REAR seamlessly integrates with established resource management systems and platforms. This protocol accommodates diverse resource types and allows for future expansions.

REAR has been designed with a focus on **generality**. This is because it allows to perform resource exchange for (possibly) any type of resources and services, ranging from traditional VMs, Kubernetes clusters, services (e.g., DBs), and sensors and actuators (e.g., humidity and temperature sensors).

5.1 REAR messages

REAR defines a set of messages that facilitate the client/provider interaction for the purchase of available computing resources or services. At its core, REAR has been designed with a focus on generality (i.e., able to be general enough to describe a huge variety of computing and/or service instances). The figure below depicts a possible interaction between a customer and a provider using the REAR protocol.

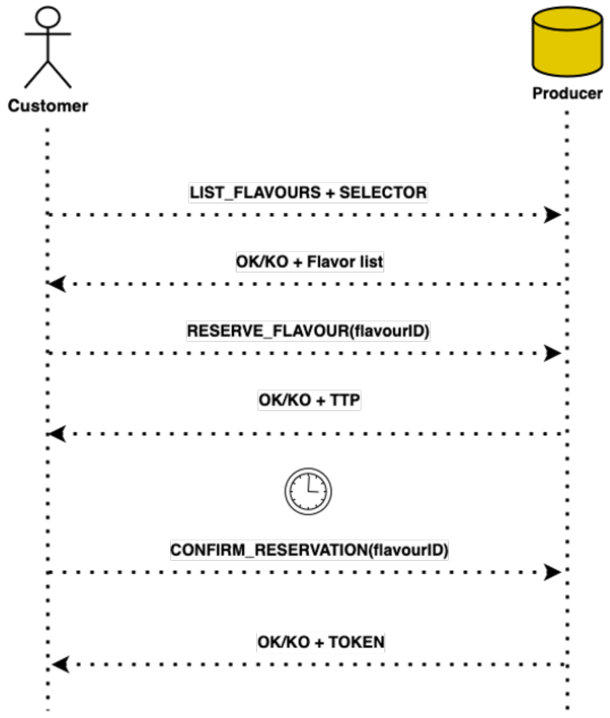

Figure 5.1: Interaction between client and provider using the required messages

This section describes the main interaction enabled by the REAR protocol, whereas the details of the different APIs will be provided in the following chapter.

5.1.1 Get the list of available flavours

The list flavour message provides the client with the list of available flavours offered by a given producer. Using a standardized selector, a client can request the list of available flavours matching specific needs, like a given amount of computing resources (e.g., CPU, RAM, storage), the flavour type (e.g., VM, Kubernetes cluster, DB service), and additional policies (e.g., maximum price).

If properly formatted, the list flavour message returns the list of available flavours offered by a given producer (if any). Specifically, each item in the list will have the following key information:

- **flavour ID**: Each offer should be identified by a unique flavour ID instead of just the name.

- **Provider ID**: Associate the flavour with the corresponding Provider ID.
- **Type**: Specify the type of the flavour (e.g., VM/K8s Cluster/etc.).
- **Characteristics**: Specify the capacities and resources provided by the flavour (CPU, RAM, etc.).
- **Policy**: Specify if the flavour is aggregatable/partitionable
- **Owner**: represents the entity that owns the flavour (FQDN/unknown). It can correspond to the Provider ID of the flavour.
- **Price or Fee**: If applicable, specify the price or fee associated with the flavour.
- **Expiration Time**: It represents the duration after which the flavour needs to be refreshed. If the flavour is not refreshed within the Expiration Time, it becomes invalid or expires. The Expiration Time can be calculated by adding a specific timestamp to the current time, indicating the number of hours or days until expiration.
- **Optional Fields**: Other details such as limitations, promotions, availability etc., can be included as optional information.

Note that if the producer does not have available flavours, or does not have flavours matching the provided selector, it may return an empty list. The interaction is always initiated by the client and can be summarized as follows:

- The client wants to retrieve the list of available favors offered by a provider.
- The client creates the selector using one of the standardized ones based on the requirements.
- After the message is ready, an HTTP GET is sent to the provider to get the list of filtered flavours.
- The provider returns the list of matching flavours.
- If the provider does not have available flavours, or does not have flavours matching the specified selector, an empty list will be returned.

Policy

If the **partitionable** field is available, it indicates that the Flavour can be divided or partitioned into smaller units. However, it is also specified the minimum amount of CPU and RAM that must be present for the Flavour (e.g., if CPU must be at least one, the CPU cannot be "partitioned" below that unit). If the field is false, client has no possibilities to divide the Flavour. Additionally, a step value is defined, which determines the increment between valid quantities for CPU and RAM. For example, if the step value for CPU is 1, users can request CPU quantities such as 2, 3, or 4, but not decimal values like 1.4 or 2.6. The step value ensures that CPU and RAM quantities align with the defined increments and maintain consistency within the Flavour's specifications.

When the **aggregatable** field is available, it means that multiple instances of the same Flavour can be combined or aggregated together. This enables the pooling of resources to meet higher demands or optimize resource utilization. The mincount field specifies the minimum number of Flavours that must be aggregated if "aggregatable" is true. If the field is false, client can choose that single instance (e.g., a single VM instead of a set of VM).

5.1.2 Reserve flavour

The **reserve flavour message** is sent by the client to the provider to *notify the intention of reserving an offered flavour*. It is the first step that requires handling concurrency in client requests, as different clients may be interested in the same flavour. Note that this message only notifies the provider of the intention to purchase a flavour. The request must then be finalized using the *confirm purchase message* (see the following subsection).

Specifically, the interaction between the client and the provider can be summarized as follows:

- After the client has collected the list of available flavours offered by the provider, it *notifies the intention of reserving a specific flavour* by sending an HTTP POST and including the ID of the flavour to be reserved.

- Once received by the provider, two separate actions are performed:

  - The provider checks if the flavour is still available (there might be some delay between the list flavour message and the subsequent reserve flavour request, thus the flavour may no longer be available). In case the flavour is still available, the provider *replies with a summary of the reservation process*; otherwise, a 404 error message is sent to the client.

  – The provider instantiates a timer to limit the reservation time for that specific flavour. This allows reserved flavours to be released in case either the client becomes *completely unresponsive*, or the subsequent purchase process exceeds a predefined threshold.

Figure 5.2 below extends the non-concurrent interaction, including concurrent access to shared resources (i.e., flavours), from multiple clients. Specifically, the interaction can be summarized with the following steps:

- **Customer 1** and **2** both request the list of available flavours based on predefined selectors, and they both *notify the intention to reserve a specific flavour* (i.e., flavour 1234 in this case).
- The **first customer** to send the reserve flavour message triggers, on the provider side, the acquisition of the lock associated with the shared flavour.
- The **first customer** can thus continue with the purchase of the selected flavour, whereas the second will not receive any further messages until the first customer releases the shared lock, either finalizing the purchase, or exceeding the predefined timeout.
- In case the **first customer** finalized the purchase, the **second customer** will acquire the shared lock and receive a 404-error message, notifying that the flavour is no longer available. In case the first customer didn't finalize the purchase, the second customer can proceed with the normal interaction described in the previous use case.

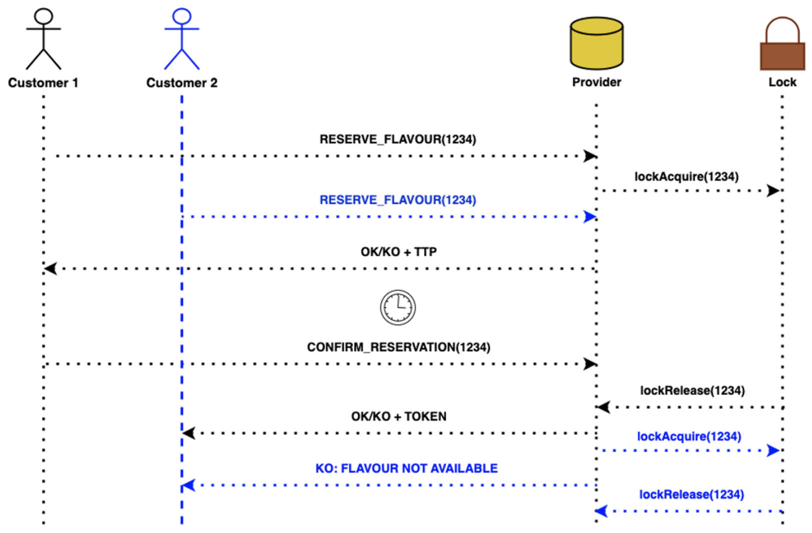

Figure 5.2: Concurrent flavour access from two different client

5.1.3 Subscribe to Changes

REAR defines a set of optional messages that extend the expressiveness of the protocol, summarized as *subscribe to changes*. Specifically, we include two optional interactions:

- **Refresh**, sent by the provider to refresh a particular flavour. By sending a refresh message, the provider helps maintain the availability of flavours and allows the consumer to effectively manage and allocate resources based on the updated expiration time.
- **Withdraw**, sent by the provider to the consumer to notify that a specific flavour is no longer available. This message serves as a notification mechanism to inform the consumer that the requested flavour is no longer available.

Figure 5.3 details the REAR interaction using the combination of both optional and required messages. Specifically, the interaction can be summarized as follows:

- The client sends a request to get the list of available flavours matching a predefined selector.
- The client notifies the provider of the intention to receive continuous updates on a specific flavour, using the *subscribe flavour message*, which is mapped onto an appropriate request message. The request message may vary depending on the implementation technology used (e.g. Websockets, publish/subscribe technologies).
- In case the client is interested in multiple flavours, this results in multiple *subscribe flavour messages*, one for each flavour.
- This internally triggers the creation of a stateful communication channel between the client and the provider.
- At this point, the provider sends asynchronous updates over the created channel to the client for the specified flavour. Two different types of updates are defined:
  - The *refresh expiration time message* notifies the client that a previous offer for a specific flavour is still valid.
  - The *withdraw message* notifies the client that a previous flavour offer is no longer available for purchase.

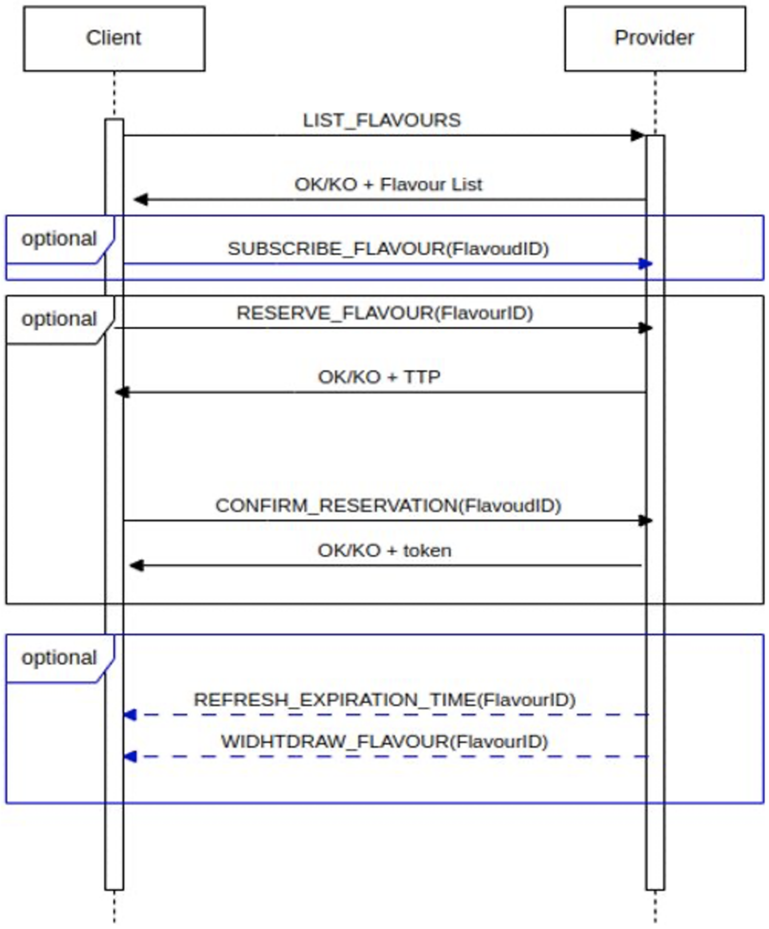
Client
Provider
LIST_FLAVOURS
OK/KO + Flavour List
optional
SUBSCRIBE_FLAVOUR(FlavoudID)
optional
RESERVE_FLAVOUR(FlavourID)
OK/KO + TTP
CONFIRM_RESERVATION(FlavoudID)
OK/KO + token
optional
REFRESH_EXPIRATION_TIME(FlavourID)
WIDHTDRAW_FLAVOUR(FlavourID)

Figure 5.3: REAR interaction using optional messages

5.2 REAR APIs

This chapter *details all the REAR messages*, their purpose, and the message body. Specifically, we can distinguish the messages as **required and optional**:

- **Required** (**Figure 5.1** details an example of possible interaction between client and provider using the required messages):
  - **List flavours**, sent by the client to probe the available flavours offered by a given provider.
  - **Reserve flavour**, sent by the client to perform a reservation on a specific flavour.

  - **Purchase flavour**, sent by the client to complete the purchase of an offered flavour.

- **Optional**:
  - **Refresh**, sent by the provider to refresh a particular flavour. By sending a refresh message, the provider helps maintain the availability of flavours and allows the consumer to effectively manage and allocate resources based on the updated expiration time.
  - **Withdrawal**, sent by the provider to the consumer to notify that a specific flavour is no longer available. This message serves as a notification mechanism to inform the consumer that the requested flavour is no longer available.

Note that the sequence of messages between the client and the provider is *fixed*, as well as the order. This is because each step requires a set of information returned from the previous step(s).

Moreover, there is a **huge difference in the communication pattern between required and optional messages**. Indeed, required messages follow a client/server approach, i.e., with the client always initiating the communication, whereas the optional messages are sent asynchronously by the server towards the clients. Such a design choice greatly improves the expressiveness of the protocol, but it calls for a different architectural style for communication (e.g., REST, Websocket, . . .), as the different types of messages have different requirements.

5.2.1 Required messages

This section details the required messages in the REAR protocol for resource advertisement, reservation and purchase.

LIST_FLAVOURS

No **request body** is required for this message. The **response body** is a list of flavours, as described in the following example:

Listing 5.1: Example of list flavours response body

```
[
    {
        "flavourID": "k8s-002",
        "providerID": "provider-001",
        "type": "k8s-fluidos",
        "characteristics": {
            "cpu": 8,
            "ram": 32
        },
        "policy": {
            "partitionable": {
                "cpuMinimum": 4,
                "ramMinimum": 16,
                "cpuStep": 2,
                "ramStep": 8
            }
        },
        "owner": {
            "ID": "owner-002",
            "IP": "192.168.0.2",
            "domainName": "example.com"
        },
        "price": {
            "amount": 29.99,
            "currency": "USD",
            "period": "month"
        },
        "expirationTime": "2023-07-31T12:00:00Z",
        "optionalFields": {}
    }
]
```

RESERVE_FLAVOUR

In the **request body** of this message, the client must specify the *flavourID* of the flavour to be reserved and its *identity*. The **response body** is a summary of the reservation process called *Transaction*, as described in the following example:

Listing 5.2: Example of reserve flavour response body

```
    {
        "transactionID": "1693306570195586000",
        "flavourID": "k8s-001",
        "startTime": "2023-08-29T12:56:10.19559+02:00"
    }
```

PURCHASE_FLAVOUR

In the **request body** of this message, the client must specify the *transactionID* of the transaction to be completed, the *identity* of the client and the *flavourID*. The **response body** is a confirmation message, like:

Listing 5.3: Example of purchase flavour response body

```
{
    "statusCode": 200
}
```

The implementation of what to return as a response is left to the user (by default, it returns 200 OK). However, one possible solution could be to return a “Contract” that confirms the successful acquisition of the flavour between the two parties.

5.2.2 Optional messages

The file format depends on the type of implementation. For now, the proposed implementation is with WebSocket, so messages are defined in XML (with the possibility of defining them in other formats such as JSON, etc.). As new technologies could be used in the future, other message formats may be introduced.

REFRESH_FLAVOUR

Talking about the *REFRESH_FLAVOUR* message, its XML structure is defined as follows:

- <RefreshMessage> is the root element of the "refresh" message.
- <Flavour> contains the details of the "Flavour" object that has been refreshed, with fields like FlavourID, ProviderID, FlavourType, and others.
- <ModificationDetails> contains the details of the changes made to the Flavour, including the modified fields, the old values, and the new values. It is possible to add additional fields to this section if necessary.

Listing 5.4: Structure XML of the refresh message

```
<RefreshMessage>
    <Flavour>
        <!-- Details of Flavour object that has been refreshed -->
        <FlavourID>string</FlavourID>
        <ProviderID>string</ProviderID>
        <FlavourType>string</FlavourType>
        <!-- Other Flavour fields ... -->
    </Flavour>
    <ModificationDetails>
        <!-- Details of the changes made to the Flavour -->
        <FieldModified>string</FieldModified>
        <OldValue>string</OldValue>
        <NewValue>string</NewValue>
        <!-- It is possibile to add other fields if needed -->
    </ModificationDetails>
</RefreshMessage>
```

WITHDRAW_FLAVOR

Talking about the *WITHDRAW_FLAVOR* message, its XML structure is defined as follows:

- <WithdrawMessage> is the root element of the "withdraw" message.
- <Flavour> contains the details of the "Flavour" object that is no longer available, with fields like FlavourID, ProviderID, FlavourType, and others.
- <Reason> contains the details of the reason for the withdrawal of the Flavour, including the message and other fields for more detailed reasons if needed.

Listing 5.5: Structure XML of the withdraw message

```
<WithdrawMessage>
    <Flavour>
        <!-- Details of the Flavour that is no longer available -->
        <FlavourID>string</FlavourID>
        <ProviderID>string</ProviderID>
        <FlavourType>string</FlavourType>
        <!-- Other Flavour fields ... -->
    </Flavour>
    <Reason>
        <!-- Reason for the withdrawal of the Flavour offer -->
        <Message>string</Message>
        <!-- Other fields for more detailed reasons if needed -->
    </Reason>
</WithdrawMessage>
```

5.3 Selector

A *selector* is a criterion or rule used to choose or select specific entities from a larger set based on certain characteristics or properties.

In the context of choosing a Virtual Machine (VM), a selector can be defined as a set of criteria that specify the desired characteristics of the VM, such as CPU, RAM, storage capacity, operating system, or any other relevant attributes.

The goal is to establish a standardized selector that allows users to define their selection criteria based on their specific needs. For example, if a user requires a VM, they would select the VM type in the selector, which would then reveal the corresponding fields specific to VMs. Similarly, if a user needs a service, they would choose the service type, and additional fields specific to that service type would be displayed.

One of the challenges in utilizing selectors effectively is **understanding the syntax required to construct them accurately**. Clients need to be familiar with the specific syntax and structure of the selector to request resources correctly. To address this issue, clients are allowed to use **standardized selectors**, allowing users to choose from a variety of available **flavours** based on their specific needs. However, a crucial question arises: How will the client know which types of flavours are offered by the provider? One possible solution is to implement a *pre-request mechanism* (*init*), where the client sends a GET request to the provider to retrieve a comprehensive list of available flavours before engaging any resources. This approach ensures that the client is aware of the available options and can make informed decisions when selecting the appropriate flavour. Furthermore, it allows adding new providers as plugins (e.g., adding AWS VM type).

Chapter 6

Development of FLUIDOS Components

This chapter explores the development of critical system components, which were implemented using Golang and driven by **controller logic**. It delves into the architectural considerations, design principles, and implementation details that underpin these components.

Readers will gain insights into the choices made during development, the reasoning behind them, and how they contributed to the overall system's functionality. Challenges encountered in the development process and their resolutions will also be discussed.

The chapter offers a comprehensive view of the development efforts, shedding light on the core components that form the backbone of our system.

The development of FLUIDOS components was divided into two releases:

- the **first release** leverages the power of NATS messaging, offering a seamless and efficient communication layer within our system. It showcases the integration of NATS as a fundamental component, emphasizing its role in enhancing communication between system modules.
- In the **second release**, a different approach is adopted, primarily relying on Kubernetes resources, including Custom Resources and the Kubernetes API itself. This release demonstrates a commitment to leveraging Kubernetes' native capabilities to drive the system's functionalities.

6.1 First Release

In the initial release of our system, two fundamental concepts played pivotal roles: the **Producer** and the **Consumer**. These concepts serve as the core components facilitating communication and data flow within our Kubernetes ecosystem.

Listing 6.1: Functions used to publish and consume messages Flavours on NATS

```go
// publishMessage publishes a message to a subject
func publishMsg(nc *nats.Conn, subject string, message []byte) error {
    // Publish message
    err := nc.Publish(subject, message)
    if err != nil {
        return err
    }
    // Flush connection to ensure that message is sent
    err = nc.Flush()
    if err != nil {
        return err
    }

    return nil
}

// handleMsg handles the message received
func handleMsg(body []byte) {
    var flavour Flavour
    err := json.Unmarshal(body, &flavour)

    if err != nil {
        log.Printf("failed to unmarshal JSON: %v", err)
        return
    }

    updateFlavourMap(flavourMap, flavour)

}
```

6.1.1 The Producer

At the heart of our system lies the Producer, a critical component functioning as a Kubernetes controller. This component is meticulously crafted using the controller-runtime library and operates as a vigilant overseer of the Kubernetes cluster. Its primary responsibilities include monitoring resource changes within the cluster and retrieving essential node metrics when resources are added, modified, or removed.

Once gathered, these metrics undergo careful processing before being transmitted to a NATS server. The Producer is implemented using the Golang programming language, and NATS serves as the efficient communication channel. Together, they ensure that our system remains well-informed, responsive, and prepared for subsequent actions.

6.1.2 The Consumer

Conversely, the Consumer plays a crucial role in our system. This component functions as an application built on the NATS client library, specializing in receiving messages from a designated NATS topic. It acts as a vigilant sentinel, continuously monitoring the specified NATS topic for incoming messages.

Upon receiving messages, the Consumer employs predefined functionality to process or log the data according to specific requirements. Its actions are tailored to the desired functionality, encompassing tasks ranging from data processing to meticulous logging.

In summary, the Consumer ensures that our system remains receptive to incoming data, efficiently handling messages dispatched via NATS channels.

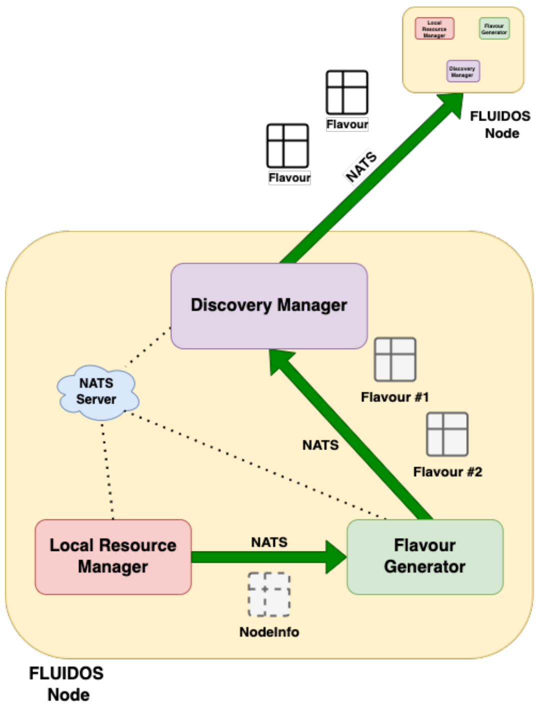

Figure 6.1: Overview of system components developed using NATS

6.1.3 Components

Both the Producer and the Consumer have the same components, which are described below:

Local Resource Manager

Local Resource Manager in Fluidos is a vital component responsible for managing the local resources available within each Fluidos node.

The Local Resource Manager operates within an individual node and is responsible for monitoring and controlling the resources present on that node.

Listing 6.2: Function to get the resources of a node

```
// GetNodesResources retrieves the metrics from all the worker nodes in the cluster
func GetNodesResources(ctx context.Context, cl client.Client) (*[]NodeInfo, error) {
    // Set a label selector to filter worker nodes
    labelSelector := labels.Set{workerLabelKey: ""}.AsSelector()

    // Get a list of nodes
    nodes := &corev1.NodeList{}
    err := cl.List(ctx, nodes, &client.ListOptions{
        LabelSelector: labelSelector,
    })
    if err != nil {
        return nil, err
    }

    // Get a list of nodes metrics
    nodesMetrics := &metricsv1beta1.NodeMetricsList{}
    err = cl.List(ctx, nodesMetrics, &client.ListOptions{
        LabelSelector: labelSelector,
    })
    if err != nil {
        return nil, err
    }

    var nodesInfo []NodeInfo
    // Print the name of each node
    for _, node := range nodes.Items {
        for _, metrics := range nodesMetrics.Items {
            if node.Name != metrics.Name {
                // So that we can select just the nodes that we want
                continue
            }

            metricsStruct := getNodeResourceMetrics(&metrics, &node)
            nodeInfo := getNodeInfo(&node, metricsStruct)
            nodesInfo = append(nodesInfo, *nodeInfo)
            uids = append(uids, string(node.UID))
        }
    }
    return &nodesInfo, nil
}
```

Through the Local Resource Manager, each node can effectively manage its processing power, storage capacity, memory, network bandwidth, and other resources. It provides a centralized interface for interacting with the node's resources.

The Local Resource Manager is implemented as a Kubernetes controller, which is a component that continuously monitors the state of the Kubernetes cluster and makes changes to the cluster's resources when necessary.

Discovery Manager

Discovery Manager in Fluidos is a crucial component responsible for facilitating the dynamic discovery of nodes within the Fluidos network.

By leveraging the Discovery Manager, nodes can efficiently communicate and exchange information about their capabilities, resources, and availability. This enables the Fluidos ecosystem to adapt and optimize resource allocation based on the dynamic changes in the network.

The Discovery Manager is implemented as a Kubernetes controller, which is a component that continuously monitors the state of the Kubernetes cluster and makes changes to the cluster's resources when necessary.

Flavour Generator

The Flavour Generator provides:

- A Southbound API to collect capabilities from the Local Resource Manager and from the Discovery Manager; these capabilities are "static," so we are not referring to dynamic metrics.
- An aggregator based on a custom algorithm to sum numeric resource properties, perform computations, and build a set of flavors representing the maximum homogeneous set of resources to be announced and exploited for a remote workflow.
- A Northbound API to export the results of aggregation to the Discovery Manager, allowing it to disseminate this information to the Discovery Managers of other nodes or to a Catalog.

Listing 6.3: Function to generate a flavour

```go
// splitResources produces different Flavours with intelligent resource allocation
func splitResources(node NodeInfo) []Flavour {
    AvailCPU := node.ResourceMetrics.CPUAvailable
    AvailMemory := node.ResourceMetrics.MemoryAvailable

    // Define initial values for resource allocation
    var cpuAllocation float32 = 1.0
    var memoryAllocation float32 = 1.0

    flavours := []Flavour{}
    for AvailCPU > cpuAllocation && AvailMemory > memoryAllocation {

        // Create the flavour
        flavour := Flavour{
            FlavourID: generateUniqueString(node.UID + "-flavour-" + strconv.Itoa(len(
                flavours)+1)),
            NodeUID: node.UID,
            Name: node.Name + "-flavour-" + strconv.Itoa(len(flavours)+1),
            Architecture: node.Architecture,
            OperatingSystem: node.OperatingSystem,
            CPUOffer: fmt.Sprintf("%.2f", cpuAllocation),
            MemoryOffer: fmt.Sprintf("%.2fGi", memoryAllocation),
            Available: true,
            PodsOffer: []PodsPlan{
                {Name: "Small", Available: true, Pods: 11},
                {Name: "Medium", Available: true, Pods: 33},
                {Name: "Large", Available: true, Pods: 66},
            },
        }
        flavours = append(flavours, flavour)

        // Increase the resource allocation for the next flavour
        cpuAllocation += float32(len(flavours) + 1)
        memoryAllocation += float32(len(flavours) + 1)

        // Check if the allocation exceeds the available resources
        if cpuAllocation > AvailCPU {
            cpuAllocation = AvailCPU
        }
        if memoryAllocation > AvailMemory {
            memoryAllocation = AvailMemory
        }
    }

    return flavours
}
```

Once these flavors are created, they will be pushed to a remote database. For this release, *MongoDB* was chosen as the database. The procedure for pushing these flavors to the remote DB also simulated the behavior of the Catalog. As we will see later, the remote DB will be abandoned in the second release.

6.2 Second Release

In the second release, we have no longer used NATS as a communication channel. Instead, we have used the Kubernetes API to communicate with the Kubernetes cluster. This release is more focused on the use of Kubernetes resources, such as Custom Resources, to implement the functionalities of the system.

Moreover, to communicate outside the cluster, we use REAR protocol, based on REST APIs + WebSocket in order to have a bidirectional communication between the nodes (or the Catalog) and the remote clients for the discovery, reserve and acquisition of resources.

6.2.1 Components

In this release, we have the following components:

- **Local Resource Manager**: it is the same component as in the first release, but it has been adapted to use the Kubernetes Custom Resources instead of NATS.
- **Available Resources**: component that is in charge of collecting the resources available flavours.
- **Discovery Manager**: it is the same component as in the first release, but it has been adapted to use the Kubernetes Custom Resources instead of NATS (and also some logic has been added to manage the new functionalities).
- **Peering Candidates**: component that is in charge of collecting the potential peering candidates from the outside of the cluster.
- **REAR Manager**: central component that is in charge of managing the request for resource advertisement, discovery, reservation and acquisition.
- **Contract Manager**: component that is in charge of managing the reserve and acquisition of resources.

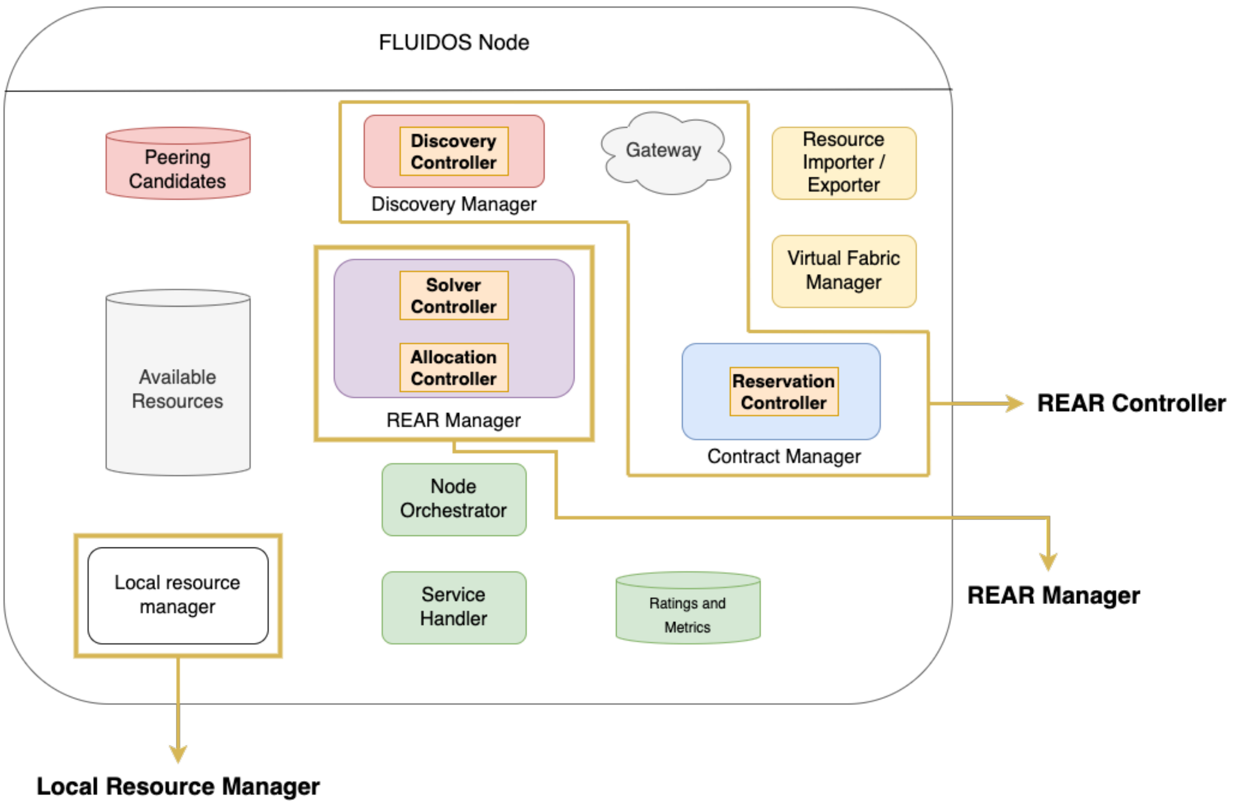

Figure 6.2: Overview of system components developed in the last release

Local Resource Manager

The Local Resource Manager is the same component as in the first release, but it has been adapted to use the Kubernetes Custom Resources instead of NATS.

For each worker node in the FLUIDOS Node, the Local Resource Manager starting from the node's resources, creates a *Flavour Custom Resource* and pushes it to the Kubernetes API.

Available Resources

The Available Resources component is in charge of collecting the resources available flavours. It is implemented as a simple database (*etcd*) that stores the Flavour Custom Resources created by the Local Resource Manager.

Discovery Manager

The Discovery Manager is the same component as in the first release, but it has been adapted to use the Kubernetes Custom Resources instead of NATS (and also some logic has been added to manage the new functionalities).

Its main objective is to populate the *Peering Candidates* table and offer appropriate flavors to the *REAR Manager*. This process hinges on the initial messages exchanged as part of the REAR protocol.

The Discovery Manager plays a pivotal role in this process. If there are no appropriate peering candidates available, the REAR Manager constructs a **flavor selector** and forwards it to the Discovery Manager, seeking a suitable peering candidate. Subsequently, the Discovery Manager on the Consumer Node initiates a **LIST_FLAVOURS** message, disseminating it to all known endpoints, whether they are local Nodes, a Supernode, or a Catalog.

Listing 6.4: Function to search for flavours given a Flavor Selector

```go
// SearchFlavour is a function that returns an array of Flavour that fit the Selector by performing a get request to an http server
func SearchFlavour(selector nodecorev1alpha1.FlavourSelector) ([]*nodecorev1alpha1.Flavour, error) {
    // Marshal the selector into JSON bytes
    body := parseutil.ParseSelectorValues(selector)

    klog.Info("Selector is ", body)
    selectorBytes, err := json.Marshal(body)
    if err != nil {
        return nil, err
    }

    // Create the Flavour CR from the first flavour in the array of Flavour
    var flavoursCR []*nodecorev1alpha1.Flavour

    // Send the POST request to all the servers in the list
    for _, ADDRESS := range flags.SERVER_ADDRESSES {
        resp, err := http.Post(ADDRESS+"/listflavours/selector", "application/json", bytes.NewBuffer(selectorBytes))
        if err != nil {
            return nil, err
        }
        defer resp.Body.Close()

        // Check if the response status code is 200 (OK)
        if resp.StatusCode != http.StatusOK {
            return nil, fmt.Errorf("received non-OK response status code: %d", resp.StatusCode)
        }

        // Read the response body
        respBody, err := ioutil.ReadAll(resp.Body)
        if err != nil {
            return nil, err
        }

        // Unmarshal the response JSON into an array of Flavour Object
        var flavours []*models.Flavour
        if err := json.Unmarshal(respBody, &flavours); err != nil {
            return nil, err
        }

        for _, flavour := range flavours {
            klog.Infof("Flavour found: %s", flavour.FlavourID)
            cr := resourceforge.ForgeFlavourCustomResource(*flavour)
            flavoursCR = append(flavoursCR, cr)
        }

    }
    klog.Info("Flavours created", flavoursCR)
    return flavoursCR, nil

}
```

On the Provider Node's side, its Discovery Manager searches for suitable local flavors that match the received flavor selector within its Available Resources table. Should one or more fitting flavors be identified, the Provider Node's Discovery Manager responds with an OK message, including the list of discovered flavors, to the Consumer Node's Discovery Manager.

The Discovery Manager on the Consumer Node, in turn, populates the **Peering Candidates** table with the newly acquired flavors, facilitating the subsequent steps in the process.

Peering Candidates

The Peering Candidates component manages a dynamic list of nodes that are potentially suitable for establishing peering connections. This list is continuously updated based on the available resources in the nodes and the requests for flavours from the Discovery Manager.

REAR Manager

The REAR Manager plays a pivotal role in orchestrating the service provisioning process. It receives service requests, translates them into resource or service requests, and looks up suitable resources.

If none are found, it initiates the Discovery, Reservation, and Peering phases. It selects potential peering candidates and can request discovery. The suitable local flavors are identified and shared. If a suitable candidate is found, it triggers the Reservation phase. Resources are allocated, contracts are created, and peering is established. For Liqo releases with declarative Virtual Node definitions, the process is modified. Finally, the Node Orchestrator deploys workloads using Kubernetes primitives and the retrieved resources.

Contract Manager

The Contract Manager is in charge of managing the reserve and acquisition of resources. It handles the negotiation and management of resource contracts between nodes.

When a suitable peering candidate is identified, the Contract Manager initiates the Reservation phase by sending a **RESERVE_FLAVOUR** message.

Listing 6.5: Function to reserve a flavour

```go
// reserveFlavourHandler reserves a Flavour by its flavourID
func reserveFlavourHandler(cl client.Client) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        params := mux.Vars(r)
        flavourID := params["flavourID"]
        var transaction models.Transaction

        var request struct {
            FlavourID string `json:"flavourID"`
            Buyer models.Owner `json:"buyer"`
        }

        if err := json.NewDecoder(r.Body).Decode(&request); err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        if flavourID != request.FlavourID {
            http.Error(w, "Mismatch body & param", http.StatusConflict)
            return
        }

        if models.Transactions == nil {
            models.Transactions = make(map[string]models.Transaction)
        }

        found := false
        for _, t := range models.Transactions {
            if t.FlavourID == flavourID && t.Buyer.ID == request.Buyer.ID {
                found = true
                t.StartTime = common.GetTimeNow()
                transaction = t
                addNewTransacion(t)
                break
            }
        }

        if !found {
            klog.Infof("Reserving flavour %s started", flavourID)

            flavour, _ := services.GetFlavourByID(flavourID, cl)
            if flavour == nil {
                http.Error(w, "Flavour not found", http.StatusNotFound)
                return
            }

            transactionID, err := namings.ForgeTransactionID()
            if err != nil {
                http.Error(w, "Error generating transaction ID", http.StatusInternalServerError)
                return
            }

            transaction := resourceforge.ForgeTransactionObject(flavourID, transactionID, request.Buyer)
            addNewTransacion(transaction)
        }
        encodeResponse(w, transaction)
    }
}
```

Upon successful reservation of resources, it proceeds to create a new contract. Following this, a **PURCHASE_FLAVOUR** message is sent to the Provider Node's Contract Manager.

Listing 6.6: Function to purchase a flavour

```go
// purchaseFlavourHandler is an handler for purchasing a Flavour
func purchaseFlavourHandler(cl client.Client) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        params := mux.Vars(r)
        flavourID := params["flavourID"]
        var purchase models.Purchase

        if err := json.NewDecoder(r.Body).Decode(&purchase); err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        klog.Infof("Purchasing flavour %s started", flavourID)

        if models.Transactions == nil {
            klog.Infof("No active transactions found")
            http.Error(w, "Error: no active transactions found.", http.StatusNotFound)
            return
        }

        transaction, exists := models.Transactions[purchase.TransactionID]
        if !exists {
            klog.Infof("Transaction not found")
            http.Error(w, "Error: transaction not found", http.StatusNotFound)
            return
        }

        if common.CheckExpiration(transaction.StartTime, flags.EXPIRATION_TRANSACTION) {
            http.Error(w, "Error: transaction Timeout", http.StatusRequestTimeout)
            delete(models.Transactions, purchase.TransactionID)
            return
        }

        var contractList *reservationv1alpha1.ContractList
        var contract *reservationv1alpha1.Contract

        if err := cl.List(context.Background(), contractList, client.MatchingFields{"spec.TransactionID": purchase.TransactionID}); err != nil {
            if client.IgnoreNotFound(err) != nil {
                klog.Errorf("Error when listing Contracts: %s", err)
                http.Error(w, "Error when listing Contracts", http.StatusInternalServerError)
                return
            }
        }

        if len(contractList.Items) > 0 {
            contract = &contractList.Items[0]
            contractObject := resourceforge.ForgeContractObject(contract, purchase.BuyerID, transaction.TransactionID, purchase.Partition)
            responsePurchase := resourceforge.ForgeResponsePurchaseObject(contractObject)
            encodeResponse(w, responsePurchase)
            return
        }

        klog.Infof("Performing purchase of flavour %s...", flavourID)
        delete(models.Transactions, purchase.TransactionID)
        klog.Infof("Flavour %s purchased!", flavourID)

        flavourSold, err := services.GetFlavourByID(flavourID, cl)
        if err != nil {
            klog.Errorf("Error getting the Flavour by ID: %s", err)
            http.Error(w, "Error getting the Flavour by ID", http.StatusInternalServerError)
            return
        }

```

```
		klog.Infof("Creating a new contract...")
		contract = resourceforge.ForgeContractCustomResource(*flavourSold, purchase.
			BuyerID)
		err = cl.Create(context.Background(), contract)
		if err != nil {
			klog.Errorf("Error creating the Contract: %s", err)
			http.Error(w, "Error creating the Contract: "+err.Error(), http.
				StatusInternalServerError)
			return
		}
		klog.Infof("Contract created!")

		contractObject := resourceforge.ForgeContractObject(contract, purchase.BuyerID,
			transaction.TransactionID, purchase.Partition)
		responsePurchase := resourceforge.ForgeResponsePurchaseObject(contractObject)

		encodeResponse(w, responsePurchase)
	}
}
```

Once both sides have agreed to the terms, the Contract Manager on each node creates a new contract. Subsequently, the REAR Manager is informed of the reserved resources, leading to updates in the Available Resources table. Finally, the Contract Manager triggers the establishment of peering reachability through network managers, and the Node Orchestrator can proceed with workload deployment.

6.2.2 Controllers

In the following, the controllers developed for the FLUIDOS Node are described.

Solver Controller

The Solver controller, tasked with reconciliation on the `Solver` object, continuously monitors and manages its state to ensure alignment with the desired configuration. It follows the following steps:

1. When there is a new Solver object, it firstly checks if the `Solver` has expired or failed (if so, it marks the Solver as `Timed Out`).
2. It checks if the Solver has to find a candidate.
3. If so, it starts to search a matching Peering Candidate if available.
4. If some Peering Candidates are available, it selects one and books it.

5. If no Peering Candidates are available, it starts the discovery process by creating a `Discovery`.
6. If the `findCandidate` status is solved, it means that a Peering Candidate has been found. Otherwise, it means that the `Solver` has failed.
7. If in the `Solver` there is also a `ReserveAndBuy` phase, it starts the reservation process. Otherwise, it ends the process, the solver is already solved.
8. Firstly, it starts to get the `PeeringCandidate` from the `Solver` object. Then, it forges the Partition starting from the `Solver` selector. At this point, it creates a `Reservation` object.
9. If the `Reservation` is successfully fulfilled, it means that the `Solver` has reserved and purchased the resources. Otherwise, it means that the `Solver` has failed.
10. If in the `Solver` there is also an `EnstablishPeering` phase, it starts the peering process (to be implemented). Otherwise, it ends the process.

Discovery Controller

The Discovery controller, tasked with reconciliation on the `Discovery` object, continuously monitors and manages its state to ensure alignment with the desired configuration. It follows the following steps:

1. When there is a new Discovery object, it firstly starts the discovery process by contacting the `Gateway` to discover flavors that fit the `Discovery` selector.
2. If no flavors are found, it means that the `Discovery` has failed. Otherwise, it refers to the first `PeeringCandidate` as the one that will be reserved (more complex logic should be implemented), while the others will be stored as not reserved.
3. It updates the `Discovery` object with the `PeeringCandidates` found.
4. The `Discovery` is solved, so it ends the process.

Reservation Controller

The Reservation controller, tasked with reconciliation on the `Reservation` object, continuously monitors and manages its state to ensure alignment with the desired configuration. It follows the following steps:

1. When there is a new `Reservation` object, it checks if the `Reserve` flag is set. If so, it starts the **Reserve** process.

2. It retrieves the FlavourID from the `PeeringCandidate` of the `Reservation` object. With this information, it starts the reservation process through the `Gateway`.

3. If the reserve phase of the reservation is successful, it will create a `Transaction` object from the response received. Otherwise, the `Reservation` has failed.

4. If the `Reservation` has the `Purchase` flag set, it starts the **Purchase** process. Otherwise, it ends the process because the `Reservation` has already succeeded.

5. Using the `Transaction` object from the `Reservation`, it starts the purchase process.

6. If the purchase phase is successfully fulfilled, it will update the status of the `Reservation` object and it will store the received `Contract`. Otherwise, the `Reservation` has failed.

Allocation Controller

The Allocation controller, tasked with reconciliation on the `Allocation` object, continuously monitors and manages its state to ensure alignment with the desired configuration.

6.2.3 Custom Resources

In the following, the Custom Resources developed for the FLUIDOS Node are described.

Discovery

Here is a `Discovery` Custom Resource example:

Listing 6.7: Discovery Custom Resource example

```
apiVersion: advertisement.fluidos.eu/v1alpha1
kind: Discovery
metadata:
  name: discovery-solver1
spec:
  selector:
    type: k8s−fluidos
    architecture: arm64
    rangeSelector:
      minCpu: 1
      minMemory: 1
  solverID: solver1
  subscribe: true
```

Reservation

Here is a `Reservation` Custom Resource example:

Listing 6.8: Solver Custom Resource example

```
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Reservation
metadata:
  name: reservation−solver1
spec:
  buyer:
    domain: topix.fluidos.eu
    ip: 17.3.4.11
    nodeID: 95c0614o1d0
  flavourID: k8s−002
  peeringCandidate:
    name: peeringcandidate−k8s−002
    namespace: default
  purchase: true
  reserve: true
  seller:
    domain: polito.fluidos.eu
    ip: 13.3.5.1
    nodeID: 91cbd32s0q1
```

Flavour

Here is a `Flavour` Custom Resource example:

Listing 6.9: Flavour Custom Resource example

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Flavour
```

```
metadata:
  name: k8s-fluidos-002
  namespace: default
spec:
  characteristics:
    architecture: amd64
    cpu: 4
    memory: 16
  optionalFields:
    availability: true
  owner:
    domain: polito.fluidos.eu
    ip: 13.3.5.1
    nodeID: 91cbd32s0q1
  policy:
    aggregatable:
      maxCount: 5
      minCount: 1
  price:
    amount: "10"
    currency: USD
    period: hourly
  providerID: 05a2a55a-9939-4e94-9587-barlo14
  type: k8s-fluidos
```

Contract

Here is a `Contract` Custom Resource example:

Listing 6.10: Contract Custom Resource example

```
Name: contract-k8s-fluidos-002-4o5g
API Version: reservation.fluidos.eu/v1alpha1
Kind: Contract
Spec:
  Buyer:
    domain: topix.fluidos.eu
    ip: 17.3.4.11
    nodeID: 95c0614o1d0
  Credentials:
    Cluster ID:
    Cluster Name:
```

```
    Endpoint:
    Token:
  Flavour:
    Spec:
      Characteristics:
        Architecture:
        Cpu: 4
        Ephemeral - Storage: 0
        Gpu: 0
        Memory: 16
        Persistent - Storage: 0
      Optional Fields:
        Availability: true
      Owner:
        domain: polito.fluidos.eu
        ip: 13.3.5.1
        nodeID: 91cbd32s0q1
      Policy:
        Aggregatable:
          Max Count: 5
          Min Count: 1
      Price:
        Amount: 10
        Currency: USD
        Period: hourly
      Provider ID: 05a2a55a-9939-4e94-9587-barlo14
      Type: k8s-fluidos
    Status:
      Creation Time: 2023-09-29T10:22:13+02:00
      Expiration Time: 2023-11-29T10:22:13+02:00
      Last Update Time: 2023-09-29T11:26:37+02:00
  Partition:
    Cpu: 0
    Ephemeral Storage: 0
    Gpu: 0
    Memory: 0
    Storage: 0
  Seller:
    domain: polito.fluidos.eu
    ip: 13.3.5.1
    nodeID: 91cbd32s0q1
```

PeeringCandidate

Here is a `PeeringCandidate` Custom Resource example:

Listing 6.11: PeeringCandidate Custom Resource example

```
Name: peeringcandidate-k8s-fluidos-002
API Version: advertisement.fluidos.eu/v1alpha1
Kind: PeeringCandidate
Spec:
  Flavour:
    Spec:
      Characteristics:
        Architecture:
        Cpu: 4
        Ephemeral - Storage: 0
        Gpu: 0
        Memory: 16
        Persistent - Storage: 0
      Optional Fields:
        Availability: true
      Owner:
        domain: polito.fluidos.eu
        ip: 13.3.5.1
        nodeID: 91cbd32s0q1
      Policy:
        Aggregatable:
          Max Count: 5
          Min Count: 1
      Price:
        Amount: 10
        Currency: USD
        Period: hourly
      Provider ID: 05a2a55a-9939-4e94-9587-barlo14
      Type: k8s-fluidos-fluidos
    Status:
      Creation Time: 2023-09-29T10:22:13+02:00
      Expiration Time: 2023-11-29T10:22:13+02:00
      Last Update Time: 2023-09-29T11:26:37+02:00
  Reserved: true
  Solver ID: solver1
```

Solver

Here is a `Solver` Custom Resource example:

Listing 6.12: Solver Custom Resource example

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Solver
metadata:
  name: solver1
spec:
  selector:
    type: k8s-fluidos-fluidos
    architecture: arm64
    rangeSelector:
      minCpu: 1
      minMemory: 1
  solverID: solver1
  findCandidate: true
  enstablishPeering: false
```

Transaction

Here is a `Transaction` Custom Resource example:

Listing 6.13: Transaction Custom Resource example

```
Name: 2738d980b9c4c45a172c7a4e18273492-1693646797264948000
API Version: reservation.fluidos.eu/v1alpha1
Kind: Transaction
Spec:
  Flavour ID: k8s-fluidos-002
  Start Time: 2023-09-29T11:26:37+02:00
```

Chapter 7

Validation

This chapter aims to validate the effectiveness, reliability, and usability of the core components of the FLUIDOS Node system, shedding light on their role in efficient communication and resource management. Moreover, it seeks to evaluate the practicality and security of the REAR Protocol in enabling secure data exchange of resources and capabilities among diverse cloud providers.

7.1 Methodology

We conducted different tests in different settings to evaluate their real-world performance.

Our validation process began with controlled tests using local Kind clusters, each running FLUIDOS. These tests helped us assess FLUIDOS' core functionalities under controlled conditions.

Subsequently, we extended our validation to real-world Kubernetes clusters. This shift allowed us to evaluate FLUIDOS' adaptability and scalability in authentic edge computing environments, considering factors like network latency, resource availability, and security.

7.2 Usage Scenario

To validate the system, we considered a usage scenario involving two FLUIDOS nodes within the same domain. Specifically, we examined a scenario where an **intent** (a request for a specific flavor) cannot be fulfilled by one node. In response, a discovery process is initiated towards the other node to determine if it has the required flavor available. If so, the process proceeds to reserve and acquire that flavor, culminating in the creation of a contract between the two parties.

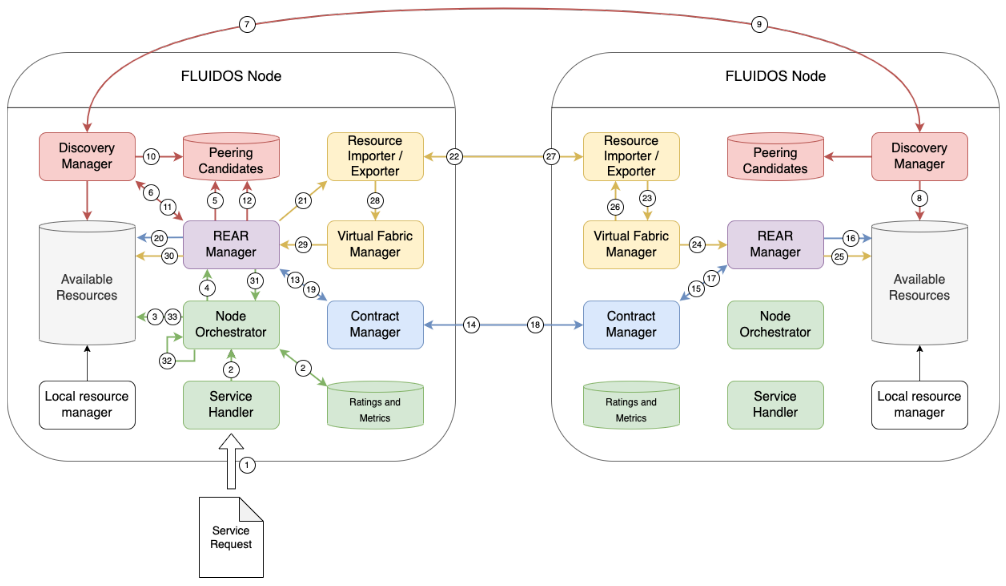

Figure 7.1: Usage scenario of the FLUIDOS Node system.

7.3 Functional Testing

We conducted functional testing to ensure that the system's core functionalities work as expected. We tested the following components:

- **Local Resource Manager**
- **Discovery Manager**
- **REAR Manager**
- **Contract Manager**

Upon booting, the **Local Resource Manager** is tasked with generating a Flavour object based on the resources of each worker node. Subsequently, the Flavour object is utilized to establish a Kubernetes Custom Resource Definition (CRD), enabling the creation of a Kubernetes Custom Resource (CR) for each Flavour. These CRs are then employed by the Discovery Manager to disseminate information about available resources to other FLUIDOS nodes.

For instance, in a cluster with multiple worker nodes, the expected outcome is to have one `Flavour` object and one CRD generated for each node. To illustrate, let's consider an example starting from one of the machines where the Local Resource Manager is running:

Listing 7.1: Flavour object generated by the Local Resource Manager.

```yaml
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Flavour
metadata:
  name: k8s-fluidos-002
  namespace: default
spec:
  characteristics:
    architecture: amd64
    cpu: 4
    memory: 16
  optionalFields:
    availability: true
  owner:
    domain: polito.fluidos.eu
    ip: 13.3.5.1
    nodeID: 91cbd32s0q1
  policy:
    aggregatable:
      maxCount: 5
      minCount: 1
  price:
    amount: "10"
    currency: USD
    period: hourly
  providerID: 05a2a55a-9939-4e94-9587-barlo14
  type: k8s-fluidos
```

For this functional test, let's consider an intent from a user to acquire a flavor with the following characteristics:

Listing 7.2: Intent to acquire a flavor with a set of characteristics.

```yaml
selector:
  type: k8s-fluidos-fluidos
  architecture: arm64
  rangeSelector:
    minCpu: 1
    minMemory: 1
```

7.4 Results

Starting from the intent, the **REAR Manager** looks up in the `Peering Candidates` table for potential peering candidates (suitable flavours) matching the request.

7.4.1 Solver Creation

In case there are no suitable peering candidates, firstly it will be created a `Solver` with the following characteristics:

Listing 7.3: Solver created by the REAR Manager.

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Solver
metadata:
  name: solver1
spec:
  selector:
    type: k8s-fluidos-fluidos
    architecture: arm64
    rangeSelector:
      minCpu: 1
      minMemory: 1
  solverID: solver1
  findCandidate: true
  reserveAndBuy: true
  enstablishPeering: false
```

7.4.2 Discovery Phase

With this solver, it will be created a new `Discovery` CR to find potential Peering Candidates, which in this functional test is the following:

Listing 7.4: Discovery created by the REAR Manager.

```
apiVersion: advertisement.fluidos.eu/v1alpha1
kind: Discovery
metadata:
  name: discovery-solver1
spec:
  selector:
    type: k8s-fluidos
    architecture: arm64
    rangeSelector:
      minCpu: 1
      minMemory: 1
  solverID: solver1
  subscribe: true
```

The Discovery Manager of the **Consumer Node** sends a LIST_FLAVOURS message to all the endpoints it already knows (in this case, the other FLUIDOS Nodes).

The Discovery Manager of the **Provider Node** looks-up for suitable local flavours matching the received flavour selector in its Available Resources table. In case one or more suitable flavours are found, the Discovery Manger of the Provider Node sends back an OK message to the Discovery Manager of the Consumer Node attaching the flavours list.

The Discovery Manager of the Consumer Node populates the `Peering Candidates` table with the newly discovered flavours. In this case, there was just one flavour matching the request, so the `Peering Candidates` table will be populated with the following entry:

Listing 7.5: Peering Candidate created if a suitable flavour is found.

```
Name: peeringcandidate-k8s-fluidos-002
API Version: advertisement.fluidos.eu/v1alpha1
Kind: PeeringCandidate
Spec:
  Flavour:
    Spec:
      Characteristics:
        Architecture:
        Cpu: 4
        Ephemeral - Storage: 0
        Gpu: 0
        Memory: 16
        Persistent - Storage: 0
      Optional Fields:
        Availability: true
      Owner:
        domain: polito.fluidos.eu
        ip: 13.3.5.1
        nodeID: 91cbd32s0q1
      Policy:
        Aggregatable:
          Max Count: 5
          Min Count: 1
      Price:
        Amount: 10
        Currency: USD
        Period: hourly
      Provider ID: 05a2a55a-9939-4e94-9587-barlo14
      Type: k8s-fluidos-fluidos
    Status:
      Creation Time: 2023-09-29T10:22:13+02:00
      Expiration Time: 2023-11-29T10:22:13+02:00
      Last Update Time: 2023-09-29T11:26:37+02:00
  Reserved: true
  Solver ID: solver1
```

After that, the Discovery will be solved and updated with the new Peering Candidate.

7.4.3 Reserve and Buy Phases

In the solver, there is also the intent to Reserve and Buy the flavour. For this reason, the REAR Manager will create a new `Reservation` CR, which in this functional test is the following:

Listing 7.6: Reservation created by the REAR Manager.

```
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Reservation
metadata:
  name: reservation-solver1
spec:
  buyer:
    domain: topix.fluidos.eu
    ip: 17.3.4.11
    nodeID: 95c0614o1d0
  flavourID: k8s-002
  peeringCandidate:
    name: peeringcandidate-k8s-002
    namespace: default
  purchase: true
  reserve: true
  seller:
    domain: polito.fluidos.eu
    ip: 13.3.5.1
    nodeID: 91cbd32s0q1
```

This `Reservation` CR will be used by the Contract Manager to send a RESERVE_FLAVOUR message to the Provider Node. The Provider Node will then reserve the flavour and send back an `Transaction` message to the Consumer Node with the following characteristics:

Listing 7.7: Transaction message sent by the Provider Node.

```
Name: 2738d980b9c4c45a172c7a4e18273492-1693646797264948000
API Version: reservation.fluidos.eu/v1alpha1
Kind: Transaction
Spec:
  Flavour ID: k8s-fluidos-002
  Start Time: 2023-09-29T11:26:37+02:00
```

After that, the Contract Manager will send a PURCHASE_FLAVOUR message to the Provider Node using the TransactionID previously received. The Provider Node will then create a `Contract` to notify the Consumer Node that the flavour has been purchased.

7.4.4 Contract Storage

Finally, the Contract Manager of the Consumer will create a new `Contract` based on the Contract received from the Provider Node:

Listing 7.8: Contract created by the Contract Manager.

```
Name: contract-k8s-fluidos-002-4o5g
API Version: reservation.fluidos.eu/v1alpha1
Kind: Contract
Spec:
  Buyer:
    domain: topix.fluidos.eu
    ip: 17.3.4.11
    nodeID: 95c0614o1d0
  Credentials:
    Cluster ID:
    Cluster Name:
    Endpoint:
    Token:
  Flavour:
```

```
  Spec:
    Characteristics:
      Architecture:
      Cpu: 4
      Ephemeral - Storage: 0
      Gpu: 0
      Memory: 16
      Persistent - Storage: 0
    Optional Fields:
      Availability: true
    Owner:
      domain: polito.fluidos.eu
      ip: 13.3.5.1
      nodeID: 91cbd32s0q1
    Policy:
      Aggregatable:
        Max Count: 5
        Min Count: 1
    Price:
      Amount: 10
      Currency: USD
      Period: hourly
    Provider ID: 05a2a55a−9939−4e94−9587−barlo14
    Type: k8s−fluidos
  Status:
    Creation Time: 2023−09−29T10:22:13+02:00
    Expiration Time: 2023−11−29T10:22:13+02:00
    Last Update Time: 2023−09−29T11:26:37+02:00
Partition:
  Cpu: 0
  Ephemeral Storage: 0
  Gpu: 0
  Memory: 0
  Storage: 0
Seller:
  domain: polito.fluidos.eu
  ip: 13.3.5.1
  nodeID: 91cbd32s0q1
```

7.5 Performance Testing

Tests presented in this section aim at a preliminary assessment of the performance and scalability of the solution. These tests were executed on a Linux virtual machine with 8 GB of RAM and 4 CPU cores. Clusters installed were **kind**.

In order to evaluate the performance of the solution, some time intervals have been analysed, in particular the ones between the following time instants:

- `LIST_FLAVOURS`
- `RESERVE_FLAVOUR`
- `PURCHASE_FLAVOUR`

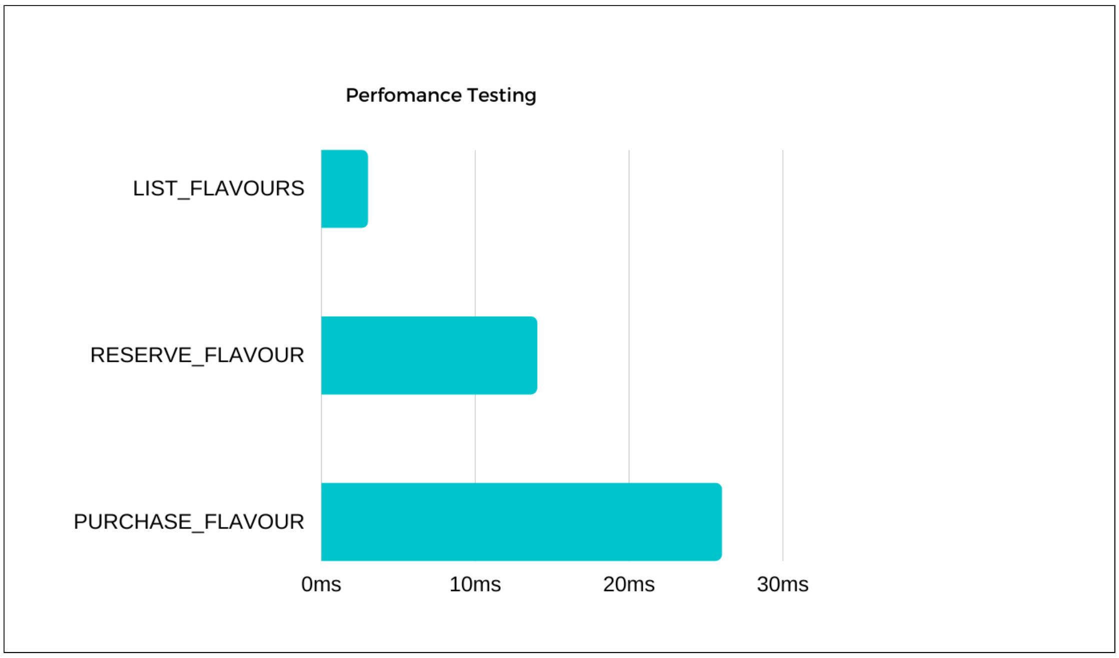

Figure 7.2: Performance testing for listing, reserving, and purchasing a flavor.

As we can see, the duration of a call, especially in the case of the `RESERVE_FLAVOUR` and `PURCHASE_FLAVOUR` operations, is influenced by various factors. These factors include the checks that are performed for each call.

In the case of the `RESERVE_FLAVOUR` operation, the call takes the specified time because, as observed, a check is conducted to determine whether it is possible to reserve the flavor. This involves the creation of a Reservation Custom Resource (CR) to ensure that the flavor can be allocated.

The `PURCHASE_FLAVOUR` operation, on the other hand, typically has a longer duration compared to standard HTTP calls. This is because, after the flavor is purchased, a contract is generated, which is then returned as a response to the call. This contract serves as proof of the flavor's purchase and will also be created on the consumer side.

These additional processes, such as the checks for reservation and the creation of contracts, contribute to the extended duration of these specific API calls in the FLUIDOS system.

7.6 Final Considerations

As we can see from the results, the system works as expected. The functional tests have been successful, and the system has been able to find a suitable flavour, reserve it, and buy it.

However, we can define some pros and cons of the system using this type of architecture (CRDs and CRs):

7.6.1 Pros and Cons of the system

Pros:

- **Native Kubernetes Integration:** CRDs are tightly integrated into Kubernetes, simplifying deployment and management.
- **Scalability:** CRDs are suitable for scaling resources within Kubernetes, leveraging auto-scaling capabilities.
- **Real-time Control:** CRDs allow real-time configuration changes and resource management without relying on external components, reducing response times and latency.

Cons:

- **Complexity:** Setting up and managing CRDs can be complex, especially for those unfamiliar with Kubernetes, leading to a steep learning curve.
- **Kubernetes Dependency:** Using CRDs ties the solution to Kubernetes, limiting portability to non-Kubernetes environments.

7.6.2 Possible Improvements

Efforts to enhance the use of CRDs could include:

- **Simplified Configuration:** Develop tools or frameworks to simplify the setup and management of CRDs for broader accessibility.
- **Cross-Platform Compatibility:** Explore options for making CRDs more compatible with non-Kubernetes environments to enhance portability.
- **Real-time Features:** Enhance real-time capabilities within CRDs to reduce the gap compared to messaging systems.

Chapter 8

Conclusions

In conclusion, this thesis has delved into the **FLUIDOS** project, which aims to revolutionize resource management in edge computing. We have examined key **FLUIDOS Node** components and their development, focusing on their role in communication and resource handling. Additionally, we introduced the **REAR Protocol**, designed to facilitate secure data exchange between cloud providers.

Throughout this research, we harnessed various technologies such as **Kubernetes**, **kubebuilder**, **NATS**, and **Docker**. These technologies were instrumental in addressing the challenges posed by the FLUIDOS project and driving innovation in the field of edge computing.

The FLUIDOS architecture, designed to be *flexible, scalable, secure, and decentralized*, breaks down traditional boundaries between edge and cloud computing. It supports multi-ownership and fluid topology, making it suitable for a wide range of use cases and domains.

The **REAR Protocol**, with its support for various resource types and standardized criteria selection, enables secure resource and capability exchange among cloud providers and customers.

The development of FLUIDOS components spanned two releases, with the first emphasizing **NATS** messaging and the second introducing **Kubernetes** resources and the **REAR** protocol for external communication. These components, including the *Local Resource Manager*, *Discovery Manager*, and *Contract Manager*, were adapted for Custom Resources.

Extensive testing, both in controlled environments and real-world **Kubernetes** clusters, validated FLUIDOS' core functionalities, adaptability, and scalability in edge computing scenarios. These tests demonstrated resource exchange, reservation, and contract creation within the FLUIDOS ecosystem.

In closing, this research contributes significantly to the evolution of FLUIDOS, advancing the concept of decentralized computing in edge environments. As we look ahead, FLUIDOS continues to grow and make a substantial impact on the landscape of edge computing. Future research and development efforts can continue to refine and expand FLUIDOS, making it a robust and indispensable tool for edge computing scenarios across various domains.

Bibliography

[1] Kubernetes official documentation, `https://kubernetes.io/docs/home/`

[2] Kubebuilder official documentation, `https://book.kubebuilder.io/`

[3] Controller-runtime official documentation, `https://pkg.go.dev/sigs.k8s.io/controller-runtime`

[4] Docker official documentation, `https://docs.docker.com/`

[5] Dockerfile official documentation, `https://docs.docker.com/engine/reference/builder/`

[6] Docker Compose official documentation, `https://docs.docker.com/compose/`

[7] Liqo official documentation, `https://docs.liqo.io/en/v0.5.0/`

[8] NATS official documentation, `https://docs.nats.io/`

[9] Booking Connectivity APIs, `https://connect.booking.com/user_guide/site/en-US/user_guide.html`

[10] Ticketmaster developer docs, `https://developer.ticketmaster.com/`

[11] Zhang, Lixia, et al. "RSVP: A new resource reservation protocol." IEEE network 7.5 (1993): 8-18

[12] "MRSVP: A resource reservation protocol for an integrated services network with mobile hosts." Wireless Networks 7 (2001): 5-19

[13] Awduche, Daniel, et al. RSVP-TE: extensions to RSVP for LSP tunnels. No. rfc3209. 2001

[14] Berger, Lou. Generalized multi-protocol label switching (GMPLS) signaling resource reservation protocol-traffic engineering (RSVP-TE) extensions. No. rfc3473. 2003

[15] Czajkowski, Karl, et al. "SNAP: A protocol for negotiating service level agreements and coordinating resource management in distributed systems." Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002 Revised Papers 8. Springer Berlin Heidelberg, 2002

[16] Wang, Xin, and Henning Schulzrinne. "RNAP: A resource negotiation and pricing protocol." Transit 6.B7 (1999): B8

[17] Venugopal, Srikumar, Xingchen Chu, and Rajkumar Buyya. "A negotiation mechanism for advance resource reservations using the alternate offers protocol." 2008 16th Interntional Workshop on Quality of Service. IEEE, 2008

[18] Andrieux, Alain, et al. "Web services agreement specification (WS-Agreement)." Open grid forum. Vol. 128. No. 1. 2007

[19] Smith, Reid G. "The contract net protocol: High-level communication and control in a distributed problem solver." IEEE Transactions on computers 29.12 (1980): 1104-1113

[20] Elmroth, Erik, and Johan Tordsson. "A grid resource broker supporting advance reservations and benchmark-based resource selection." International Workshop on Applied Parallel Computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004

[21] GSMA Operator Platform Telco Edge Requirements 2022 `https://www.gsma.com/futurenetworks/resources/gsma-operator-platform-telco-edge-requirements-2022/`

[22] GSMA Operator Platform Group – East-Westbound Interface APIs `https://www.gsma.com/futurenetworks/resources/east-westbound-interface-apis/`

[23] FLUIDOS official website, `https://www.fluidos.eu/public-deliverables/`

[24] FLUIDOS Architecture and Components docs, `https://github.com/fluidos-project/Docs`

[25] REAR Protocol docs, `https://github.com/topix-hackademy/Rear`