Grant Agreement No.: 101070473
Call: HORIZON-CL4-2021-DATA-01
Topic: HORIZON-CL4-2021-DATA-01-05
Type of action: HORIZON-RIA

D3.1 MODULAR AND EXTENSIBLE FLUIDOS NODE (V1)

Revision: 0.6

Work package	WP3
Task	All
Due date	30/11/2023
Submission date	22/12/2023
Deliverable lead	TOP-IX
Version	0.6
Authors	Lorenzo Moro (TOP-IX) Alessandro Cannarella (TOP-IX)

	Francesco Barletta (TOP-IX) Roberto Politi (TOP-IX) Stefano Braghin (IBM) George Kornaros (HMU) Marcello Coppola (STM) Francesco Cappa (POLITO) Stefano Galantino (POLITO)
Reviewers	Stefano Braghin (IBM) Emna Amri (CYSEC)

Abstract	The extension of Kubernetes to achieve a workload distribution across different multi-tenant clusters is definitely a challenge both on the administrative and technical aspects. FLUIDOS tries to fulfil this objective by creating an ecosystem capable of managing the interactions among Kubernetes clusters and the overlay technology which exploits this infrastructure to schedule and run tasks. This deliverable presents the results of the main activities of WP3, which concentrated on the implementation of the FLUIDOS Node and the definition of all the related aspects.
Keywords	Node, Kubernetes, Cluster

DOCUMENT REVISION HISTORY

Version	Date	Description of change	List of contributor(s)
0.1	15/01/2018	1st version of the template for comments	Margherita Facca (MARTEL)
0.2	23/10/2023	1st draft of D3.1 document	Lorenzo Moro (TOP-IX)
0.3	17/11/2023	Incorporated IBM contribution	Lorenzo Moro (TOP-IX), Stefano Braghin (IBM)
0.4	04/12/2023	Incorporate HMU/STM contribution	Lorenzo Moro, Roberto Politi (TOP-IX), George Kornaros (HMU), Marcello Coppola (STM)
0.5	18/12/2023	Ready for review	Lorenzo Moro, Roberto Politi (TOP-IX)
0.6	22/12/2023	Update based on IBM and CYSEC review	Stefano Braghin (IBM), Emna Amri (CYSEC)

Funded by Horizon Europe Framework Programme of the European Union

DISCLAIMER

COPYRIGHT NOTICE

Project co-funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	R	
Dissemination Level		
PU	Public, fully open, e.g. web	X
SEN	Sensitive, limited under the conditions of the Grant Agreement	
Classified R-UE/ EU-R	EU RESTRICTED under the Commission Decision No2015/ 444	
Classified C-UE/ EU-C	EU CONFIDENTIAL under the Commission Decision No2015/ 444	
Classified S-UE/ EU-S	EU SECRET under the Commission Decision No2015/ 444	

** R: Document, report (excluding the periodic and final reports)*

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

DATA: Data sets, microdata, etc

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

OTHER: Software, technical diagram, algorithms, models, etc.

Funded by Horizon Europe Framework Programme of the European Union

EXECUTIVE SUMMARY

This document provides a comprehensive description of the work carried out in Work Package 3 of the FLUIDOS project. It concentrates on the key aspects of the three tasks in which the activity has been divided, namely the node core, the ontology and the edge architectures.

The content of this document concentrates on the implementation choices, describing the software developed, the interactions among components, the designed data structures, the abstractions of resources and services collected into the ontology and the work carried out to cope with resource constrained edge devices.

A broader overview of the FLUIDOS ecosystem and a deep dive into the several architectural aspects is addressed in Work Package 2, while the other technical Work Packages (4, 5 and 6) manage other implementation aspects respectively of the orchestration algorithms, the security aspects and the energy related analysis.

Funded by Horizon Europe Framework Programme of the European Union

TABLE OF CONTENTS

Funded by Horizon Europe Framework Programme of the European Union

LIST OF FIGURES

Funded by Horizon Europe Framework Programme of the European Union

1 INTRODUCTION

The primary ambitious objective of FLUIDOS is the creation of a cloud-to-edge computing continuum. This requires the design and implementation of several software components on top of Kubernetes, the main virtualization and orchestration technology FLUIDOS relies on.

The technical work has been organized through several work packages, each focusing on a specific aspect: architecture (WP2), node (WP3), orchestration (WP4), security (WP5) and energy (WP6). Work Package 3 concentrates on three main objectives: (1) node core services and interfaces, (2) abstractions and models and (3) edge to cloud hierarchical architectures.

The first task is related to the implementation of the node core: much effort has been spent in the translation of the node architecture principles designed in Work Package 2 to a suitable implementation strategy, focused on the main software components required to establish the continuum. This software receives as an input a service request from the node orchestrator (which is being designed and developed in Work Package 4) and produces as an output a suitable physical or virtual Kubernetes node (belonging to a FLUIDOS Node) to schedule the job on.

The second task is related to the definition of objects, resource and service abstraction, which means the semantics of FLUIDOS. The inputs of this task are all the possible resources and services to be managed in the FLUIDOS ecosystem, from the simple computing resources (CPU, RAM, storage etc.) to sensors, specific hardware, algorithms etc. The output is a mapping of all these requirements and the creation of a comprehensive ontology to define them all.

Finally, the third task concentrates on the hierarchical architectures: this means spanning from Edge Nodes, which act on behalf of specialized hardware not capable of natively running Kubernetes, to simple Nodes and even Supernodes, acting as gateways of a FLUIDOS domain to other domains. These interactions are mediated through Catalogues, external servers acting both as a true marketplace to engage FLUIDOS services and as a relay to let different FLUIDOS domains discover each other and facilitate them exchange resources and services.

This report describes the implementation choices adopted to build the FLUIDOS Node, the designed ontology and the first work on the edge architectures.

Funded by Horizon Europe Framework Programme of the European Union

2 NODE CORE

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 modelled 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.

A Node is a set of software components that live on a Kubernetes cluster, whatever their kind (eg. Pod, DaemonSet, Service, Job etc.). Each Node embeds all the components, regardless of the role it takes on, while its behavior is role based. The components interact with each other exploiting several transmission systems according to the number of interactions and the messages to be exchanged: as a rule of thumb, local interactions are mediated by Kubernetes Controllers, while external interactions are managed through RESTful APIs.

The FLUIDOS Node has been designed and implemented in Go, leveraging Kubernetes native programming patterns and objects (CRDs, controllers, etcd etc.).

This section describes the FLUIDOS Node Core, the design choices taken and the implementation patterns adopted. The following picture represents the actual implementation, the functional elements, the custom resources and the related interactions of the FLUIDOS Node:

Funded by Horizon Europe Framework Programme of the European Union

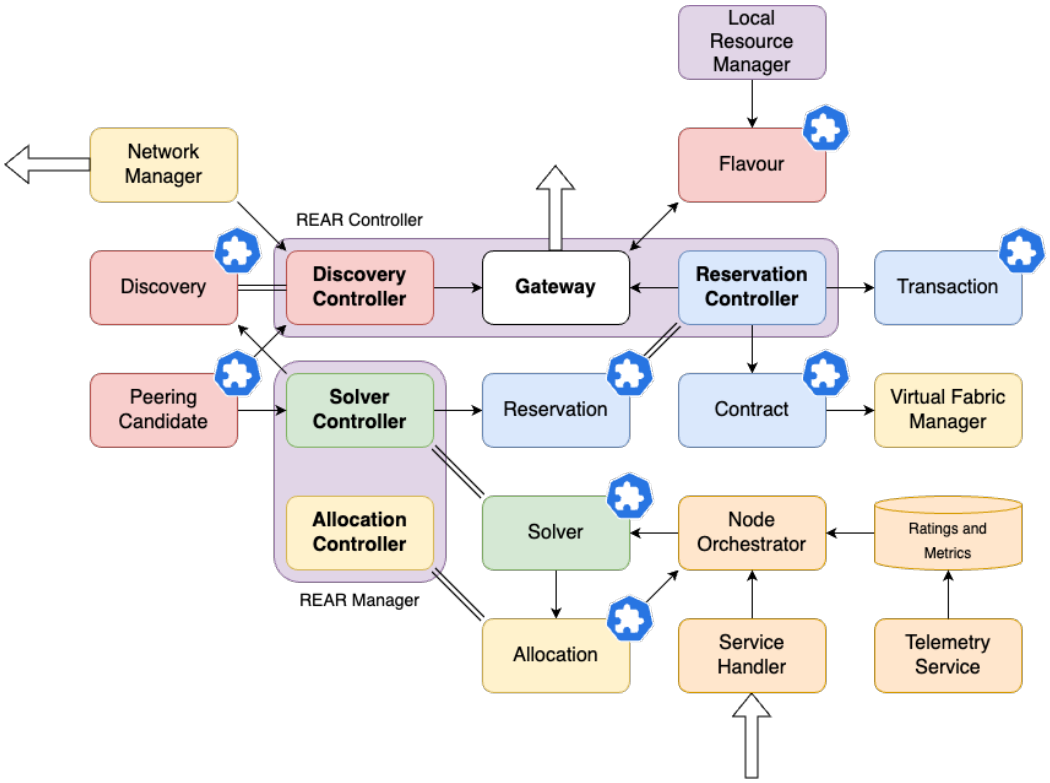

FIGURE 1: FLUIDOS NODE IMPLEMENTATION OVERVIEW

2.1 FUNCTIONAL ELEMENTS

FLUIDOS Node Core is base of several functional elements, each one responsible for a specific task:

- Local ResourceManager
- Available Resources
- Discovery Manager
  - Discovery Controller
- Peering Candidates
- REAR Manager
  - Solver Controller
  - Allocation Controller
- Contract Manager
  - Reservation Controller

Funded by Horizon Europe Framework Programme of the European Union

2.1.1 Local ResourceManager

The **Local Resource Manager** was constructed through the development of a Kubernetes controller. This controller serves the purpose of monitoring the internal resources of individual nodes within a FLUIDOS Node, representing a cluster. Subsequently, it generates a *Flavour Custom Resource (CR)* for each node and stores these CRs within the cluster for further management and utilization.

2.1.2 Available Resources

Available Resources component is a critical part of the FLUIDOS system responsible for managing and storing Flavours. It consists of two primary data structures:

- **Free Flavours**: This section is dedicated to storing Flavours, as defined in the REAR component.
- **Reserved Flavours**: In this section, objects or documents representing various resource allocations are stored, with each represented as a Flavour.

The primary function of Available Resources is to serve as a centralized repository for Flavours. When queried, it provides a list of these resources. Under the hood, Available Resources seamlessly integrates with Kubernetes' `etcd`, ensuring efficient storage and retrieval of data.

This component plays a crucial role in facilitating resource management within the FLUIDOS ecosystem, enabling efficient allocation and utilization of computing resources.

2.1.3 Discovery Manager

The **Discovery Manager** component within the FLUIDOS system serves as a critical part of the resource discovery process. It operates as a Kubernetes controller, continuously monitoring Discovery Custom Resources (CRs) generated by the Solver Controller.

The primary objectives of the Discovery Manager are as follows:

- **Populating Peering Candidates Table (Client)**: The Discovery Manager's primary responsibility is to populate the Peering Candidates table. It achieves this by identifying suitable resources known as "Flavours" based on the initial messages exchanged as part of the REAR protocol.
- **Discovery (Client)**: it initiates a LIST_FLAVOURS message, broadcasting it to all known list of FLUIDOS Nodes.
- **Offering Appropriate Flavours (Provider)**: In response to incoming requests, it will provide Flavours that best match the specific request.

Funded by Horizon Europe Framework Programme of the European Union

2.1.3.1 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 flavours that fits the `Discovery` selector.
2. If no flavours 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 other 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.

2.1.4 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 by the Discovery Manager.

Under the hood, Peering Candidates are stored through an appropriate Custom Resource.

2.1.5 REAR Manager

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

- if no Peering Candidates are found, it initiates the **Discovery**.
- optionally, if a suitable candidate is found, it triggers the **Reservation** phase.
- if this process is successfully fulfilled, resources are allocated, contracts are stored, and optionally can start the **Peering** phase.

2.1.5.1 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 book it.

Funded by Horizon Europe Framework Programme of the European Union

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 forge 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 a `EnstablishPeering` phase, it starts the peering process (to be implemented). Otherwise, it ends the process.

2.1.5.2 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.

Once the `Solver` has concluded the reservation and purchase of the resources, a new `Allocation` object is created in the Inactive status. If the Allocation is of type Node, this means the Controller is operating inside the Provider Node:

1. The previous `Flavour` is invalidated, eventually a new one is created detaching the right Partition from the old one, and the `Allocation` becomes Reserved.
2. If the ForeignCluster is Ready the `Allocation` can be set to Active else the Controller waits for the ForeignCluster to be Ready.
3. Once the ForeignCluster isn't Ready any more, the `Allocation` can be set to Released state.

If the Allocation is of type VirtualNode, this means the Controller is operating inside the Consumer Node:

1. The `Allocation` can is set to Reserved state.
2. If the ForeignCluster is Ready the `Allocation` can be set to Active else the Controller waits for the ForeignCluster to be Ready.
3. Once the ForeignCluster isn't Ready any more, the `Allocation` can be set to Released state.

Funded by Horizon Europe Framework Programme of the European Union

2.1.6 Contract Manager

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

- When a suitable peering candidate is identified and a Reservation is forged, the Contract Manager initiates the `Reserve` phase by sending a **RESERVE_FLAVOUR** message.
- Upon successful reservation of resources, it proceeds to the `Purchase` phase by sending a **PURCHASE_FLAVOUR** message. Following this, it stores the contract received.

2.1.6.1 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.

2.2 CUSTOM RESOURCES

A *resource* is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind; for example, the built-in *pods* resource contains a collection of Pod objects. A *custom resource* is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster

Funded by Horizon Europe Framework Programme of the European Union

itself. Once a custom resource is installed, users can create and access its objects using kubectl, just as they do for built-in resources like *Pods*.

On their own, custom resources let the developer store and retrieve structured data. By combining a custom resource with a *custom controller*, custom resources provide a true *declarative API*. The Kubernetes declarative API enforces a separation of responsibilities. The developer declares the desired state of the resource: the Kubernetes controller keeps the current state of Kubernetes objects in sync with the declared desired state.

Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The Operator pattern combines custom resources and custom controllers. Custom controllers can be used to encode domain knowledge for specific applications into an extension of the Kubernetes API.

In the FLUIDOS Node implementation, we've adopted the paradigm here described to manage the workflow, hence defining the following custom resources:

- Discovery
- Reservation
- Allocation
- Flavour
- Contract
- Peering Candidate
- Solver
- Transaction

2.2.1 Discovery

Here is a Discovery sample:

```
apiVersion: advertisement.fluidos.eu/v1alpha1
kind: Discovery
metadata:
  creationTimestamp: "2023-11-16T16:16:44Z"
   generation: 1
   name: discovery-solver-sample
   namespace: fluidos
   resourceVersion: "1593"
   uid: b084e36b-c9c0-4519-9cbe-7393d3b24cc9
 spec:
   selector:
     architecture: arm64
     rangeSelector:
      MaxCpu: "0"
      MaxEph: "0"
      MaxGpu: "0"
      MaxMemory: "0"
      MaxStorage: "0"
```


Funded by Horizon Europe Framework Programme of the European Union

```
      minCpu: "1"
      minEph: "0"
      minGpu: "0"
      minMemory: 1Gi
      minStorage: "0"
    type: k8s-fluidos
  solverID: solver-sample
  subscribe: false
status:
  peeringCandidate:
    name: peeringcandidate-fluidos.eu-k8s-fluidos-bba29928
    namespace: fluidos
  phase:
    lastChangeTime: "2023-11-16T16:16:44Z"
    message: 'Discovery Solved: Peering Candidate found'
    phase: Solved
    startTime: "2023-11-16T16:16:44Z"
```

2.2.2 Reservation

Here is a Reservation sample

```
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Reservation
metadata:
  creationTimestamp: "2023-11-16T16:16:44Z"
  generation: 1
  name: reservation-solver-sample
  namespace: fluidos
  resourceVersion: "1606"
  uid: fb6ad873-7b51-4946-a016-bfdbcff0d2e4
spec:
  buyer:
    domain: fluidos.eu
    ip: 172.18.0.2:30000
    nodeID: jlhfplohpf
  partition:
    architecture: arm64
    cpu: "1"
    ephemeral-storage: "0"
    gpu: "0"
    memory: 1Gi
    storage: "0"
  peeringCandidate:
    name: peeringcandidate-fluidos.eu-k8s-fluidos-bba29928
    namespace: fluidos
  purchase: true
  reserve: true
  seller:
    domain: fluidos.eu
    ip: 172.18.0.7:30001
    nodeID: 46ltws9per
  solverID: solver-sample
status:
  contract:
    name: contract-fluidos.eu-k8s-fluidos-bba29928-3c6e
    namespace: fluidos
  phase:
    lastChangeTime: "2023-11-16T16:16:45Z"
    message: Reservation solved
    phase: Solved
```


Funded by Horizon Europe Framework Programme of the European Union

```
    startTime: "2023-11-16T16:16:44Z"
  purchasePhase: Solved
  reservePhase: Solved
  transactionID: 37b53e2ba2b7b6ecb96bd56989baecf2-1700151404800502383
```

2.2.3 Allocation

Here is an Allocation sample:

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Allocation
metadata:
  creationTimestamp: "2023-11-16T16:16:45Z"
  generation: 1
  name: allocation-fluidos.eu-k8s-fluidos-bba29928-3c6e
  namespace: fluidos
  resourceVersion: "1716"
  uid: 87894f6c-24a9-4389-bd90-a9e9641aa337
spec:
  destination: Local
  flavour:
    metadata:
      name: fluidos.eu-k8s-fluidos-bba29928
      namespace: fluidos
    spec:
      characteristics:
        architecture: ""
        cpu: 7970838142n
        ephemeral-storage: "0"
        gpu: "0"
        memory: 7879752Ki
        persistent-storage: "0"
      optionalFields:
        availability: true
        workerID: fluidos-provider-worker
      owner:
        domain: fluidos.eu
        ip: 172.18.0.7:30001
        nodeID: 46ltws9per
      policy:
        aggregatable:
          maxCount: 0
          minCount: 0
        partitionable:
          cpuMin: "0"
          cpuStep: "1"
          memoryMin: "0"
          memoryStep: 100Mi
      price:
        amount: ""
        currency: ""
        period: ""
      providerID: 46ltws9per
      type: k8s-fluidos
    status:
      creationTime: ""
      expirationTime: ""
      lastUpdateTime: ""
  intentID: solver-sample
  nodeName: liqo-fluidos-provider
  partitioned: true
```


Funded by Horizon Europe Framework Programme of the European Union

```
  remoteClusterID: 40585c34-4b93-403f-8008-4b1ecced6f62
  resources:
    architecture: ""
    cpu: "1"
    ephemeral-storage: "0"
    gpu: "0"
    memory: 1Gi
    persistent-storage: "0"
  type: VirtualNode
status:
  lastUpdateTime: "2023-11-16T16:16:49Z"
  message: Outgoing peering ready, Allocation is now Active
  status: Active
```

2.2.4 Flavour

Here is a Flavour sample:

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Flavour
metadata:
  creationTimestamp: "2023-11-16T16:13:52Z"
  generation: 2
  name: fluidos.eu-k8s-fluidos-bba29928
  namespace: fluidos
  resourceVersion: "1534"
  uid: 5ce9f378-014e-4cb4-b173-5a3530d8f78d
spec:
  characteristics:
    architecture: arm64
    cpu: 7970838142n
    ephemeral-storage: "0"
    gpu: "0"
    memory: 7879752Ki
    persistent-storage: "0"
  optionalFields:
    workerID: fluidos-provider-worker
  owner:
    domain: fluidos.eu
    ip: 172.18.0.7:30001
    nodeID: 46ltws9per
  policy:
    aggregatable:
      maxCount: 0
      minCount: 0
    partitionable:
      cpuMin: "0"
      cpuStep: "1"
      memoryMin: "0"
      memoryStep: 100Mi
  price:
    amount: ""
    currency: ""
    period: ""
  providerID: 46ltws9per
  type: k8s-fluidos
```


Funded by Horizon Europe Framework Programme of the European Union

2.2.5 Contract

Here is a Contract sample:

```
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Contract
metadata:
  creationTimestamp: "2023-11-16T16:16:45Z"
  generation: 1
  name: contract-fluidos.eu-k8s-fluidos-bba29928-3c6e
  namespace: fluidos
  resourceVersion: "1531"
  uid: baa27f94-ebd8-4e7b-98fe-3e7d5b09d4bb
spec:
  buyer:
    domain: fluidos.eu
    ip: 172.18.0.2:30000
    nodeID: jlhfplohpf
  buyerClusterID: 14461b0e-446d-4b05-b1f8-9ddb6765ac02
  expirationTime: "2024-11-15T16:16:45Z"
  flavour:
    apiVersion: nodecore.fluidos.eu/v1alpha1
    kind: Flavour
    metadata:
      name: fluidos.eu-k8s-fluidos-bba29928
      namespace: fluidos
    spec:
      characteristics:
        architecture: arm64
        cpu: 7970838142n
        ephemeral-storage: "0"
        gpu: "0"
        memory: 7879752Ki
        persistent-storage: "0"
      optionalFields:
        availability: true
        workerID: fluidos-provider-worker
      owner:
        domain: fluidos.eu
        ip: 172.18.0.7:30001
        nodeID: 46ltws9per
      policy:
        aggregatable:
          maxCount: 0
          minCount: 0
        partitionable:
          cpuMin: "0"
          cpuStep: "1"
          memoryMin: "0"
          memoryStep: 100Mi
      price:
        amount: ""
        currency: ""
        period: ""
      providerID: 46ltws9per
      type: k8s-fluidos
    status:
      creationTime: ""
      expirationTime: ""
      lastUpdateTime: ""
  partition:
    architecture: ""
```


Funded by Horizon Europe Framework Programme of the European Union

```
      cpu: "1"
      ephemeral-storage: "0"
      gpu: "0"
      memory: 1Gi
      storage: "0"
    seller:
      domain: fluidos.eu
      ip: 172.18.0.7:30001
      nodeID: 46ltws9per
    sellerCredentials:
      clusterID: 40585c34-4b93-403f-8008-4b1ecced6f62
      clusterName: fluidos-provider
      endpoint: https://172.18.0.6:31780
      token:
0959ee7a6290b51cb223f971e30455043a1af01b383b5dc8cd13650a4d61e9b6184c524fc56699d427b
37044fa1e3b05179ecf19f807aa67d26fcaa1cebe4d68
    transactionID: 37b53e2ba2b7b6ecb96bd56989baecf2-1700151404800502383
```

2.2.6 PeeringCandidate

Here is a PeeringCandidate sample:

```
apiVersion: advertisement.fluidos.eu/v1alpha1
kind: PeeringCandidate
metadata:
  creationTimestamp: "2023-11-16T16:16:44Z"
  generation: 1
  name: peeringcandidate-fluidos.eu-k8s-fluidos-bba29928
  namespace: fluidos
  resourceVersion: "1592"
  uid: 030c441f-a17e-4778-9515-9cd4293f656a
spec:
  flavour:
    metadata:
      name: fluidos.eu-k8s-fluidos-bba29928
      namespace: fluidos
    spec:
      characteristics:
        architecture: ""
        cpu: 7970838142n
        ephemeral-storage: "0"
        gpu: "0"
        memory: 7879752Ki
        persistent-storage: "0"
      optionalFields:
        availability: true
        workerID: fluidos-provider-worker
      owner:
        domain: fluidos.eu
        ip: 172.18.0.7:30001
        nodeID: 46ltws9per
      policy:
        aggregatable:
          maxCount: 0
          minCount: 0
        partitionable:
          cpuMin: "0"
          cpuStep: "1"
          memoryMin: "0"
          memoryStep: 100Mi
      price:
```


Funded by Horizon Europe Framework Programme of the European Union

```
          amount: ""
          currency: ""
          period: ""
        providerID: 46ltws9per
        type: k8s-fluidos
      status:
        creationTime: ""
        expirationTime: ""
        lastUpdateTime: ""
    reserved: true
    solverID: solver-sample
```

2.2.7 Solver

Here is a Solver sample:

```
  apiVersion: nodecore.fluidos.eu/v1alpha1
  kind: Solver
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"nodecore.fluidos.eu/v1alpha1","kind":"Solver","metadata":{"annotatio
ns":{},"name":"solver-
sample","namespace":"fluidos"},"spec":{"enstablishPeering":true,"findCandidate":tru
e,"intentID":"intent-
sample","reserveAndBuy":true,"selector":{"architecture":"arm64","rangeSelector":{"m
inCpu":"1000m","minMemory":"1Gi"},"type":"k8s-fluidos"}}}
    creationTimestamp: "2023-11-16T16:16:44Z"
    generation: 1
    name: solver-sample
    namespace: fluidos
    resourceVersion: "1717"
    uid: 392ae554-69e4-4639-90ef-34d8f3a83aef
  spec:
    enstablishPeering: true
    findCandidate: true
    intentID: intent-sample
    reserveAndBuy: true
    selector:
      architecture: arm64
      rangeSelector:
        minCpu: 1000m
        minMemory: 1Gi
      type: k8s-fluidos
  status:
    allocation:
      name: allocation-fluidos.eu-k8s-fluidos-bba29928-3c6e
      namespace: fluidos
    contract:
      name: contract-fluidos.eu-k8s-fluidos-bba29928-3c6e
      namespace: fluidos
    credentials:
      clusterID: 40585c34-4b93-403f-8008-4b1ecced6f62
      clusterName: fluidos-provider
      endpoint: https://172.18.0.6:31780
      token:
0959ee7a6290b51cb223f971e30455043a1af01b383b5dc8cd13650a4d61e9b6184c524fc56699d427b
37044fa1e3b05179ecf19f807aa67d26fcaa1cebe4d68
    discoveryPhase: Solved
    findCandidate: Solved
```


Funded by Horizon Europe Framework Programme of the European Union

```
  peering: Solved
  peeringCandidate:
    name: peeringcandidate-fluidos.eu-k8s-fluidos-bba29928
    namespace: fluidos
  reservationPhase: Solved
  reserveAndBuy: Solved
  solverPhase:
    endTime: "2023-11-16T16:16:49Z"
    lastChangeTime: "2023-11-16T16:16:49Z"
    message: Solver has enstablished a peering
    phase: Solved
```

2.2.8 Transaction

Here is a Transaction sample:

```
apiVersion: reservation.fluidos.eu/v1alpha1
kind: Transaction
metadata:
  creationTimestamp: "2023-11-16T16:16:44Z"
  generation: 1
  name: 37b53e2ba2b7b6ecb96bd56989baecf2-1700151404800502383
  namespace: fluidos
  resourceVersion: "1600"
  uid: a1d73148-0795-4061-80cf-197e6a83379c
spec:
  buyer:
    domain: fluidos.eu
    ip: 172.18.0.2:30000
    nodeID: jlhfplohpf
  clusterID: 14461b0e-446d-4b05-b1f8-9ddb6765ac02
  flavourID: fluidos.eu-k8s-fluidos-bba29928
  partition:
    architecture: ""
    cpu: "1"
    ephemeral-storage: "0"
    gpu: "0"
    memory: 1Gi
    storage: "0"
  startTime: "2023-11-16T16:16:44Z"
```

2.3 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.

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

Funded by Horizon Europe Framework Programme of the European Union

- **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).

2.3.1 State of the art

This section introduces the state of the art for the REAR protocol, analysing both commercial solutions and research proposals. We will present how existing companies tackle the problem of resource acquisition workflows, hence showing which solutions are adopted in real use cases. This can be used to gain insights into how, different companies such as Booking.com, 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.

Booking.com

The Booking.com Connectivity APIs enable to send and retrieve data for properties listed on Booking.com. It is possible to manage room availability, reservations, prices, and many other things [1].

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.

Funded by Horizon Europe Framework Programme of the European Union

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[1] (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[2] (/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:

1 https://connect.booking.com/user_guide/site/en-US/reservations-api/reservations-process-ota/

2 https://connect.booking.com/user_guide/site/en-US/reservations-api/reservations-process-bxml/

Funded by Horizon Europe Framework Programme of the European Union

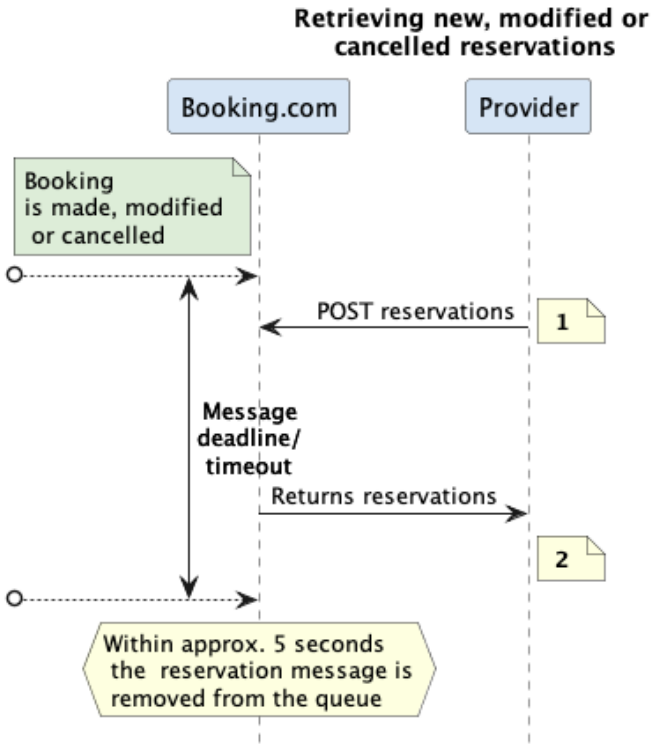

FIGURE 2: RETRIEVING NEW, MODIFIED OR CANCELLED RESERVATIONS

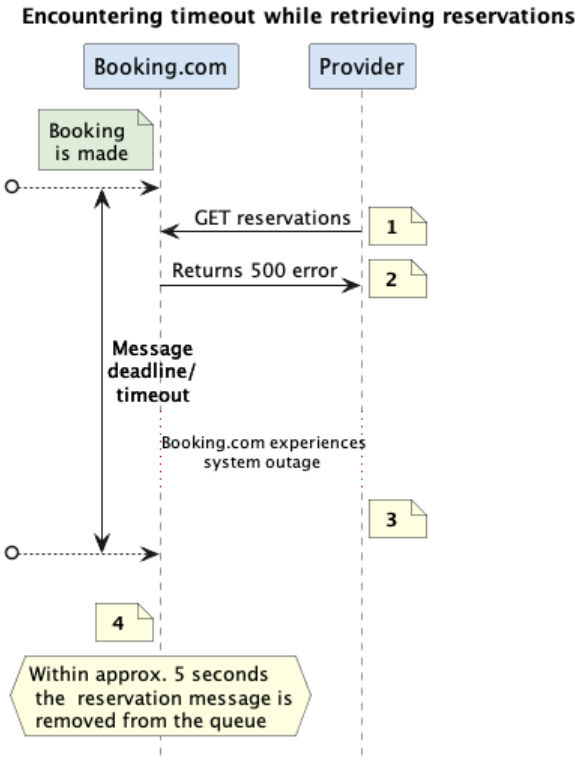

FIGURE 3: ENCOUNTERING TIMEOUT WHILE RETRIEVING RESERVATIONS

Funded by Horizon Europe Framework Programme of the European Union

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 catalogue, 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 retrieve ticket and event information [2].

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:

Funded by Horizon Europe Framework Programme of the European Union

- 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:

Funded by Horizon Europe Framework Programme of the European Union

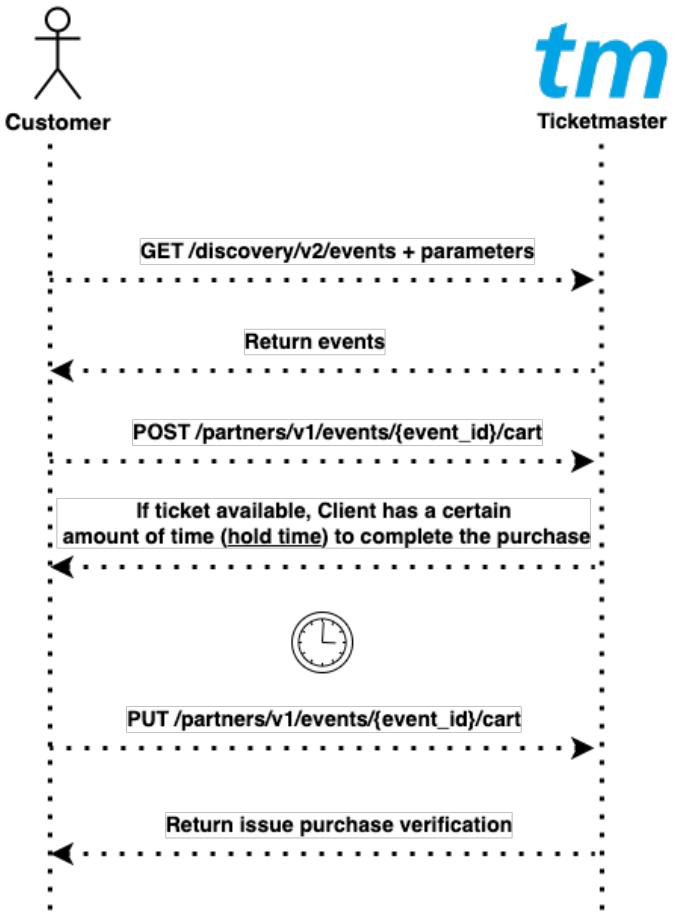

FIGURE 4: SEARCH FOR AN EVENT AND BUY A TICKET

Research Solutions

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 protocols in computer networks is RSVP (Resource Reservation Protocol [3]). 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 limitations MRSVP [4] has been proposed, allowing mobile devices to perform reservations not only for the current location, but also for future locations. In addition, [5] extends the problem formulation, including the price of networking resources,

Funded by Horizon Europe Framework Programme of the European Union

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 signalling in a wide area network [6][7].

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 [8] 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, [9] 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 [10] also describe a brokering architecture that is able to make advance resource reservations and create SLAs using the WS-Agreement standard [11], based on the Contract Net protocol for negotiating SLAs [12].

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 [13], and to the definition of a suitable protocol to allow seamless application deployment across different Telco providers [14]. 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 applications 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.

2.3.2 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

Funded by Horizon Europe Framework Programme of the European Union

computing and/or service instances). The figure below depicts a possible interaction between a customer and a provider using the REAR protocol.

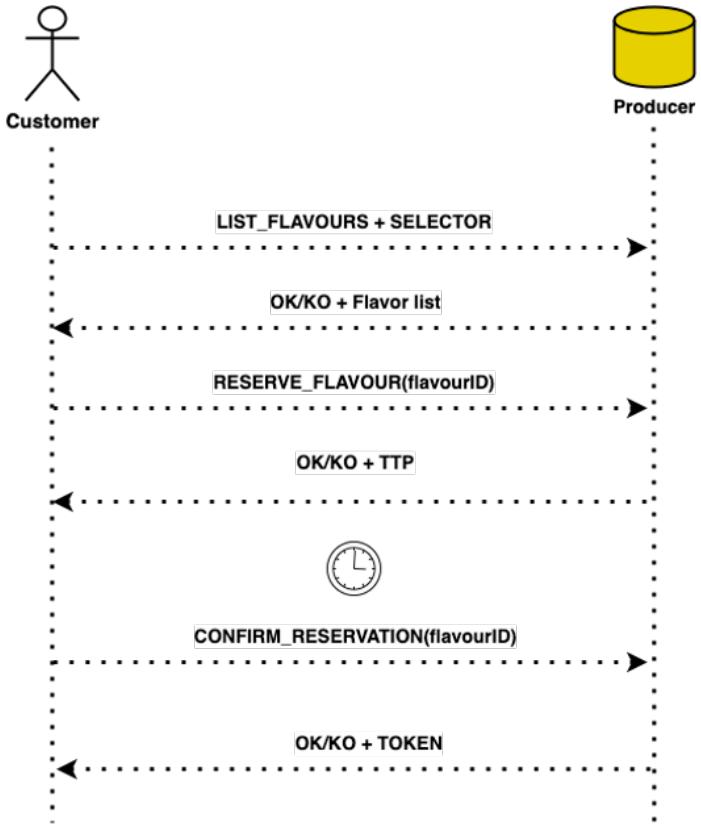

FIGURE 5: INTERACTION BETWEEN CLIENT AND PROVIDER USING THE REQUIRED MESSAGES

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.

The protocol comprises various messages, which can be classified as either *required* or *optional*:

- Required:
  - LIST_FLAVOURS, sent by the customer to probe the available flavours offered by a given provider;
  - RESERVE_FLAVOUR, sent by the customer to inform the provider about its willingness to reserve a specific flavour;
  - CONFIRM_RESERVATION, sent by the customer to complete the purchase of an offered flavour;
- Optional:
  - REFRESH_FLAVOUR, 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;

Funded by Horizon Europe Framework Programme of the European Union

- WITHDRAW_FLAVOUR, 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.

Get the list of available flavour

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 flavours 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.

Funded by Horizon Europe Framework Programme of the European Union

- 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.

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 Flavours. 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).

Reserve a 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 to handle the concurrency in client requests, as different clients may be interested in the same flavour. Note that this message only notifies the provider the intention of purchasing a flavour, the request must then be finalized using the confirm purchase message (see following subsection).

Specifically, the client/provider interaction can be summarized in the following:

- 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 irresponsive, or the subsequent purchase process exceeds a predefined threshold.

Funded by Horizon Europe Framework Programme of the European Union

The following figure 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.

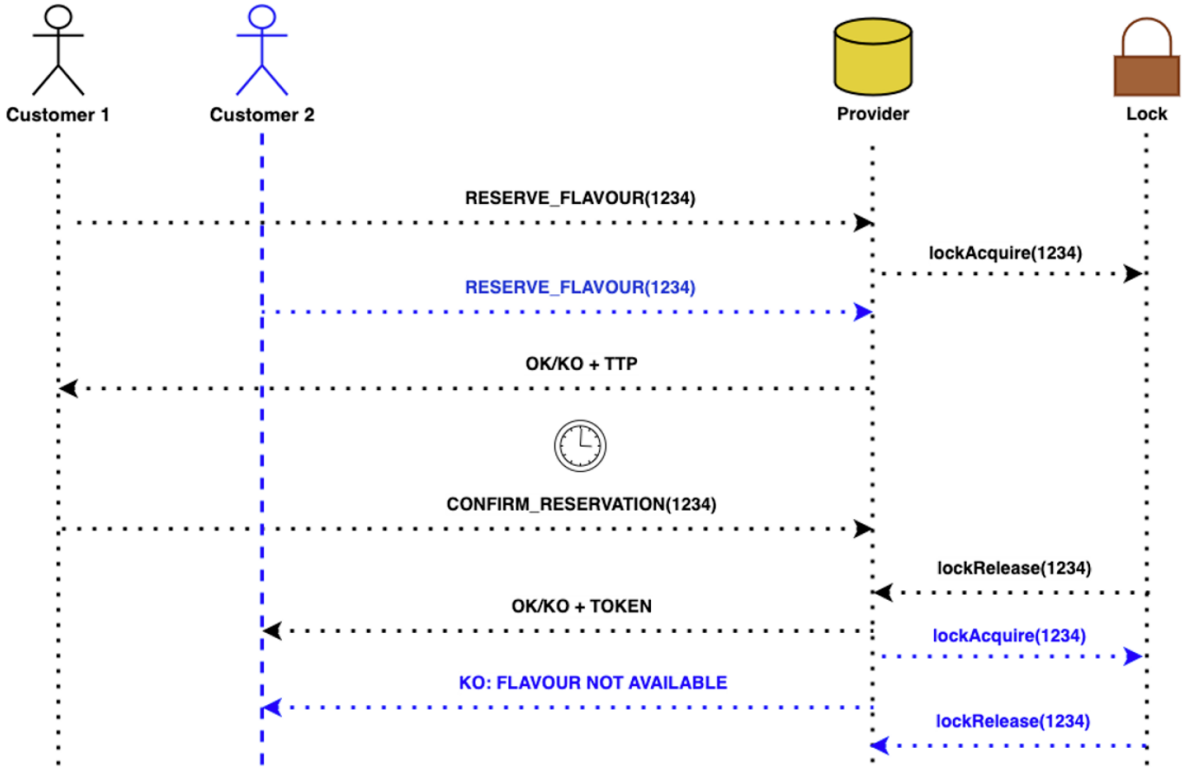

FIGURE 6: CONCURRENT FLAVOUR ACCESS FROM TWO DIFFERENT CLIENTS

Funded by Horizon Europe Framework Programme of the European Union

Confirm a Reservation

The **CONFIRM_RESERVATION** message is transmitted by the client following the receipt of the provider's response during the reservation phase. In this context, the client must confirm the purchase within the timeframe defined as the **Time To Purchase (TTP)**, as outlined in previous figure.

Subscribe to changes

REAR defines a set of optional messages that extend the expressiveness of the protocol, and we summarize 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.

The following figure 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 the intention to receive continuous updates on a specific flavour, using the SUBSCRIBE_FLAVOUR message, that is mapped onto an appropriate request message which 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 provided sends asynchronous updates over the created channel to the client for the specified flavour. We define two different types of updates:
  - 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 the purchase.

Funded by Horizon Europe Framework Programme of the European Union

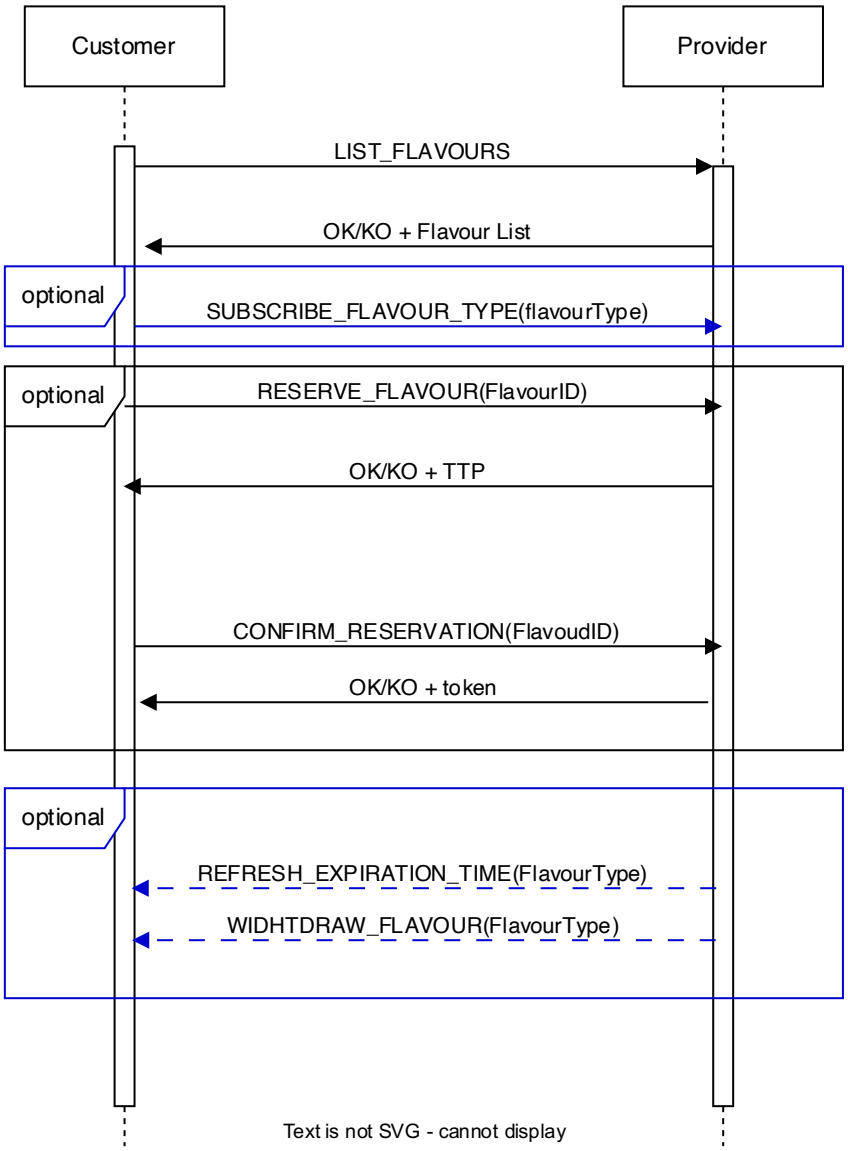

FIGURE 7: CONCURRENT FLAVOUR ACCESS FROM TWO DIFFERENT CLIENTS

2.3.3 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:
  - **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

Funded by Horizon Europe Framework Programme of the European Union

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.

2.3.3.1 Required messages

LIST_FLAVOURS

Request body

None

Response body

```
Items: [
  Flavor: {
            # FlavorID is the ID of the Flavor
            FlavorID string

            # ProviderID is the ID of the provider of this Flavor.
            ProviderID string

            # Type is the type of the Flavor
            FlavorType string

            # Characteristics contains the characteristics of the Flavor.
            Characteristics Characteristics

            # Policy contains the policy of the Flavor.
            Policy Policy

            # Owner contains the identity info of the owner of the Flavor.
            Owner NodeIdentity

            # Price contains the price model of the Flavor.
            Price Price

            # Optional fields that can be retrieved from the Flavor.
            OptionalFields OptionalFields

  }
]
```


Funded by Horizon Europe Framework Programme of the European Union

```
  Characteristics: {
    # Architecture is the architecture of the Flavor.
    Architecture string

    # Cpu is the number of CPU cores of the Flavor.
    Cpu int

    # Memory is the amount of RAM of the Flavor.
    Memory int

    # Gpu is the number of GPU cores of the Flavor.
    Gpu int

    # EphemeralStorage is the amount of ephemeral storage of the Flavor.
    EphemeralStorage int

    # PersistentStorage is the amount of persistent storage of the Flavor.
    PersistentStorage int
  }


  Price: {
    # Amount is the amount of the price.
    Amount string

    # Currency is the currency of the price.
    Currency string

    # Period is the period of the price.
    Period string
  }


  Policy: {
    Partitionable: {
      # CpuMin is the minimum requirable number of CPU cores of the Flavor.
      CpuMin int `json:"cpuMin"`

      # MemoryMin is the minimum requirable amount of RAM of the Flavor.
      MemoryMin int `json:"memoryMin"`

      # CpuStep is the incremental value of CPU cores of the Flavor.
      CpuStep int `json:"cpuStep"`

      # MemoryStep is the incremental value of RAM of the Flavor.
      MemoryStep int `json:"memoryStep"`
    },
    Aggregatable: {
      # MinCount is the minimum requirable number of instances of the Flavor.
      MinCount int

      # MaxCount is the maximum requirable number of instances of the Flavor.
      MaxCount int
    }
  }


  NodeIdentity: {
    # Domain is the domain of the node.
    Domain string

    # NodeID is the ID of the node.
    NodeID string
```


Funded by Horizon Europe Framework Programme of the European Union

```
  # IP is the IP of the node.
  IP     string
}


OptionalFields: {
    # Availability is the availability flag of the Flavor
    Availability bool

    # WorkerID is the ID of the worker that provides the Flavor.
    WorkerID string
}
```

LIST_FLAVOURS + Selector

Request body

```
Selector: {
    # FlavorType specifies the type of Flavor.
    FlavorType string

    # Architecture specifies the architecture of the resource.
    Architecture string

    # Cpu is the exact desired CPU quantity.
    Cpu int

    # Memory is the exact desired memory quantity.
    Memory int

    # EphemeralStorage is the exact desired ephemeral storage quantity.
    EphemeralStorage int

    # MoreThanCpu specifies the minimum CPU quantity desired.
    MoreThanCpu int

    # MoreThanMemory specifies the minimum memory quantity desired.
    MoreThanMemory int

    # MoreThanEph specifies the minimum ephemeral storage quantity desired.
    MoreThanEph int

    # LessThanCpu specifies the maximum CPU quantity desired.
    LessThanCpu int

    # LessThanMemory specifies the maximum memory quantity desired.
    LessThanMemory int

    # LessThanEph specifies the maximum ephemeral storage quantity desired.
    LessThanEph int
}
```

Response body

```
Flavor: {
  # FlavorID is the ID of the Flavor
  FlavorID string

  # ProviderID is the ID of the provider of this Flavor.
```


Funded by Horizon Europe Framework Programme of the European Union

```
    ProviderID string

    # Type is the type of the Flavor
    FlavorType string

    # Characteristics contains the characteristics of the Flavor.
    Characteristics Characteristics

    # Policy contains the policy of the Flavor.
    Policy Policy

    # Owner contains the identity info of the owner of the Flavor.
    Owner NodeIdentity

    # Price contains the price model of the Flavor.
    Price Price

    # Optional fields that can be retrieved from the Flavor.
    OptionalFields 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:

Request body

```
Reservation: {
    # FlavorID specifies the ID of the Flavor to be reserved
    FlavorID string

    # Buyer is the buyer Identity of the Fluidos Node that is reserving the
    # Flavor
    Buyer NodeIdentity
}
```

Response body

```
Transaction: {
    # TransactionID is the ID of the Transaction linked to the Reservation
    TransactionID string

    # FlavorID is the ID of the Flavor that is being reserved
    FlavorID string

    # Buyer is the buyer Identity of the Fluidos Node that is reserving the
    # Flavor
    Buyer NodeIdentity

    # StartTime is the time at which the reservation should start
    StartTime string
}
```


Funded by Horizon Europe Framework Programme of the European Union

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.

Request body

```
Purchase: {
  # TransactionID is a unique identifier for the transaction.
  TransactionID string
}
```

Response body

```
<empty> Status: 200 OK
```

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. An example could be:

```
Contract: {
  ContractID string
  Flavor Flavor
  Buyer string
  Seller string
  Credentials LiqoCredentials
}
```

2.3.3.2 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

The 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.

Funded by Horizon Europe Framework Programme of the European Union

```
<RefreshMessage>
  <Flavor>
      <!-- Details of the Flavor object that has been refreshed -->
      <FlavorID>string</FlavorID>
      <ProviderID>string</ProviderID>
      <FlavorType>string</FlavorType>
      <!-- Other Flavor fields like Characteristics, Policy, Owner, Price -->
  </Flavor>
  <ModificationDetails>
      <!-- Details of the changes made to the Flavor -->
      <FieldModified>string</FieldModified>
      <OldValue>string</OldValue>
      <NewValue>string</NewValue>
      <!-- It is possibile to add other fields if needed -->
  </ModificationDetails>
</RefreshMessage>
```

WITHDRAW_FLAVOUR

The 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.

```
<WithdrawMessage>
  <Flavor>
      <!-- Details of the Flavor object that is no longer available -->
      <FlavorID>string</FlavorID>
      <ProviderID>string</ProviderID>
      <FlavorType>string</FlavorType>
      <!-- Other Flavor fields like Characteristics, Policy, Owner, Price -->
  </Flavor>
  <Reason>
      <!-- Reason for the withdrawal of the Flavor offer -->
      <Message>string</Message>
      <!-- It is possibile to add other fields for detailed reasons if needed -->
  </Reason>
</WithdrawMessage>
```


Funded by Horizon Europe Framework Programme of the European Union

2.3.4 Implementation

The Discovery Controller implements a state machine to manage all the operations and phases involved. The following figure shows the state diagram for the global discovery process.

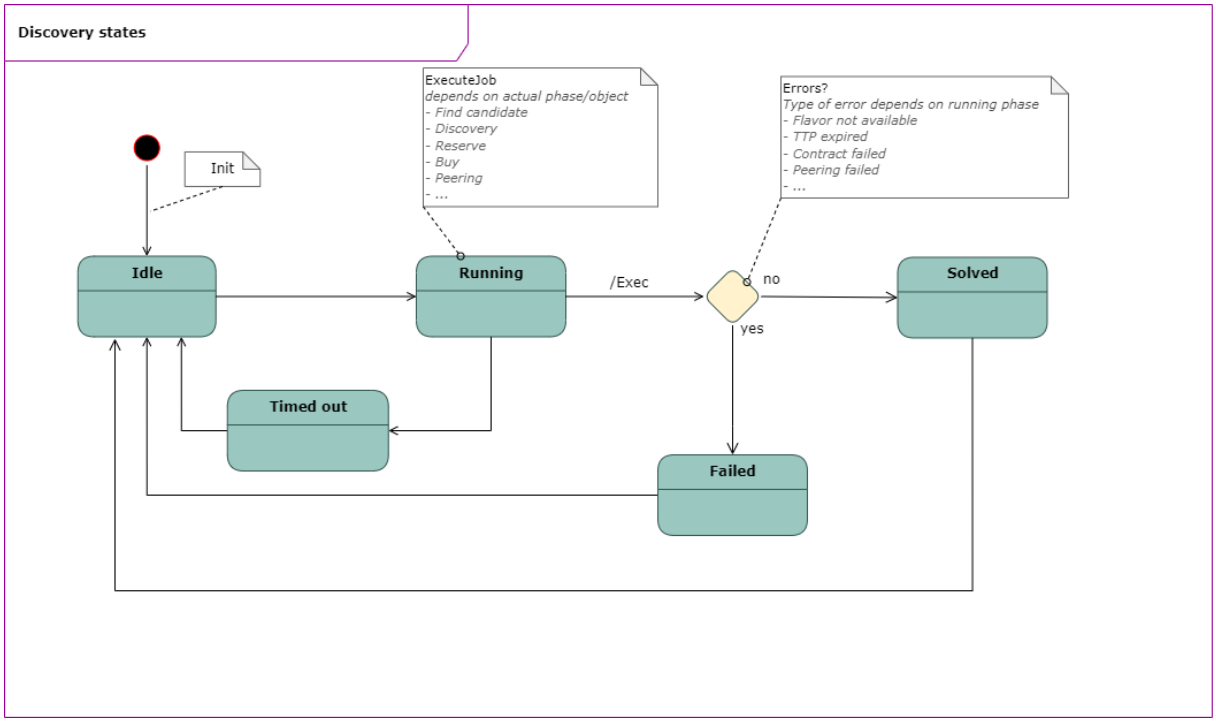

FIGURE 8: DISCOVERY STATE DIAGRAMS

- The three main states are `idle` (starting state), `solved` (final/success state) and `running` (when the actual discovery is in progress).
- The system also uses two more states (`failed` and `timed out`) to manage any error that may occur during the discovery process.

For example, the discovery may transition into a state error if:

- the whole discovery process has expired before finding a candidate;
- no flavours are found to fulfil the consumer requests;
- another consumer has purchased the found flavour (i.e., the flavour is not available anymore);
- any other problem/error.

The discovery process is implemented in a class called `Solver`, that implements all the logic to try to "solve" all the discovery phases (from the research of the resources, to the reservation, purchase and creation and instantiation of the node).

The whole discovery process however is composed of several subtasks, each one representing a particular phase.

Funded by Horizon Europe Framework Programme of the European Union

The main phases (shown in the following figure) are:

- `FindCandidate`: during this phase, the discovery system finds a candidate node that fulfil the consumer requirements (the flavour);
- `ReserveAndBuy`: this phase starts if a flavour is found and the consumer decides to proceed with the reservation and purchase. This phase also manages all the possible conflicts to avoid the same node to be reserved/purchased by more than one consumer;
- `Consume`: in this phase the node is actually instantiated and deployed on the cluster, and is made available for usage.

Any error that may occur during each phase lead to an error that move all the discovery phase to an error state

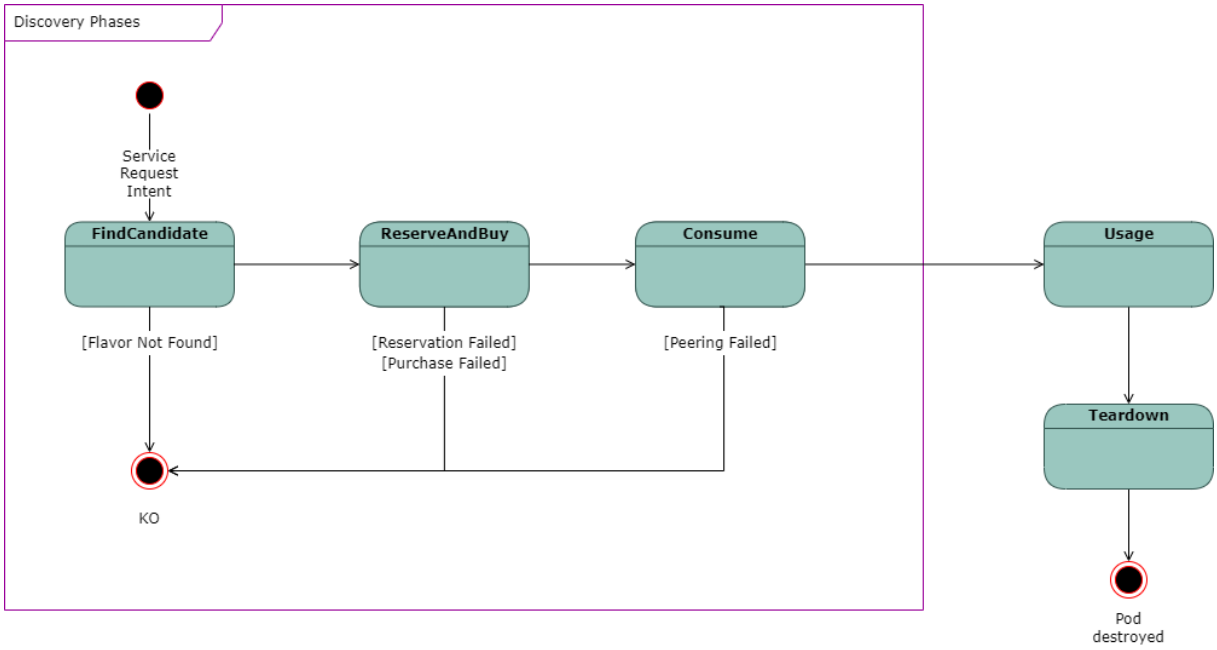

FIGURE 9: DISCOVERY CONTROLLER PHASES

After the discovery has been completed, the node has been created/deployed, and is ready for usage.

Optionally, if not needed anymore, the consumer can request a teardown operation to destroy the node and release the resources. These two phases have been represented in the picture to increase the comprehension of the mechanism, even if they are not part of the discovery process.

Each discovery phase is implemented with a particular object/class that internally implements another state machine, and each phase-level state machine follows the same diagram of the main state machine as the discovery controller, as visible in the following figure.

Funded by Horizon Europe Framework Programme of the European Union

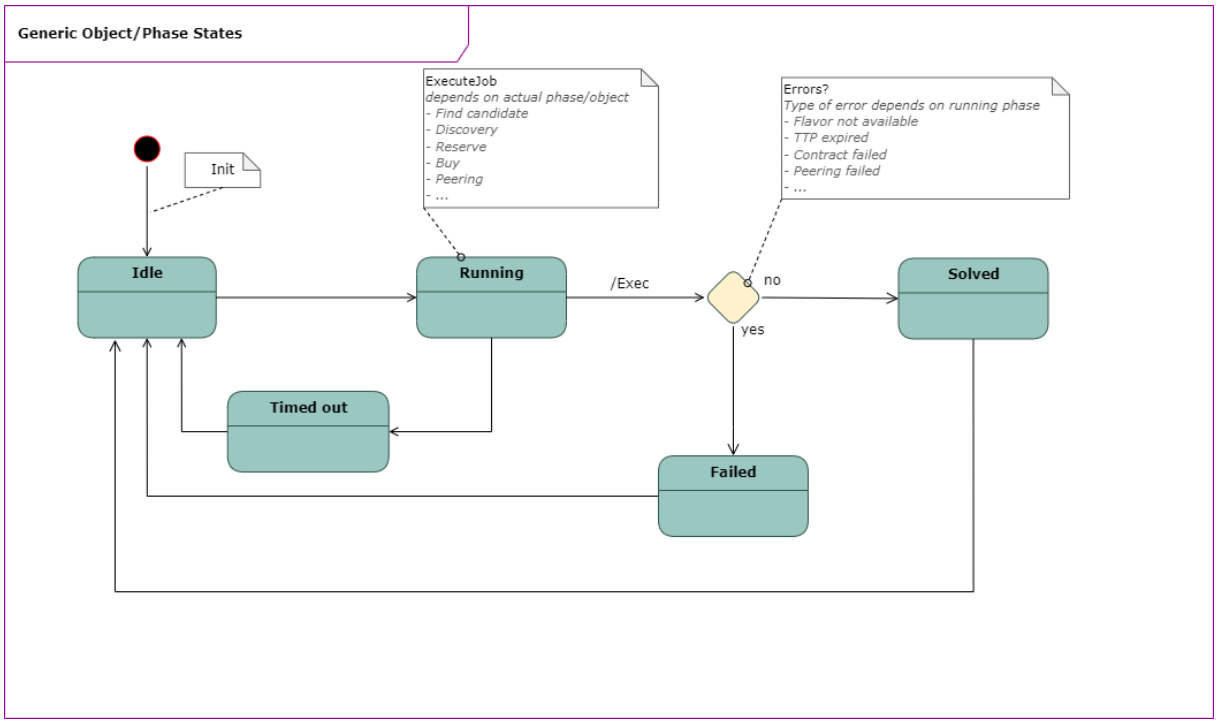

FIGURE 10: STATE DIAGRAM FOR SPECIFIC OBJECT/PHASE/SUBTASK

These state machines of course are related to the main state machine at discovery controller level, so for example:

- if the solver is in `Idle` state, all the sub-phases are also in `Idle` state;
- if the solver is in `Solved` state, all the sub-phases are also in `Solved` state;
- the situation is more complex when the discovery (solver) is in `Running` state, because, internally, the solver manages the specific subtasks/subphases that can assume different states, depending on the outcome of the phase itself or other correlated phases.

The following picture represents a simplified diagram of the relationship between the phases and their states.

Funded by Horizon Europe Framework Programme of the European Union

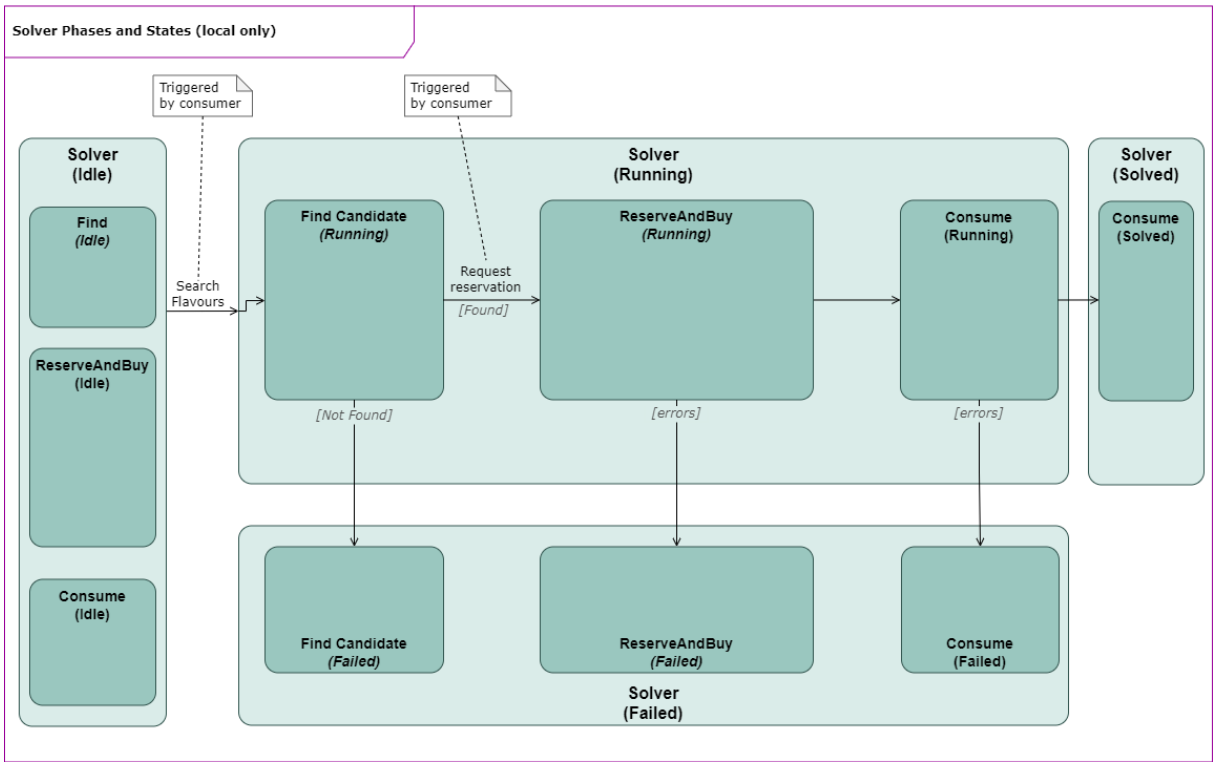

FIGURE 11: SOLVER PHASES AND STATES

We also have to consider that the three subphases `FindCandidate`, `ReserveAndBuy` and `Consume`, managed by the solver/discovery, may need to perform remote operations, by using REAR, Liqo, or other specific protocols or APIs.

For example:

- the `FindCandidate` phase as first operation, searches for a local node (flavour) that meets the consumer requirements but, if no suitable node is found locally, it uses the REAR controller to forward the same search on remote FLUIDOS Nodes;
- the `ReserveAndBuy` phase uses the REAR controller to reserve the node (making it unavailable for other consumers), and, if reservation is completed successfully, it continues to next subphase, to complete the purchase operation (creating the contract);
- the `Consume` phase calls the Peering functions (ForeignCluster::PeerWithCluster) to actually create the Liqo node.

Also, the remote phases (that uses REAR and Liqo controllers) implements state machines, so the whole discovery process creates a hierarchy of interdependent states that has the Solver/discovery at the top level, and goes down to a second level for specific tasks operating on local node, and a third level for remote operations (REAR and Liqo).

Funded by Horizon Europe Framework Programme of the European Union

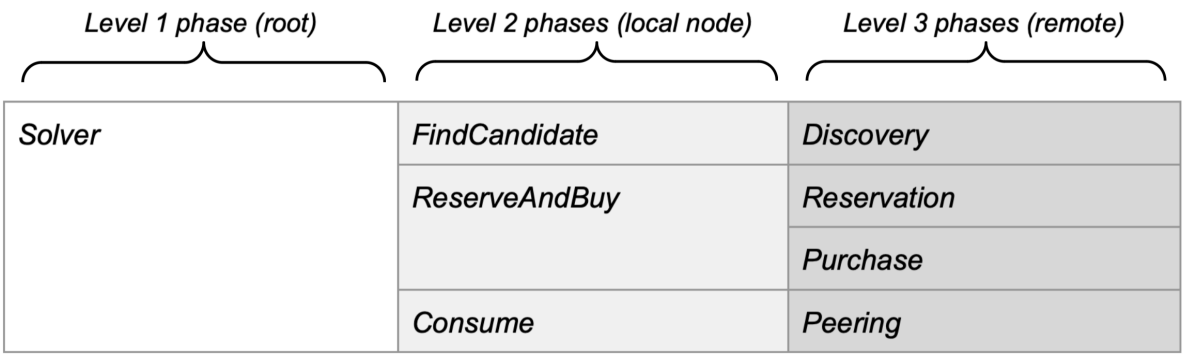

FIGURE 12: PHASES HIERARCHY

Each one of these phases in the hierarchy follows the same state diagram represented in Picture 3, but the change of its state can be triggered by either the phase itself or as result of an operation performed by one of its ancestor or child phases.

So, a parent phase can trigger the change of state of one of its child phases from idle to running, and the result of the children propagates up to its ancestors.

All the phases/subphases controllers maintain the right coherence between parent/children phases avoiding inconsistent or confusing states (i.e., a subphase cannot be in `running` state if parent phase is `not running`).

For example, when the remote reservation operation is in progress, the `Solver`, `ReserveAndBuy` and `Reservation` phases are all in a `Running` state, but if the `Reservation` (level 3) fails, it changes its state to `Failed` and also propagate the change of state to its ancestor phases (`ReserveAndBuy` and `Solver`) from `Running` to `Failed`.

The following picture shows a simplified diagram of the states of all the phases with the main actions that triggers a state change to a phase, to an ancestor phase, or to a child phase:

Funded by Horizon Europe Framework Programme of the European Union

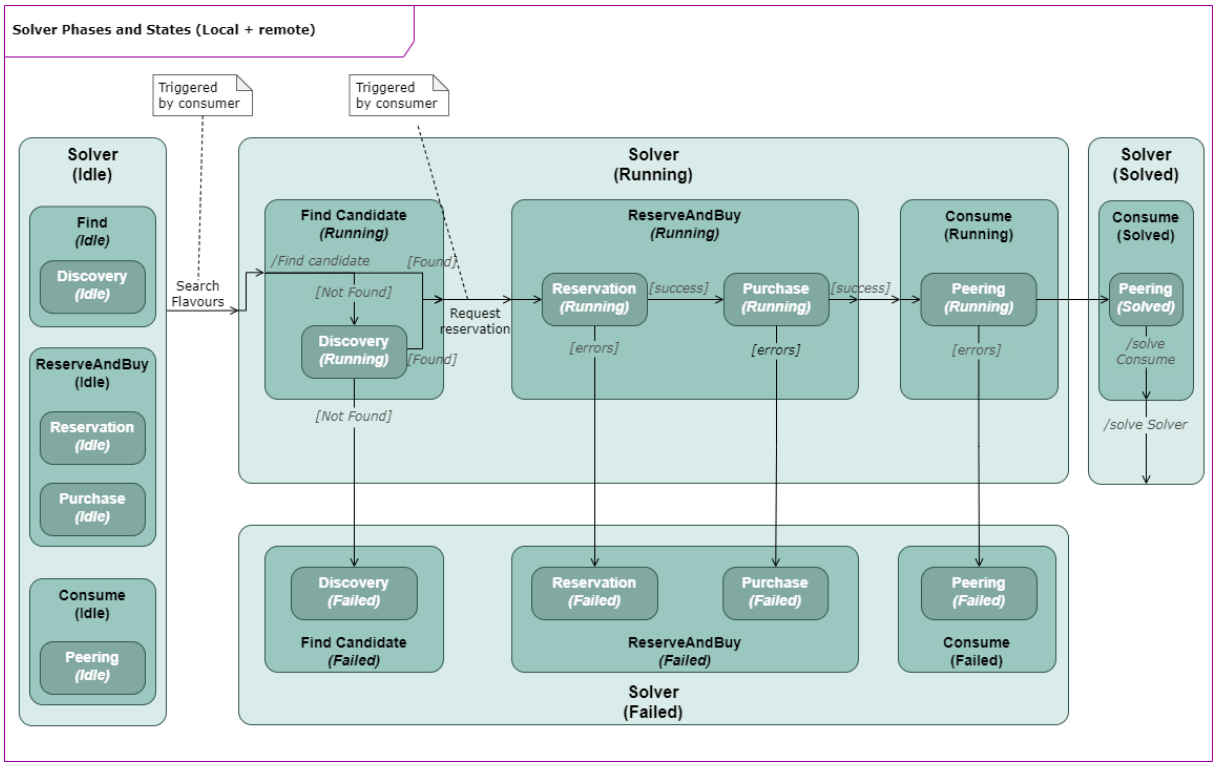

FIGURE 13: SOLVER PHASES AND STATES (LOCAL AND REMOTES)

Funded by Horizon Europe Framework Programme of the European Union

3 ABSTRACTIONS AND MODELS

3.1 INTRODUCTION

The activities in T3.2 focused on defining a common knowledge base for offering and acquiring resources. This is required because of the various actors operating within the space, in several occasions lacking a common vocabulary for the specification of requirements and resources. To do that, we relied on the standard practice of defining one or more ontologies for characterizing data and relationships within the FLUIDOS architecture.

Ontologies are a formal way to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains: the nouns representing classes of objects and the verbs representing relations between the objects.

The most well-known ontologies are the ones defined by the World Wide Web Consortium (W3C) to allow the creation of the so-called Semantic Web, also known as Linked Data. W3C published along the years several standards about formal languages for data and relationships description, and formal reasoning methods.

The main standards we want to reference here are:

- the Resource Description Framework (RDF)[3], a general framework for representing interconnected data on the web. RDF statements are used for describing and exchanging metadata, which enables standardized exchange of data based on relationships. RDF is used to integrate data from multiple sources.
- RDF Schema (Resource Description Framework Schema, variously abbreviated as RDFS, RDF(S), RDF-S, or RDF/S)[4], a set of classes with certain properties using the RDF extensible knowledge representation data model, providing basic elements for the description of ontologies.
- the Web Ontology Language (OWL)[5], a family of knowledge representation languages for authoring ontologies.
- Turtle, a textual syntax for RDF that allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes. Turtle provides levels of compatibility with the N-Triples format as well as the triple pattern syntax of the SPARQL W3C Recommendation.

3 https://www.w3.org/TR/rdf12-concepts/

4 https://www.w3.org/wiki/RDFS

5 https://www.w3.org/OWL/

Funded by Horizon Europe Framework Programme of the European Union

3.2 ONTOLOGIES

Namely, in T3.2 we leverage RDF, RDFS, and OWL for the definition of two main ontologies, which are publicly available in a repository called [fluidos-project/WP3_6-ontology].

3.2.1 Kubernetes

The first one describes Kubernetes data model, mainly the manifest language, in a formal manner. This ontology is machine generated, using a python script available in the utility directory of the repository.

This ontology is instrumental as a bridge between the resources offered by each FLUIDOS node, which we remind corresponds to a Kubernetes cluster consisting of one or more Kubernetes nodes, and the languages being developed to express user requirements.

This ontology contains the formal description of the existing relationships between (Kubernetes) nodes, pods, deployments, and the various components existing within a standard Kubernetes environment.

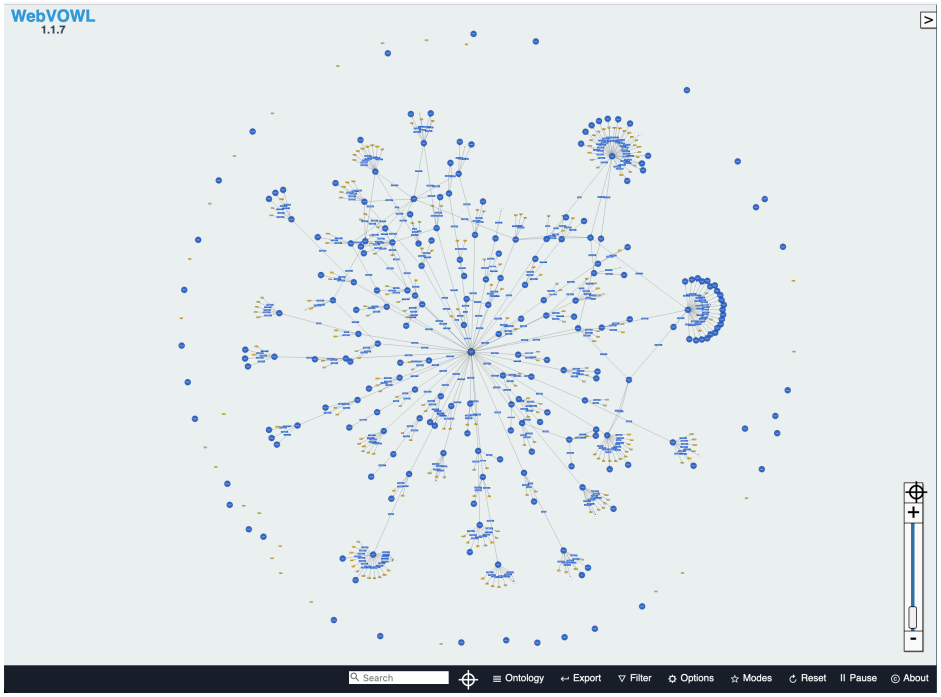

FIGURE 14: KUBERNETES ONTOLOGY VISUAL REPRESENTATION

3.2.2 FLUIDOS

The second ontology developed as part of this effort formally characterises FLUIDOS concepts. In its first version, the FLUIDOS ontology represents the architectural concepts developed in WP2 and implemented in WP3.

Please, refer to D2.1 and REAR protocol description for a more in-depth analysis of the mentioned architectural components.

Funded by Horizon Europe Framework Programme of the European Union

Moreover, the ontology is also an attempt to the formal characterisation of the concept of intent. Within FLUIDOS, an intent is a somehow defined user request. We here use the term "somehow" because according to interaction with use cases provided, both within FLUIDOS and prospective ones for the open calls, such a term has a very broad range of interpretation.

The definition spans from a simple set of requests in terms of resources (memory, CPU, etc.), to vague requests of high-level services (a SQL99 compatible database), passing through measurable request for application and infrastructure performances (latency, throughput, etc.) or to general availability, resilience, and compliance (.9 availability, to be HIPAA[6] compliant).

Current definition of intent attempts to capture such a requirement by leveraging the "flavour" concept for resource offering and demanding, alongside service requests, to express needs for databases or application servers for example, and ancillary information like compliance and geographical requirements.

Subsequent work will define mapping between established languages, like the Medium-level security policy language (MSPL) [15] used by partners in WP4 and WP5 or the intent definition from Intel, and FLUIDOS concepts.

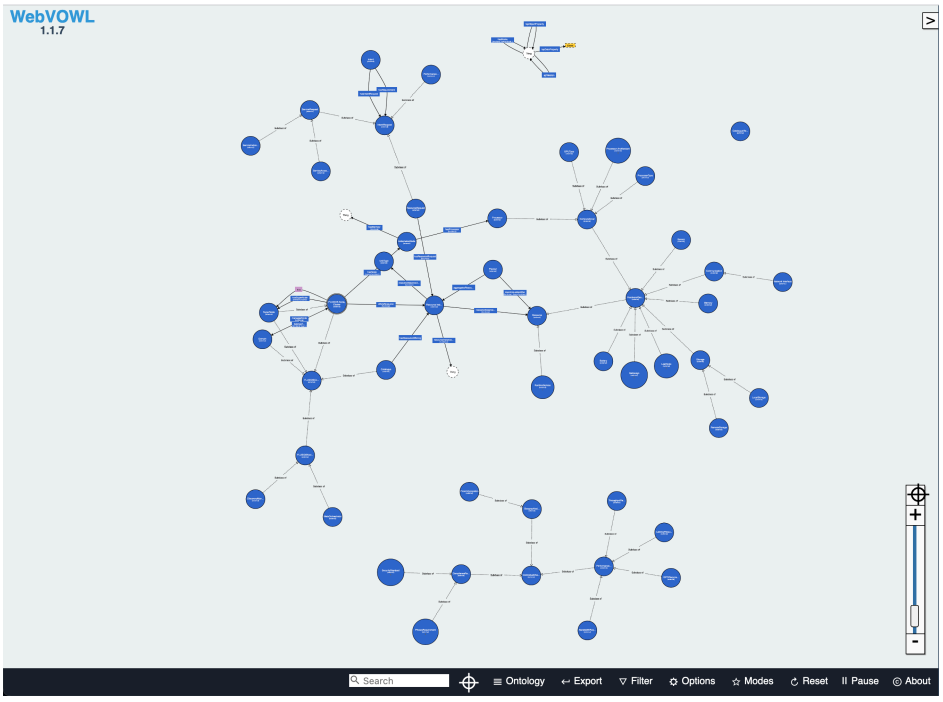

FIGURE 15: FLUIDOS ONTOLOGY VISUAL REPRESENTATION

6 https://www.cdc.gov/phlp/publications/topic/hipaa.html

Funded by Horizon Europe Framework Programme of the European Union

4 EDGE ARCHITECTURES

As presented in D2.1, through tethering in FLUIDOS edge architecture, data processing and analysis closer to the source reduces latency and enhances overall system efficiency in regard to management, data processing and communication with edge nodes (Meta-edge) and IoT devices (Deep-edge and Micro-edge). The following figure depicts the overall architecture that enables FLUIDOS nodes to leverage the power and capacity of edge devices.

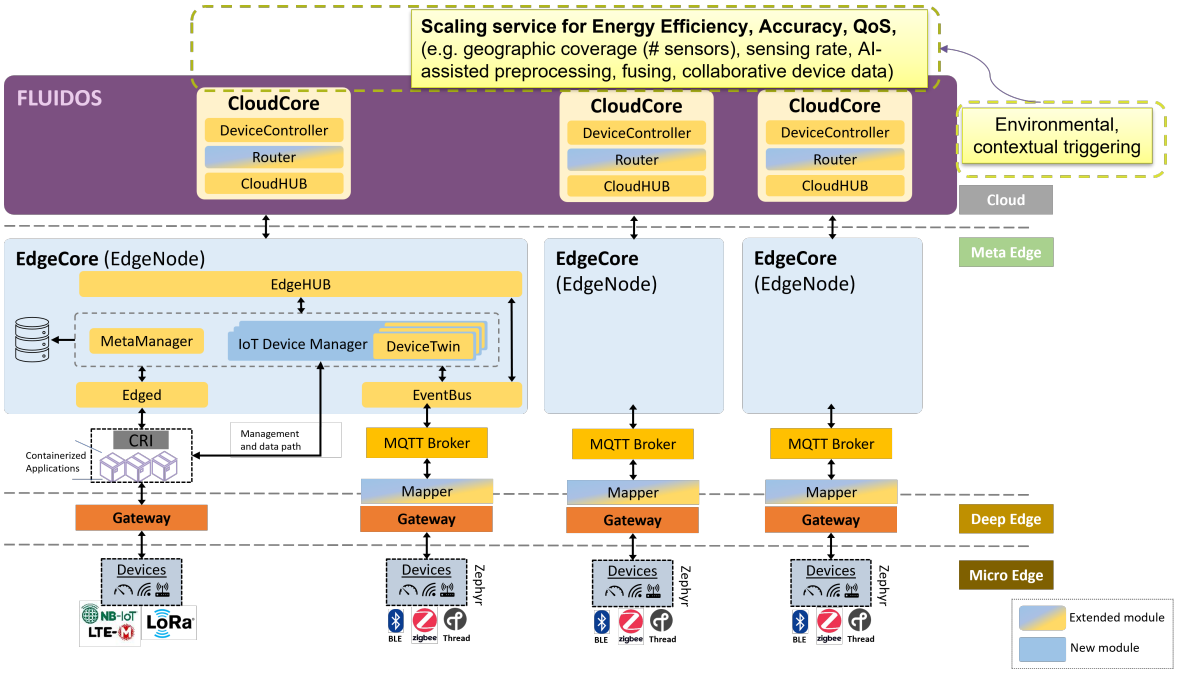

FIGURE 16: FLUIDOS EDGE ARCHITECTURE OVERVIEW

In FLUIDOS Edge, we leverage the Kubernetes custom resources to establish communication between the cloud and the Meta-edge, and to provide edge resource management, by leveraging KubeEdge (we use v1.14 as the baseline) functionality. KubeEdge provides the following functionality:

- Deep-edge and Micro-edge device management.
- Cloud to Edge and Edge to Cloud traffic routing.
- Kubernetes objects (e.g., devices, pods, etc.) synchronization between the Cloud and the Edge layers.

However, the vanilla version of KubeEdge has many limitations, including, the limited support of the well-known IoT protocols such as LoRA, sigfox, MATTER etc. For this reason, FLUIDOS provides uniformity to manage the heterogeneity of IoT Edge devices by improving and extending several components of the vanilla KubeEdge. Furthermore, FLUIDOS introduces novel features to enable the sharing of EdgeIoT resources and

Funded by Horizon Europe Framework Programme of the European Union

computations performed at the edge among different application executed at the cloud level. Namely, these improvements and extensions are the following:

- Cloud to Edge and Edge to Cloud traffic routing with software multicast support
- Uniformity to manage the heterogeneity of the leaf edge device (LED)

Related guides and source code is available at the FLUIDOS GitHub project: https://github.com/fluidos-project/fluidos-edge.

4.1 CLOUD LAYER

A thorough guide for properly installing and configuring the cloud layer can be found at the FLUIDOS GitHub project:

https://github.com/fluidos-project/fluidos-edge/tree/main/doc/installation-guide#cloud-layer-installation--configuration

4.1.1 Custom Resources

Device Management

Device models are used as templates to describe LEDs in a uniform and abstract way according to the openAPIv3Schema.

This CRD enables a device model, which is a schema for the device model API.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.6.2
  creationTimestamp: null
  name: devicemodels.devices.kubeedge.io
spec:
  group: devices.kubeedge.io
  names:
    kind: DeviceModel
    listKind: DeviceModelList
    plural: devicemodels
    singular: devicemodel
  scope: Namespaced
  versions:
  - name: v1alpha2
    schema:
      openAPIV3Schema:
        description: DeviceModel is the Schema for the device model API
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of this
representation of an object. Servers should convert recognized schemas to the
latest internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#resources'
```


Funded by Horizon Europe Framework Programme of the European Union

```
              type: string
            kind:
              description: 'Kind is a string value representing the REST resource
this object represents. Servers may infer this from the endpoint the client submits
requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#types-kinds'
              type: string
            metadata:
              type: object
            spec:
              description: DeviceModelSpec defines the model / template for a
device. It is a blueprint which describes the device capabilities and access
mechanism via property visitors.
              properties:
…
              type: object
          type: object
      served: true
      storage: true
  status:
    acceptedNames:
      kind: ""
      plural: ""
    conditions: []
    storedVersions: []
```

Device CRDs are used for applying a device instance, which a schema for the device API. This kind of CRD is used to describe in detail information related to specific devices, such as brand, model, capabilities, etc., within the limits and possibilities described by the model. At Deep/Micro Edge Device Commands subsection we provide an example where we create two devices based on different type of flavours.

```
  apiVersion: apiextensions.k8s.io/v1
  kind: CustomResourceDefinition
  metadata:
    annotations:
      controller-gen.kubebuilder.io/version: v0.6.2
    creationTimestamp: null
    name: devices.devices.kubeedge.io
  spec:
    group: devices.kubeedge.io
    names:
      kind: Device
      listKind: DeviceList
      plural: devices
      singular: device
    scope: Namespaced
    versions:
    - name: v1alpha2
      schema:
        openAPIV3Schema:
          description: Device is the Schema for the devices API
          properties:
            apiVersion:
              description: 'APIVersion defines the versioned schema of this
representation of an object. Servers should convert recognized schemas to the
latest internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#resources'
```


Funded by Horizon Europe Framework Programme of the European Union

```
              type: string
            kind:
              description: 'Kind is a string value representing the REST resource
this object represents. Servers may infer this from the endpoint the client submits
requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#types-kinds'
              type: string
            metadata:
              type: object
            spec:
              description: DeviceSpec represents a single device instance. It is an
instantation of a device model.
              properties:
                data:
                  description: Data section describe a list of time-series
properties which should be processed on edge node.
                  properties:
…
      served: true
      storage: true
  status:
    acceptedNames:
      kind: ""
      plural: ""
    conditions: []
    storedVersions: []
```

Traffic Routing

Traffic routing CRDs provide all the required information (rules) for routing traffic from the edge to the cloud and vice versa.

Here is the CRD for applying a router endpoint, which a description of the router endpoint type.

```
  apiVersion: apiextensions.k8s.io/v1
  kind: CustomResourceDefinition
  metadata:
    name: ruleendpoints.rules.kubeedge.io
  spec:
    group: rules.kubeedge.io
    versions:
      - name: v1
        served: true
        storage: true
        schema:
          openAPIV3Schema:
            type: object
            properties:
              spec:
                type: object
                properties:
                  ruleEndpointType:
                    description: ruleEndpointType is a string value representing
rule-endpoint type. its value is one of rest/eventbus/servicebus.
                    type: string
                    enum:
                      - rest
                      - eventbus
```


Funded by Horizon Europe Framework Programme of the European Union

```
                    - servicebus
                  properties:
                    description: properties is not required except for servicebus
rule-endpoint type. It is a map representing rule-endpoint properties. When
ruleEndpointType is servicebus, its value is {"service_port":"8080"}.
                    type: object
                    additionalProperties:
                      type: string
                required:
                  - ruleEndpointType
    scope: Namespaced
    names:
      plural: ruleendpoints
      singular: ruleendpoint
      kind: RuleEndpoint
      shortNames:
        - re
```

Here is the CRD for applying a router rule, which is a rule for routing the traffic coming from the edge endpoint to cloud endpoints.

```
  apiVersion: apiextensions.k8s.io/v1
  kind: CustomResourceDefinition
  metadata:
    name: rules.rules.kubeedge.io
  spec:
    group: rules.kubeedge.io
    versions:
      - name: v1
        served: true
        storage: true
        schema:
          openAPIV3Schema:
            type: object
            properties:
              spec:
                type: object
                properties:
                  source:
                    description: source is a string value representing where the
messages come from. Its value is the same with ruleendpoint name. For example, my-
rest or my-eventbus.
                    type: string
                  sourceResource:
                    description: is a map representing the resource info of source.
For rest rule-endpoint type its value is {"path":"/test"}. For eventbus
ruleendpoint type its value is {"topic":"<user define string>","node_name":"edge-
node"}
                    type: object
                    additionalProperties:
                      type: string
                  target:
                    description: target is a string value representing where the
messages go to. its value is same with ruleendpoint name. For example, my-eventbus
or my-rest or my-servicebus.
                    type: string
                  targetResource:
                    description: targetResource is a map representing the resource
info of target. For rest rule-endpoint type its value is
{"resource":"http://a.com"}. For eventbus ruleendpoint its value is
{"topic":"/test"}. For servicebus rule-endpoint type its value is
{"path":"/request_path"}.
```


Funded by Horizon Europe Framework Programme of the European Union

```
                  type: object
                  additionalProperties:
                    type: string
              required:
                - source
                - sourceResource
                - target
                - targetResource
            status:
              type: object
              properties:
                successMessages:
                  type: integer
                failMessages:
                  type: integer
                errors:
                  items:
                    type: string
                  type: array
  scope: Namespaced
  names:
    plural: rules
    singular: rule
    kind: Rule
```

Object Synchronization

It is essential to keep the state and status of the various instantiated objects in synchronization between the cloud and the edge layer, e.g., device status.

Here is the CRD for applying name-spaced objects synchronization.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.6.2
  creationTimestamp: null
  name: objectsyncs.reliablesyncs.kubeedge.io
spec:
  group: reliablesyncs.kubeedge.io
  names:
    kind: ObjectSync
    listKind: ObjectSyncList
    plural: objectsyncs
    singular: objectsync
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        description: ObjectSync stores the state of the namespaced object that
was successfully persisted to the edge node. ObjectSync name is a concatenation the
node name which receiving the object and the object UUID.
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of this
representation of an object. Servers should convert recognized schemas to the
latest internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#resources'
```


Funded by Horizon Europe Framework Programme of the European Union

```
              type: string
            kind:
            description: 'Kind is a string value representing the REST resource
this object represents. Servers may infer this from the endpoint the client submits
requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#types-kinds'
              type: string
            metadata:
              type: object
            spec:
              description: ObjectSyncSpec stores the details of objects that
persist to the edge.
              properties:
                objectAPIVersion:
                  description: ObjectAPIVersion is the APIVersion of the object
that was successfully persist to the edge node.
                  type: string
                objectKind:
                  description: ObjectType is the kind of the object that was
successfully persist to the edge node.
                  type: string
                objectName:
                  description: ObjectName is the name of the object that was
successfully persist to the edge node.
                  type: string
              type: object
            status:
              description: ObjectSyncSpec stores the resourceversion of objects
that persist to the edge.
              properties:
                objectResourceVersion:
                  description: ObjectResourceVersion is the resourceversion of the
object that was successfully persist to the edge node.
                  type: string
              type: object
          type: object
      served: true
      storage: true
      subresources:
        status: {}
  status:
    acceptedNames:
      kind: ""
      plural: ""
    conditions: []
    storedVersions: []
```

Here is the CRD for applying non-name-spaced objects synchronization.

```
  apiVersion: apiextensions.k8s.io/v1
  kind: CustomResourceDefinition
  metadata:
    annotations:
      controller-gen.kubebuilder.io/version: v0.6.2
    creationTimestamp: null
    name: clusterobjectsyncs.reliablesyncs.kubeedge.io
  spec:
    group: reliablesyncs.kubeedge.io
    names:
      kind: ClusterObjectSync
      listKind: ClusterObjectSyncList
```


Funded by Horizon Europe Framework Programme of the European Union

```
      plural: clusterobjectsyncs
      singular: clusterobjectsync
    scope: Cluster
    versions:
    - name: v1alpha1
      schema:
        openAPIV3Schema:
          description: ClusterObjectSync stores the state of the cluster level,
nonNamespaced object that was successfully persisted to the edge node.
ClusterObjectSync name is a concatenation of the node name which receiving the
object and the object UUID.
          properties:
            apiVersion:
              description: 'APIVersion defines the versioned schema of this
representation of an object. Servers should convert recognized schemas to the
latest internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#resources'
              type: string
            kind:
              description: 'Kind is a string value representing the REST resource
this object represents. Servers may infer this from the endpoint the client submits
requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#types-kinds'
              type: string
            metadata:
              type: object
            spec:
              description: ObjectSyncSpec stores the details of objects that
persist to the edge.
              properties:
                objectAPIVersion:
                  description: ObjectAPIVersion is the APIVersion of the object
that was successfully persist to the edge node.
                  type: string
                objectKind:
                  description: ObjectType is the kind of the object that was
successfully persist to the edge node.
                  type: string
                objectName:
                  description: ObjectName is the name of the object that was
successfully persist to the edge node.
                  type: string
              type: object
            status:
              description: ObjectSyncSpec stores the resource version of objects
that persist to the edge.
              properties:
                objectResourceVersion:
                  description: ObjectResourceVersion is the resource version of the
object that was successfully persist to the edge node.
                  type: string
              type: object
          type: object
      served: true
      storage: true
      subresources:
        status: {}
  status:
    acceptedNames:
      kind: ""
      plural: ""
    conditions: []
```


Funded by Horizon Europe Framework Programme of the European Union

```
  storedVersions: []
```

4.1.2 Custom Controllers

CloudHub

CloudHub module connects the controllers and the Meta-edge over web-socket and QUIC protocol. The protocol is selected by EdgeHub. CloudHub's function is to enable the communication between edge and the Controllers.

CloudHub The main functions performed by are:

- Serve messages coming from the Meta-edge.
- Forward messages to the Meta-edge.

EdgeController

EdgeController bridges the Kubernetes API server and EdgeCore. Its main functionality is to synchronise objects' state and status, serving updates coming from both the cloud and the Meta-edge layer.

DeviceController

The DeviceController provides device management. It synchronizes devices' status and metadata updates between the cloud and the Deep/Micro-edge, by leveraging Kubernetes CRDs.

Router

This controller can be used to route traffic from the Edge layers to other Cloud applications or vice versa, based on rules. We have extended its functionality to support data multicast in the Edge to Cloud direction and also to process filtering rules, i.e., which data to forward to which endpoint.

4.2 EDGE LAYER

A thorough guide for properly installing and configuring the edge layer can be found at the FLUIDOS GitHub project:

https://github.com/fluidos-project/fluidos-edge/tree/main/doc/installation-guide#meta-edge-layer-installation--configuration

Funded by Horizon Europe Framework Programme of the European Union

4.2.1 Modules

At the Meta-edge layer, the EdgeCore enables proper integration with the cloud layer and it consists of the following entities.

EdgeD

EdgeD enables deploying containerized workloads (pods) at the Meta-edge, serving cloud layer commands.

EventBus

The EventBus is an interface that provides proper communication with an MQTT broker.

DeviceTwin

The DeviceTwin maintains information related to Deep-edge and Micro-edge devices and is responsible for synchronize this information between the cloud and the Meta-Edge.

EdgeHub

The EdgeHub links the cloud and the Meta-edge layers, after establishing a connection with the CloudHub.

MetaManager

The MetaManager acts as a glue layer between EdgeD and EdgeHub. Also, it maintains metadata using a lightweight database.

IoT Device Manager

This newly introduced module can connect to and manage more complex types of IoT networks, where their higher-level entity is running at the Meta-edge in a containerized fashion. One such example is a LoRaWAN network, where the LoRaWAN servers or at least the application server are/is running inside a pod at the Meta-edge layer.

Mapper

The Mapper is an application that connects to and controls devices at the Deep-edge and the Micro-edge. We have enhanced the Mapper's functionality to be able to request from the LEDs to serve the system based on the flavour described during device creation. Moreover, we have implemented the support to associate data with multiple recipients, i.e., publish (at higher layers) data at multiple topics.

Funded by Horizon Europe Framework Programme of the European Union

4.3 APIS

A set of API endpoints enables accessing the custom resources, i.e., devices and router rules. These are the available endpoints:

- /apis/devices.kubeedge.io/v1alpha2
- /apis/rules.kubeedge.io/v1/

4.3.1 Deep/Micro Edge Device Commands

By accessing /apis/devices.kubeedge.io/v1alpha2 API endpoint, several commands (over HTTP) can be invoked to manage the Deep-edge and Micro-edge devices. These commands target device model and device objects. Device model creation should always precede device creation, and every device should be deleted before deleting a device model. The available commands are:

- Get
- List
- Create
- Patch
- Delete

List Device & Device Model

List device models example:

```
  GET https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/namespaces/
{namespace}/devicemodels
```

Response body:


```
  {
    "apiVersion": "v1",
    "items": [
      {
        "apiVersion": "devices.kubeedge.io/v1alpha2",
        "kind": "DeviceModel",
        "metadata": {
          "annotations": {
            "kubectl.kubernetes.io/last-applied-configuration":
"{\"apiVersion\":\"devices.kubeedge.io/v1alpha2\",\"kind\":\"DeviceModel\",\"metada
ta\":{\"annotations\":{},\"name\":\"bluenrg\",\"namespace\":\"default\"},\"spec\":{
\"properties\":[{\"description\":\"temperature and atmospheric
pressure\",\"name\":\"environmental\",\"type\":{\"int\":{\"accessMode\":\"ReadOnly\
"}}}]}}\n"
          },
          "creationTimestamp": "2023-10-30T12:44:12Z",
          "generation": 1,
          "name": "bluenrg",
          "namespace": "default",
```


Funded by Horizon Europe Framework Programme of the European Union


```
          "resourceVersion": "836065",
          "uid": "b4b72156-ae58-4006-93ef-29ad5b5aae82"
        },
        "spec": {
          "properties": [
            {
              "description": "temperature and atmospheric pressure",
              "name": "environmental",
              "type": {
                "int": {
                  "accessMode": "ReadOnly"
                }
              }
            }
          ]
        }
      }
    ],
    "kind": "List",
    "metadata": {
      "resourceVersion": ""
    }
  }
```


List devices example:

```
  GET https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/namespaces/
{namespace}/devices
```

Response body:


```
  {
    "apiVersion": "v1",
    "items": [
      {
        "apiVersion": "devices.kubeedge.io/v1alpha2",
        "kind": "Device",
        "metadata": {
          "annotations": {
            "kubectl.kubernetes.io/last-applied-configuration":
"{\"apiVersion\":\"devices.kubeedge.io/v1alpha2\",\"kind\":\"Device\",\"metadata\":
{\"annotations\":{},\"labels\":{\"description\":\"Sensor-Tile-
Board\",\"manufacturer\":\"STMicroelectronics\",\"model\":\"STWINKT1B\"},\"name\":\
"bluenrg-instance-
01\",\"namespace\":\"default\"},\"spec\":{\"data\":{\"dataProperties\":[{\"metadata
\":{\"type\":\"integer\"},\"propertyName\":\"environmental\"}],\"dataTopic\":\"$ke/
events/device/+/data/update\"},\"deviceModelRef\":{\"name\":\"bluenrg\"},\"nodeSele
ctor\":{\"nodeSelectorTerms\":[{\"matchExpressions\":[{\"key\":\"\",\"operator\":\"
In\",\"values\":[\"edgian\"]}]}]},\"propertyVisitors\":[{\"bluetooth\":{\"character
isticUUID\":\"00140000000111e1ac360002a5d5c51b\",\"dataConverter\":{\"endIndex\":7,
\"orderOfOperations\":null,\"startIndex\":2}},\"collectCycle\":500000000,\"property
Name\":\"environmental\",\"reportCycle\":1000000000}],\"protocol\":{\"bluetooth\":{
\"macAddress\":\"CB:B8:C2:15:19:EF\"}}},\"status\":{\"twins\":[{\"propertyName\":\"
environmental\"}]}}\n"
          },
          "creationTimestamp": "2023-10-30T12:44:16Z",
          "generation": 1,
          "labels": {
            "description": "Sensor-Tile-Board",
            "manufacturer": "STMicroelectronics",
```


Funded by Horizon Europe Framework Programme of the European Union

```
            "model": "STWINKT1B"
          },
          "name": "bluenrg-instance-01",
          "namespace": "default",
          "resourceVersion": "836071",
          "uid": "9928ce92-686d-45ed-9290-bf72897e1bfe"
        },
        "spec": {
          "data": {
            "dataProperties": [
              {
                "metadata": {
                  "type": "integer"
                },
                "propertyName": "environmental"
              }
            ],
            "dataTopic": "$ke/events/device/+/data/update"
          },
          "deviceModelRef": {
            "name": "bluenrg"
          },
          "nodeSelector": {
            "nodeSelectorTerms": [
              {
                "matchExpressions": [
                  {
                    "key": "",
                    "operator": "In",
                    "values": [
                      "edgian"
                    ]
                  }
                ]
              }
            ]
          },
          "propertyVisitors": [
            {
              "bluetooth": {
                "characteristicUUID": "00140000000111e1ac360002a5d5c51b",
                "dataConverter": {
                  "endIndex": 7,
                  "startIndex": 2
                }
              },
              "collectCycle": 500000000,
              "propertyName": "environmental",
              "reportCycle": 1000000000
            }
          ],
          "protocol": {
            "bluetooth": {
              "macAddress": "CB:B8:C2:15:19:EF"
            }
          }
        },
        "status": {
          "twins": [
            {
              "propertyName": "environmental"
            }
          ]
        }
```


Funded by Horizon Europe Framework Programme of the European Union

```
      }
    ],
    "kind": "List",
    "metadata": {
      "resourceVersion": ""
    }
  }
```

Get Device & Device Model

Get a specific device model example:

```
  GET https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/namespaces/
{namespace}/devicemodels/{device model name}
```

Get a specific device example:

```
  GET https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/namespaces/
{namespace}/devices/{device name}
```

Create Device & Device Model

Create a device model example:

```
  POST https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/
namespaces/{namespace}/devicemodels
```

Request header:

```
  'Content-Type: application/yaml'
```

Request body sample:

```
  apiVersion: devices.kubeedge.io/v1alpha2
  kind: DeviceModel
  metadata:
    name: bluenrg
    namespace: default
  spec:
    properties:
    - name: temperature
      description: environmental temperature
      type:
      - int:
          accessMode: ReadOnly
    - name: pressure
      description: atmospheric pressure
      type:
      - int:
          accessMode: ReadOnly
    - name: accelerometer
      description: acceleration X Y Z values
      type:
```


Funded by Horizon Europe Framework Programme of the European Union

```
    - float:
        accessMode: ReadOnly
    - float:
        accessMode: ReadOnly
    - float:
        accessMode: ReadOnly
```

Create a device example:

```
  POST https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/
namespaces/{namespace}/devices
```

Request header:

```
  'Content-Type: application/yaml'
```

Request body

```
  As described at the following two examples device object samples
```

Here are two examples where we instantiate device resources based on different flavour resource parameters. Based on the flavour parameters, the first one is instantiated to periodically send data collected from environmental sensors (e.g., temperature, pressure), while the second one is instantiated to periodically send data collected from motion sensors (e.g., accelerometer).

Example 1

This the first flavour resource sample that describes the interest for a device that can provide readings from temperature and pressure sensors.

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Flavour
metadata:
  creationTimestamp: "2023-11-16T16:13:52Z"
  generation: 2
  name: fluidos.eu-k8s-fluidos-bba29928
  namespace: fluidos
  resourceVersion: "1534"
  uid: 5ce9f378-014e-4cb4-b173-5a3530d8f78d
spec:
  characteristics:
    architecture: armv7
    cpu: M0+
    ephemeral-storage: "0"
    gpu: "0"
    mpu: "0"
    memory: “6144”
    persistent-storage: "0"
    crypto-accelerator:         # crypto-accelerator list
    secure-element:             # secure element list
    sensor:                           # sensors list
      - type: temperature
      - type: pressure
```


Funded by Horizon Europe Framework Programme of the European Union

```
    mems:                               # mems list
    actuator:                            # actuator list
  optionalFields:
    workerID: fluidos-provider-worker
  owner:
    domain: fluidos.eu
    ip: 172.18.0.7:30001
    nodeID: 46ltws9per
  policy:
    aggregatable:
      maxCount: 0
      minCount: 0
    partitionable:
      cpuMin: "0"
      cpuStep: "1"
      memoryMin: "0"
      memoryStep: 100Mi
  price:
    amount: ""
    currency: ""
    period: ""
  providerID: 46ltws9per
  type: k8s-fluidos
```

This is a device resource sample aligned with the flavour described above.

```
apiVersion: devices.kubeedge.io/v1alpha2
kind: Device
metadata:
  name: bluenrg-instance-01
  labels:
    description: Sensor-Tile-Board
    manufacturer: STMicroelectronics
    model: STWINBX1
spec:
  deviceModelRef:
    name: bluenrg
  protocol:
    bluetooth:
      macAddress: "CB:B8:C2:15:19:EF" #MAC address of the IoT device to pair with
  nodeSelector:
    nodeSelectorTerms:
    - matchExpressions:
      - key: ''
        operator: In
        values:
        - edgian          #pls give your edge node name
  propertyVisitors:
    - propertyName: temperature
      collectCycle: 500000000
      reportCycle: 1000000000
      bluetooth:
        characteristicUUID: 00140000000111e1ac360002a5d5c51b
        dataConverter:
         startIndex: 2
         endIndex: 4
         orderOfOperations:
    - propertyName: pressure
      collectCycle: 500000000
      reportCycle: 1000000000
      bluetooth:
        characteristicUUID: 00140000000111e1ac360002a5d5c51c
```


Funded by Horizon Europe Framework Programme of the European Union

```
        dataConverter:
         startIndex: 5
         endIndex: 7
         orderOfOperations:
  data:
    dataTopic: "$ke/events/device/+/data/update"
    dataProperties:
      - propertyName: temperature
        metadata:
        - type: integer
      - propertyName: pressure
        metadata:
        - type: integer
status:
  twins:
    - propertyName: temperature
    - propertyName: pressure
```

Example 2

This the second flavour resource sample that describes the interest for a device that can provide readings from an accelerometer.

```
apiVersion: nodecore.fluidos.eu/v1alpha1
kind: Flavour
metadata:
  creationTimestamp: "2023-11-16T16:13:52Z"
  generation: 2
  name: fluidos.eu-k8s-fluidos-bba29928
  namespace: fluidos
  resourceVersion: "1534"
  uid: 5ce9f378-014e-4cb4-b173-5a3530d8f78d
spec:
  characteristics:
    architecture: armv7
    cpu: M0+
    ephemeral-storage: "0"
    gpu: "0"
    mpu: "0"
    memory: “6144”
    persistent-storage: "0"
    crypto-accelerator:         # crypto-accelerator list
    secure-element:             # secure element list
    sensor:                           # sensors list
    mems:                           # mems list
      - type: accelerometer
    actuator:                          # actuator list
  optionalFields:
    workerID: fluidos-provider-worker
  owner:
    domain: fluidos.eu
    ip: 172.18.0.7:30001
    nodeID: 46ltws9per
  policy:
    aggregatable:
      maxCount: 0
      minCount: 0
    partitionable:
      cpuMin: "0"
      cpuStep: "1"
      memoryMin: "0"
```


Funded by Horizon Europe Framework Programme of the European Union

```
        memoryStep: 100Mi
    price:
      amount: ""
      currency: ""
      period: ""
    providerID: 46ltws9per
    type: k8s-fluidos
```

This is a device resource sample aligned with the flavour described above.

```
apiVersion: devices.kubeedge.io/v1alpha2
kind: Device
metadata:
  name: bluenrg-instance-02
  labels:
    description: Sensor-Tile-Board
    manufacturer: STMicroelectronics
    model: STWINBX1
spec:
  deviceModelRef:
    name: bluenrg
  protocol:
    bluetooth:
      macAddress: "CB:B8:C2:3F:19:EF" #MAC address of the IoT device to pair with
  nodeSelector:
    nodeSelectorTerms:
    - matchExpressions:
      - key: ''
        operator: In
        values:
        - edgian          #pls give your edge node name
  propertyVisitors:
    - propertyName: accelerometer
      collectCycle: 500000000
      reportCycle: 1000000000
      bluetooth:
        characteristicUUID: 00140000000111e1ac240002a5d5c51d
  data:
    dataTopic: "$ke/events/device/+/data/update"
    dataProperties:
      - propertyName: accelerometer
        metadata:
        - type: float
        - type: float
        - type: float
status:
  twins:
    - propertyName: accelerometer
```

Patch Device & Device Model

Patch a device model example:

```
  PATCH https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/
namespaces/{namespace}/devicemodels/{model name}
```


Funded by Horizon Europe Framework Programme of the European Union

Request header:

```
'Content-Type: application/merge-patch+json'
```

Request body sample:

```
{
  "spec": {
    "properties": [
      {
        "description": "temperature, humidity and atmospheric pressure"
      }
    ]
  }
}
```

Patch a device example:

```
PATCH https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/
namespaces/{namespace}/devices/{device name}
```

Request header:

```
'Content-Type: application/merge-patch+json'
```

Request body sample:

```
{
  "spec": {
    "deviceModelRef": {
      "name": "bluenrg"
    }
  }
}
```

Delete Device & Device Model

Delete a device model example:

```
DELETE https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/
namespaces/{namespace}/devicemodels/{model name}
```

Delete a device example:

```
DELETE https://{API Server URL}:{port}/apis/devices.kubeedge.io/v1alpha2/
namespaces/{namespace}/devices/{device name}
```


Funded by Horizon Europe Framework Programme of the European Union

4.3.2 Router Commands

By accessing /apis/rules.kubeedge.io/v1/ API endpoint, several commands (over HTTP) can be invoked to manage the Deep-edge and Micro-edge devices. These commands target device model and device objects. Device model creation should always precede device creation, and every device should be deleted before deleting a device model. The available commands are:

- Get
- List
- Create
- Patch
- Delete

List Rules & Rule Endpoints

List rule endpoints example:

```
  GET https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/ruleendpoints
```

Response body:


```
  {
    "apiVersion": "v1",
    "items": [
      {
        "apiVersion": "rules.kubeedge.io/v1",
        "kind": "RuleEndpoint",
        "metadata": {
          "annotations": {
            "kubectl.kubernetes.io/last-applied-configuration":
"{\"apiVersion\":\"rules.kubeedge.io/v1\",\"kind\":\"RuleEndpoint\",\"metadata\":{\
"annotations\":{},\"labels\":{\"description\":\"test\"},\"name\":\"fluidos-
rest\",\"namespace\":\"default\"},\"spec\":{\"properties\":{},\"ruleEndpointType\":
\"rest\"}}\n"
          },
          "creationTimestamp": "2023-11-21T22:23:23Z",
          "generation": 1,
          "labels": {
            "description": "test"
          },
          "name": "fluidos-rest",
          "namespace": "default",
          "resourceVersion": "3318606",
          "uid": "7ac3f2d6-4c69-475f-aaec-dcaa165f3fac"
        },
        "spec": {
          "properties": {},
          "ruleEndpointType": "rest"
        }
      }
    ],
    "kind": "List",
```


Funded by Horizon Europe Framework Programme of the European Union

```
    "metadata": {
      "resourceVersion": ""
    }
  }
```

List rules example:

```
  GET https://{API Server URL}:{port}/ apis/rules.kubeedge.io/v1/namespaces/
{namespace}/rules
```

Response body:

```
  {
    "apiVersion": "v1",
    "items": [
      {
        "apiVersion": "rules.kubeedge.io/v1",
        "kind": "Rule",
        "metadata": {
          "annotations": {
            "kubectl.kubernetes.io/last-applied-configuration":
"{\"apiVersion\":\"rules.kubeedge.io/v1\",\"kind\":\"Rule\",\"metadata\":{\"annotat
ions\":{},\"labels\":{\"description\":\"Receive_topic_btled1_tp_forward_to_targets\
"},\"name\":\"rule-eventbus-rest-bt-led-1-
tp\",\"namespace\":\"default\"},\"spec\":{\"source\":\"fluidos-
eventbus\",\"sourceResource\":{\"node_name\":\"edgian3\",\"topic\":\"cloudapp-
tp/bt-led-1\"},\"target\":\"fluidos-
rest\",\"targetResource\":{\"resource\":\"http://10.0.2.68:4487/telegraf,http://10.
0.2.69:4487/telegraf\"}}}\n"
          },
          "creationTimestamp": "2023-11-21T22:24:36Z",
          "generation": 1,
          "labels": {
            "description": "Receive_topic_btled1_tp_forward_to_targets"
          },
          "name": "rule-eventbus-rest-bt-led-1-tp",
          "namespace": "default",
          "resourceVersion": "3318701",
          "uid": "21b16d0c-0035-4260-b78a-ee32af21335a"
        },
        "spec": {
          "source": "fluidos-eventbus",
          "sourceResource": {
            "node_name": "edgian3",
            "topic": "cloudapp-tp/bt-led-1"
          },
          "target": "fluidos-rest",
          "targetResource": {
            "resource":
"http://10.0.2.68:4487/telegraf,http://10.0.2.69:4487/telegraf"
          }
        }
      }
    ],
    "kind": "List",
    "metadata": {
      "resourceVersion": ""
    }
  }
```


Funded by Horizon Europe Framework Programme of the European Union

Get Rules & Rule Endpoints

Get a specific rule endpoint example:

```
  GET https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/ruleendpoints/{rule endpoint name}
```

Get a specific rule endpoint example:

```
  GET https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/rules/{rule name}
```

Create Rules & Rule Endpoints

Create a rule endpoint example:

```
  POST https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/ruleendpoints
```

Request header:

```
  'Content-Type: application/yaml'
```

Request body sample:

```
  apiVersion: rules.kubeedge.io/v1
  kind: RuleEndpoint
  metadata:
    name: fluidos-rest
    labels:
      description: test
  spec:
    ruleEndpointType: "rest"
    properties: {}
```

Create a rule example:

```
  POST https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/rules
```

Request header:

```
  'Content-Type: application/yaml'
```

Request body sample:

```
  apiVersion: rules.kubeedge.io/v1
  kind: Rule
  metadata:
```


Funded by Horizon Europe Framework Programme of the European Union

```
    name: rule-eventbus-rest-bt-led-1-tp
    labels:
      description: Receive_topic_btled1_tp_forward_to_targets
  spec:
    source: "fluidos-eventbus"
    sourceResource: {"topic": "cloudapp-tp/bt-led-1","node_name": "edgian3"}
    target: "fluidos-rest"
    targetResource:
{"resource":"http://10.0.2.68:4487/telegraf,http://10.0.2.69:4487/telegraf"}
```

Patch Rules & Rule Endpoints

Patch a rule endpoint example:

```
  PATCH https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/ruleendpoints/{rule endpoint name}
```

Request header:

```
  'Content-Type: application/merge-patch+json'
```

Request body sample:

```
  {
    "spec": {
      "ruleEndpointType": "eventbus"
    }
  }
```

Patch a device example:

```
  PATCH https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/rules/{rule name}
```

Request header:

```
  'Content-Type: application/merge-patch+json'
```

Request body sample:

```
  {
    "spec": {
      "target": "fluidos-eventbus"
    }
  }
```


Funded by Horizon Europe Framework Programme of the European Union

Delete Rules & Rule Endpoints

Delete a rule endpoint example:

```
  DELETE https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/ruleendpoints/{rule endpoint name}
```

Delete a device example:

```
  DELETE https://{API Server URL}:{port}/apis/rules.kubeedge.io/v1/namespaces/
{namespace}/rules/{rule name}
```


Funded by Horizon Europe Framework Programme of the European Union

REFERENCES

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

[2] *Ticketmaster developer docs, https://developer.ticketmaster.com/*

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

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

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

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

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

[8] *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*

[9] *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*

[10] *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*

[11] *Andrieux, Alain, et al. "Web services agreement specification (WSAgreement)." Open grid forum. Vol. 128. No. 1. 2007*

[12] *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*

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

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

[15] *Matheu SN, Robles Enciso A, Molina Zarca A, Garcia-Carrillo D, Hernández-Ramos JL, Bernal Bernabe J, Skarmeta AF. Security Architecture for Defining and Enforcing Security Profiles in DLT/SDN-Based IoT Systems. Sensors (Basel). 2020 Mar 28;20(7):1882. doi: 10.3390/s20071882. PMID: 32231142; PMCID: PMC7180465*

Funded by Horizon Europe Framework Programme of the European Union