Next Generation Meta Operating System

D2.3 Enhancing NEMO Underlying Technology

Document Identification			
Status	Final	Due Date	30/12/2024
Version	1.0	Submission Date	31/12/2024

Related WP	WP2	Document Reference	D2.3
Related Deliverable(s)	D2.1, D2.2, D3.2	Dissemination Level (*)	PU
Lead Participant	TID	Lead Author	Luis M Contreras
Contributors	TID, TSG, COMS, INTRA, UC3M, WIND3, CMC, AEGIS, SU, UPM, STS, ATOS, SYN	Reviewers	Dimitrios Skias
			Gianluca Rizzi

Keywords:
Cybersecure Micro-services' Digital Twins, Cybersecure Federated Deep Reinforcement Learning, Federated Machine Learning, federated meta Network Cluster Controller, Time Sensitive Networking

(*) Dissemination level: **(PU)** Public, fully open, e.g., web (Deliverables flagged as public will be automatically published in CORDIS project's page). **(SEN)** Sensitive, limited under the conditions of the Grant Agreement. **(Classified EU-R)** EU RESTRICTED under the Commission Decision No2015/444. **(Classified EU-C)** EU CONFIDENTIAL under the Commission Decision No2015/444. **(Classified EU-S)** EU SECRET under the Commission Decision No2015/444.

Document Information

List of Contributors	
Name	Partner
Luis M. Contreras, Guillermo Sánchez, Alejandro Muñiz	TID
Gregor Cerar, Blaž Bertalani, Matija Cankar	COMS
Victor Gabillon	TSG
Spyridon Vantolas, Konstantinos Raftopoulos, Nikolaos Papadakis, Manos Karampinakis	AEGIS
Mika Skarp, Jose Costa-Requena	CMC
Skias Dimitrios	INTRA
G.Spanoudakis, Y.Papaefstathiou	STS
Mohamed Legheraba	SU
Borja Nogales, Iván Vidal, Francisco Valera, Luis F. González	UC3M
Alberto del Rio, Javier Serrano, David Jimenez	UPM
Gianluca Rizzi	WIND3

Document History			
Version	Date	Change editors	Changes
0.1	18/10/2024	TID	First ToC draft and work assignments
0.2	22/10/2024	TID	First Inputs in Executive Summary and section 1
	22/10/2024	COMS, TSG, TID, CMC	Definition of the final structure, first inputs and work assignment in sections 2, 3, 4 and 5.
0.3	5/11/2024	ALL	First iteration
0.4	19/11/2024	ALL	Second iteration
0.5	26/11/2024	ALL	Final contributions in sections 2, 3, 4 and 5.
	2/12/2024	ALL	Final contributions in sections 6 and 7.
0.6	17/12/2024	WIND3	First Review
0.7	20/12/2024	INTRA	Second Review
0.9	23/12/2024	TID, ALL	Review's corrections pending to be accepted
0.91	26/12/2024	TID	FINAL VERSION TO BE SUBMITTED
1.0	30/12/2024	ATOS	Quality review and final version

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Luis M Contreras (TID)	30/12/2024
Quality manager	Rosana Valle Soriano (ATOS)	30/12/2024
Project Coordinator	Enric Pages Montanera (ATOS)	30/12/2024
Technical Manager	Terpsi Velivassaki (SYN)	30/12/2024

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
AAA	Authentication, Authorization, and Accounting
AD	Architecture Description
AF	Application Function
AI	Artificial Intelligence
AMF	Access and Mobility Management Function
AMPQ	Advanced Message Queuing Protocol
API	Application Programming Interface
CFDRL	Cybersecure Federated Deep Reinforcement Learning
CI/CD	Continuous Integration & Continuous Delivery/Continuous Deployment
CMDT	Cybersecure Micro services Digital Twin
CN	Core Network
CNC	Centralised Network Configuration element
CNCF	Cloud-Native Computing Foundation
CNI	Container Network Interface
CPU	Central Processing Unit
CR	Custom Resource
DAST	Dynamic Application Security Testing
DLT	Distributed Ledger Technology
DQN	Deep Q-Network
DS TT	Device Side TSN translator
Dx.y	Deliverable number y belonging to WP x
E2E	End-to-End
EC	European Commission
GDPR	General Data Protection Regulation
gPTP	generic Precision Time Protocol
gRPC	Google Remote Procedure Call
HLA	High-Level Architecture
HW	Hardware
IAM	Identity and Access Management
IAST	Interactive Application Security Testing
IBMC	Intent-Based Migration Controller
IBS	Intent-Based System
IDCO	Inter-Cluster Connectivity Orchestrator
IMC	Intent-based Migration Controller
IoT	Internet of Things
K8s	Kubernetes
KPI	Key Performance Indicator

LCM	Life Cycle Management
LCM UI	Life Cycle Management User Interface
LLDP	Link Layer Discovery Protocol
MANO	Management and Orchestration
MEC	Multi-access Edge Computing
meta-OS	Meta-Operating System
ML	Machine Learning
mNCC	meta–Network Cluster Controller
mRA	meta-Reference Architecture
MO	meta-Orchestrator
MOCA	Monetization and Consensus-based Accountability
NAC	NEMO Access Control
NED	Network Edge Device
NFV	Network Function Virtualization
NGIoT	Next-Generation IoT
NW TT	Network TSN Translator
OS	Operating System
PA-LCM	Plugin & Apps Lifecycle Manager
PCF	Policy Control Function
PoC	Proof of Concept
PPEF	PRESS and Policies Framework
PRESS	Privacy, data pRotection, Ethics, Security & Societal
PSFP	Per-stream filtering and policing
RAN	Radio Access Network
RASP	Run-time Application Security Protection
REST	Representational State Transfer
RBAC	Role-Based Access Control
RL	Reinforcement Learning
SDK	Software Development Kit
SDN	Software Defined Networking
SDO	Standard Development Organization
SD-WAN	Software-Defined Wide Area Network
SLA	Service Level Agreement
SLO	Service Level Objective
SMF	Session Management Function
TEE	Trusted Execution Environment
TSCAI	Time Sensitive Communication Assistance Information
UPF	User Plane Function
UUID	Universally Unique Identifier
V&V	Validation & Verification
VM	Virtual Machine
VNF	Virtual Network Function

WP	Work Package
YAML	Yet Another Markup Language

Executive Summary

The main goal for this document is to outline the work conducted in the three tasks of WP2, emphasizing the components that form the underlay layer of the meta-OS of NEMO (i.e., three components plus the TSN capabilities). In here it is provided a thorough description, focusing on overcoming the initial challenges related to the development of these components as they begin to coexist within the same ecosystem. Through this discussion, the reader will gain a clear understanding of the advancements proposed by NEMO partners relative to the deployment and advances for the technologies utilized.

The document highlights the achievements made thus far during the Enhance phase, with particular attention to the overall deployments and internal integrations employed to develop and report the Proof of Concepts (PoC) for each of the main subcomponent. This phase finalizes with three of the four components (CMDT, mNCC and TSN) in an integration stage, with some functional and non-functional improvements from the previous version, and with the final stable version of the CF-DRL.

To completely understand this work it is important to have had a previous reading of the *D2.2 Enhancing NEMO Underlying Technology* [1] as this document, due to the fact that is an extension of it and it works as a continuation for the initial integration done in the previous phase.

Once finalized, the reader should end with a big-picture knowledge of the four main components depicted in here, being able to understand the motivation and functionality of each of these components and have a clear view about what to expect for the final developing phase.

The main results for this document included:

- A final approach to a stable version for the Cybersecure Micro-Services Digital Twin, with an overlook about how it is deployed and how it works.
- A final stable version of the Cybersecure Federated Deep Reinforcement Learning with a comprehension of the internal architecture and a description of each of the algorithms available to be run in the first stable version.
- A final approach to a stable version for the Federated meta-Network Cluster Controller, with an initial deployment of the service provisioning and the network monitoring and exposure capabilities.
- A final approach to a stable version of the Time Sensitive Network component for 5G networks.

As conclusion, this document depicts the main advances in the deployment of the NEMO technological enables, providing a deep understanding of how each of them works and the updated contributions already done in a concluded development and deployment phase. During the elaboration of this document some initial approaches proposed in the previous stage have been enhanced and some others have been dropped as there were considered as unnecessary or redundant. Also, the presence of external difficulties has made CMDT and mNCC suffer changes to adapt to the new consortium status, impacting these difficulties in the completeness of these components but not in a critical way.

In the current phase, it is expected to develop the final deployment and validation for each of the four components, and an improvement on the integration inside the full NEMO meta-OS ecosystem.

1 Introduction

1.1 Purpose of the document

This document works as an update from *D2.2 Enhancing NEMO Underlaying Technology* [1] ,providing an updated view of each of the three components defined in WP2. The main goal is to describe the latest version of each component and its parts, providing a complete knowledge about each of them and how is the final stable version. Additionally, it is also depicted how these components interact with other NEMO components, focusing on the functionalities provided and the interfaces exposed, being this crucial for the overall NEMO stack integration steps. Moreover, the document will serve to evaluate the latest status of the components against the NEMO KPIs. The three components described (each of them carried out in one of the Task withing the Work Package) are going to focus on topics as cloud and edge computing, networking, AI operations, and microservices-oriented architectures, being important to have a background on these technologies.

1.2 Relation to other project work

D2.3 Enhancing NEMO Underlaying Technology works as an update to *D2.2 Enhancing NEMO Underlaying Technology*, providing advancements on the technologies defined in there and giving the latest view of its implementation. D2.3 will follow the same structure and objectives as D3.2 and D4.2 providing a jointed view of NEMO architecture and implementation and working as a description of the integration work and the evaluation of Use Cases described in D5.2 NEMO Living Labs use cases evaluation results at Milestones 8 and 11.

1.3 Structure of the document

This document starts with an introduction and explanation of the context and goals defined to be addressed here. Then, there will be a section for each of the three WP2 components, and a last one for the Time Sensitive Networking, due to its characteristics. In each of these sections dedicated to the components it is provided an overview of what is being pursued in each of them, the final improved architecture, how each of these components works and the interfaces to communicate with the rest of NEMO components.

Once components are depicted in the theoretical way, there is also a section 6 for the explanation and validation of each of them, showing proofs of concept that evaluate the functionalities and bring a view on how the integration with the rest of NEMO WP2 architecture it's done. For the pending integrations with the rest of the NEMO components, proper descriptions and reporting will be included in the ongoing D4.3.

Finally, all the information expressed in sections 2 to 6 will be summarized in the Conclusions section, focusing on the results obtained related to the goals and challenges expected for the technologies depicted. Also, as this is the final document for the work package, this conclusion section will provide a big picture about the work realized and the benefits that these technologies bring not only to the rest of the NEMO architecture, also to the general State-of-the-Art.

2 Cybersecure Micro-services Digital Twins (CMDT)

2.1 Overview

The Deliverable D2.2 [1] describes all potential expected functionalities of the Cybersecure Micro-services Digital Twins (CMDT) component, which plays a pivotal role in monitoring and assessing the status and availability of each workload defined as a Digital Twin (DT). This service is responsible for aggregating data from all available clusters, organizing and enriching it with crucial metadata—such as service ownership details—and providing consolidated, actionable insights to other services.

The role and functionalities of the CMDT have improved the previous CMDT version presented in D2.2. The main changes are in the distinction of the components' goal inside the NEMO MetaOS framework, removing unnecessary functionalities and clarifying the integration interfaces of the tool. With these changes, CMDT became near real-time data fusion and distribution service, capturing and DT life-cycle anomalies and passing them to the user interface. This means that the CMDT dashboards will remain, but will be used only for NEMO super-users, while the DT information, crucial for the end user is integrated in the NEMO LCM UI.

Additionally, the CMDT integration is finally detailed, including the definition of course of the data that is passed over the AMPQ message bus (e.g., RabbitMQ). This information is crucial to transmit the status from the distributed NEMO environment to the LCM UI in a near real-time manner. The messages include the following data:

- **Timestamp:** Depicts the exact time when the data was prepared and message sent, ensuring chronological order.
- **Workload ID**: Represents a unique identifier for the workload (e.g. Digital Twin).
- **UserID**: Represents a unique identifier for the user.
- **LinkerD retrieved info** such as network traffic to/from pod, traffic rate, and response stats.
- **Kube-state-metrics info** such as memory, and CPU utilization,
- **Number of replicas**: Represents a current number of replica instances (k8s pods) running for a workload. It indicates the number of instances or replica instances.
- **Status**: Describes the current and past state of the workload or k8s pods, which can be:
    - Running
    - Restart
    - Ready
    - Failure

The message details currently cover all KPIs defined in section 2.2 of previous deliverable "*D2.2 Enhancing NEMO Underlying Technology*". The list of message data attributes is not fixed and could be updated in the future due to the stakeholder requests for better user experience. In the following subsections we point out the specific changes made on CMDT architecture in the last development and integration period.

2.2 Architecture and Approach

The initial iteration of the CMDT implementation was ambitious, incorporating numerous data sources–some of which were redundant–to gather as much information as possible and gain a comprehensive understanding of the system and its Digital Twin deployments. Following the initial deployment, we evaluated the collected data and compared the CMDT's capabilities with other NEMO services. This led us to reshape the architecture to better align with the NEMO meta-OS framework, ensuring it complemented the functionality of other services effectively.

The new version of CMDT still provides a comprehensive approach for in detail management of DT, however, the architecture is restructured in a way to guarantee seamless and real-time provision of DT status to the LCM graphical user interface (LCM UI), organized by each workload or user. In essence, this version still includes the same core functionality, without the parts as Grafana and Prometheus, which were used only to showcase the CMDT results in the past. In this final version of CMDT, which is fully integrated into the NEMO operating system, the interfaces mentioned became redundant and were removed from the core CMDT architecture.

The revised CMDT architecture is presented in Figure 1. The core services consist of **k8s Agent aggregation engine, Service Mesh Network Aggregation Engine, CMDT data fusion and broker.** Mentioned services and their processes streamline the DT status message delivery from the infrastructure and software layer to the user space in LCM UI. Both aggregation engines rely on the data gathered through special agents locally collecting the data on each pod. The reader can find these agents shown on the bottom of Figure 1, where each cluster is monitored by Service mesh monitoring and Pod monitor. The gathered and CMDT processed data is shared on the AMPQ service (RabbitMQ). The data is consumed by NEMO services, like the LCM – UI, which uses the data to represent and status of the Digital twin health.

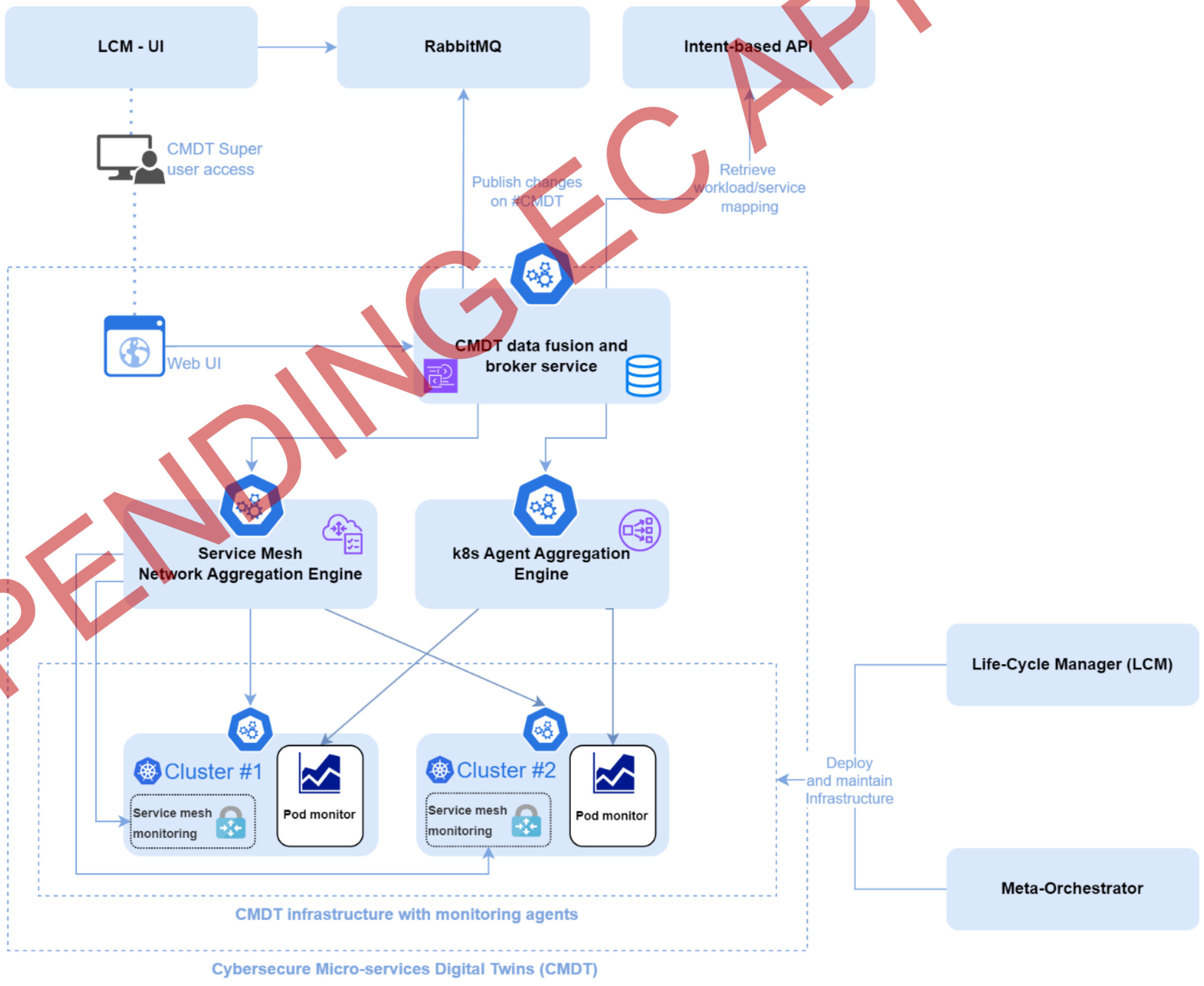

LCM - UI
RabbitMQ
Intent-based API
CMDT Super user access
Publish changes on #CMDT
Retrieve workload/service mapping
Web UI
CMDT data fusion and broker service
Service Mesh Network Aggregation Engine
k8s Agent Aggregation Engine
Life-Cycle Manager (LCM)
Cluster #1
Service mesh monitoring
Pod monitor
Cluster #2
Service mesh monitoring
Pod monitor
Deploy and maintain Infrastructure
Meta-Orchestrator
CMDT infrastructure with monitoring agents
Cybersecure Micro-services Digital Twins (CMDT)

Figure 1 CMDT architecture and integration points.

The sequence diagram from Figure 2 outlines the process of CMDT near real-time notifications for the LCM-UI within a distributed infrastructure. The LCM-UI monitors the #CMDT channel via RabbitMQ to receive updates. CMDT Core Services interact with multiple clusters (Cluster 1 to Cluster N) distributed across the infrastructure, gathering network mesh and service data from pods. This DT data is processed, aligned, annotated, and relayed to RabbitMQ as cluster status updates. RabbitMQ acts as a mediator, ensuring that LCM-UI remains informed of the latest statuses from each cluster, enabling near real-time notifications and streamlined monitoring capabilities for the system.

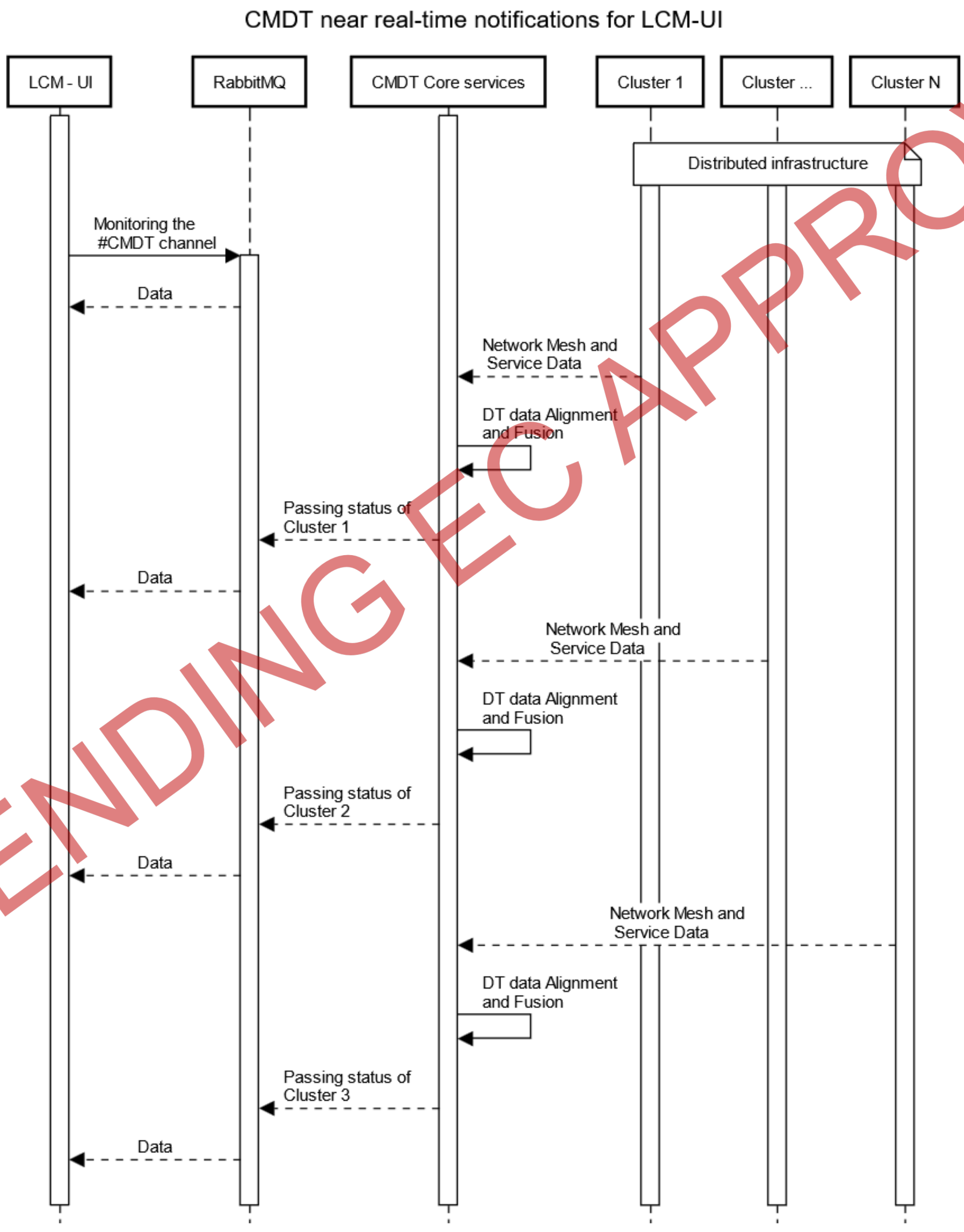

CMDT near real-time notifications for LCM-UI
LCM - UI
RabbitMQ
CMDT Core services
Cluster 1
Cluster ...
Cluster N
Distributed infrastructure
Monitoring the #CMDT channel
Data
Network Mesh and Service Data
DT data Alignment and Fusion
Passing status of Cluster 1
Data
Network Mesh and Service Data
DT data Alignment and Fusion
Passing status of Cluster 2
Data
Network Mesh and Service Data
DT data Alignment and Fusion
Passing status of Cluster 3
Data

Figure 2 CMDT data collection and passing to RabbitMQ

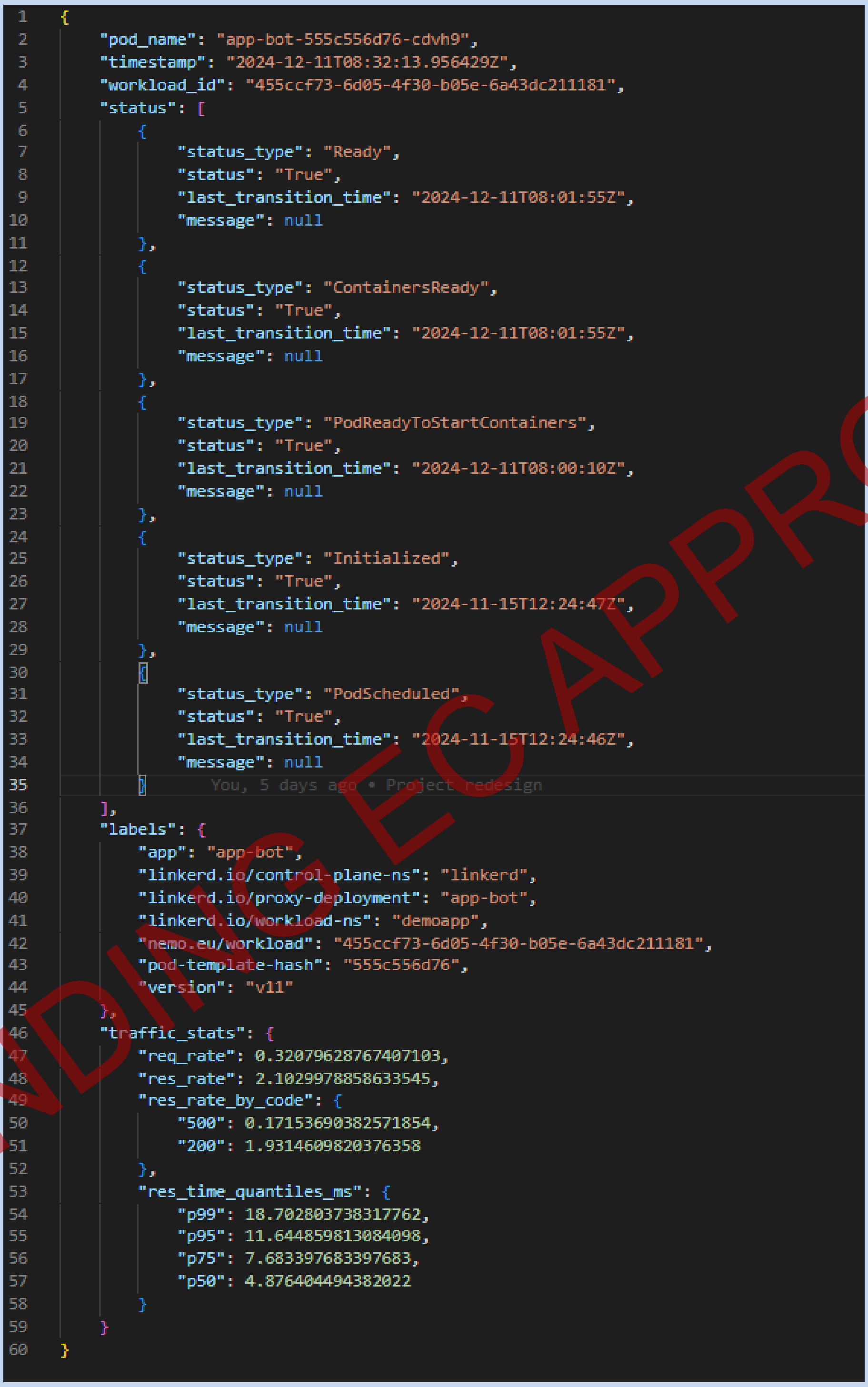
2.3 Internal and External Interfacing

CMDT uses only internal interfaces to communicate with the rest of the NEMO components. The input and output interfaces include the following information:

- **INPUT**:
  - Internal tools:
    - LinkerD (https://linkerd.io/) API to gather internal and external network traffic stats.
    - Kube-state-metrics (https://github.com/kubernetes/kube-state-metrics) API to gather Kubernetes nodes' stats (CPU, memory, storage utilization).
    - Prometheus (https://prometheus.io/) API to access internal storage to query retrospective stats.
    - Kubernetes API to gather services' and pods' state and logs.
  - Intent-based API to access mapping between pods and workload UUID.
  - User/workload relations.

- **OUTPUT**:
  - **AMPQ messages**: RabbitMQ periodic and event-driven (on events/changes) messages to be consumed by other services, such as LCM and CF-DRL.
  - **Optional GUI dashboard for NEMO**.

Message destination	Message format
<AMPQ url on RabbitMQ> Channel: #CMDT	Payload (Screeshot of JSON) The message format:



```
{
    "pod_name": "app-bot-555c556d76-cdvh9",
    "timestamp": "2024-12-11T08:32:13.956429Z",
    "workload_id": "455ccf73-6d05-4f30-b05e-6a43dc211181",
    "status": [
        {
            "status_type": "Ready",
            "status": "True",
            "last_transition_time": "2024-12-11T08:01:55Z",
            "message": null
        },
        {
            "status_type": "ContainersReady",
            "status": "True",
            "last_transition_time": "2024-12-11T08:01:55Z",
            "message": null
        },
        {
            "status_type": "PodReadyToStartContainers",
            "status": "True",
            "last_transition_time": "2024-12-11T08:00:10Z",
            "message": null
        },
        {
            "status_type": "Initialized",
            "status": "True",
            "last_transition_time": "2024-11-15T12:24:47Z",
            "message": null
        },
        {
            "status_type": "PodScheduled",
            "status": "True",
            "last_transition_time": "2024-11-15T12:24:46Z",
            "message": null
        }       You, 5 days ago • Project redesign
    ],
    "labels": {
        "app": "app-bot",
        "linkerd.io/control-plane-ns": "linkerd",
        "linkerd.io/proxy-deployment": "app-bot",
        "linkerd.io/workload-ns": "demoapp",
        "nemo.eu/workload": "455ccf73-6d05-4f30-b05e-6a43dc211181",
        "pod-template-hash": "555c556d76",
        "version": "v11"
    },
    "traffic_stats": {
        "req_rate": 0.32079628767407103,
        "res_rate": 2.1029978858633545,
        "res_rate_by_code": {
            "500": 0.17153690382571854,
            "200": 1.9314609820376358
        },
        "res_time_quantiles_ms": {
            "p99": 18.702803738317762,
            "p95": 11.644859813084098,
            "p75": 7.683397683397683,
            "p50": 4.876404494382022
        }
    }
}
```

Figure 3 Current version of CMDT AMPQ message

2.4 Conclusion

The CMDT has significantly changed in the last development period. The key improvement was the definition of the essential content gathered and annotated by CMDT and the most efficient and useful integration point through AMPQ.

Current simulations show that the LCM-UI will be able to gain responsiveness and near real-time digital twin visualization by using the CMDT notifications. For advanced NEMO administration users, the CMDT will still maintain the backdoor access for easier status debugging.

3 Cybersecure Federated Deep Reinforcement Learning (CF-DRL)

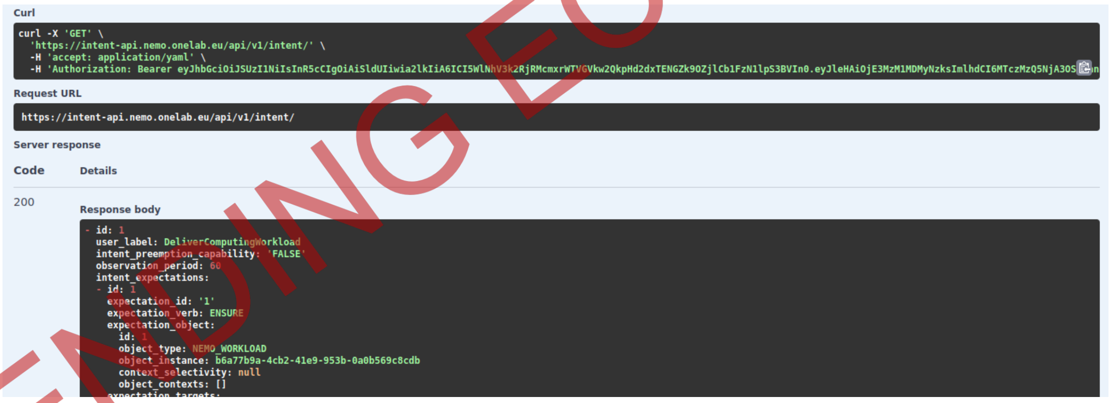
3.1 Overview

One of the main applications of CF-DRL within NEMO is to help orchestrate microservices in the Meta-Orchestrator as described in D2.2

3.2 Architecture and Approach

3.2.1 CFDRL and Meta Orchestration

- **Designing a Deep Q-Network (DQN) Algorithm**. Because the Thales library for reinforcement learning is closed source, it was not possible to share the library as is on the One lab cluster. However, it has been recoded the specific DQN algorithm for the CFDRL-NEMO application.
- **Connection to PPEF**: The PPEF can provide information about the status of the workload that concern the workload's intents' expectations set by the NEMO user. CFDRL needs to query the intent-based API automatically using a Keycloack identifier. The CFDRL queries the API at *intent-api.nemo.onelab.eu/api*. Then the information is retrieved and filtered in order to extract the relevant information. In the case of the PPEF, the relevant information[1] is the usage of CPU and RAM and whether it exceeds the limits set for a normal execution.



Curl

```
curl -X 'GET' \
  'https://intent-api.nemo.onelab.eu/api/v1/intent/' \
  -H 'accept: application/yaml' \
  -H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5WlNhV3k2RjRMcmxrWTVGVkw2QkpHd2dxTENGZk9OZjlCb1FzN1lpS3BVIn0.eyJleHAiOjE3MzM1MDMyNzksImlhdCI6MTczMzQ5NjA3OS
```

Request URL

```
https://intent-api.nemo.onelab.eu/api/v1/intent/
```

Server response

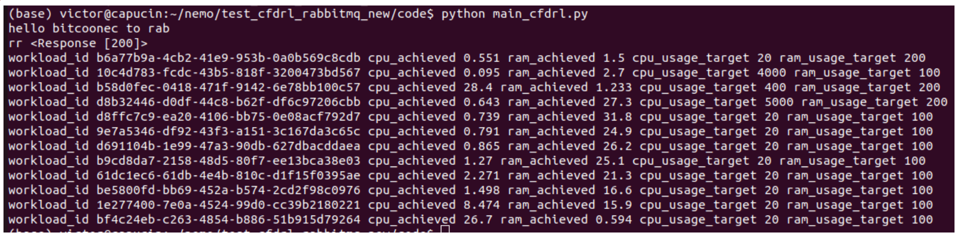
Code Details

200

Response body

```
- id: 1
  user_label: DeliverComputingWorkload
  intent_preemption_capability: 'FALSE'
  observation_period: 60
  intent_expectations:
  - id: 1
    expectation_id: '1'
    expectation_verb: ENSURE
    expectation_object:
      id: 1
      object_type: NEMO_WORKLOAD
      object_instance: b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb
      context_selectivity: null
      object_contexts: []
```

Figure 4 Intent-based API interface that contains PPEF information

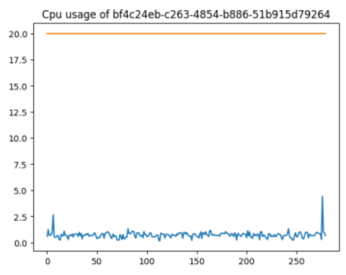


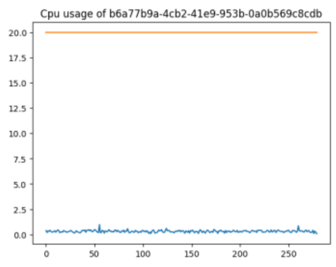
```
(base) victor@capucin:~/nemo/test_cfdrl_rabbitmq_new/code$ python main_cfdrl.py
hello bitcoonec to rab
rr <Response [200]>
workload_id b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb cpu_achieved 0.551 ram_achieved 1.5 cpu_usage_target 20 ram_usage_target 200
workload_id 10c4d783-fcdc-43b5-818f-3200473bd567 cpu_achieved 0.095 ram_achieved 2.7 cpu_usage_target 4000 ram_usage_target 100
workload_id b58d0fec-0418-471f-9142-6e78bb100c57 cpu_achieved 28.4 ram_achieved 1.233 cpu_usage_target 400 ram_usage_target 200
workload_id d8b32446-d0df-44c8-b62f-df6c97206cbb cpu_achieved 0.643 ram_achieved 27.3 cpu_usage_target 5000 ram_usage_target 200
workload_id d8ffc7c9-ea20-4106-bb75-0e08acf792d7 cpu_achieved 0.739 ram_achieved 31.8 cpu_usage_target 20 ram_usage_target 100
workload_id 9e7a5346-df92-43f3-a151-3c167da3c65c cpu_achieved 0.791 ram_achieved 24.9 cpu_usage_target 20 ram_usage_target 100
workload_id d691104b-1e99-47a3-90db-627dbacddaea cpu_achieved 0.865 ram_achieved 26.2 cpu_usage_target 20 ram_usage_target 100
workload_id b9cd8da7-2158-48d5-80f7-ee13bca38e03 cpu_achieved 1.27 ram_achieved 25.1 cpu_usage_target 20 ram_usage_target 100
workload_id 61dc1ec6-61db-4e4b-810c-d1f15f0395ae cpu_achieved 2.271 ram_achieved 21.3 cpu_usage_target 20 ram_usage_target 100
workload_id be5800fd-bb69-452a-b574-2cd2f98c0976 cpu_achieved 1.498 ram_achieved 16.6 cpu_usage_target 20 ram_usage_target 100
workload_id 1e277400-7e0a-4524-99d0-cc39b2180221 cpu_achieved 8.474 ram_achieved 15.9 cpu_usage_target 20 ram_usage_target 100
workload_id bf4c24eb-c263-4854-b886-51b915d79264 cpu_achieved 26.7 ram_achieved 0.594 cpu_usage_target 20 ram_usage_target 100
```

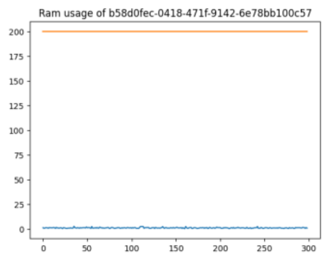
Figure 5 Result of querying the API (filtering RAM and CPU usage of the workload of interest)

[1] The information from the PPEF is not only limited to the CPU and RAM, but intents also include other details such as compute expectations (CPU, RAM), energy expectations, security, etc.

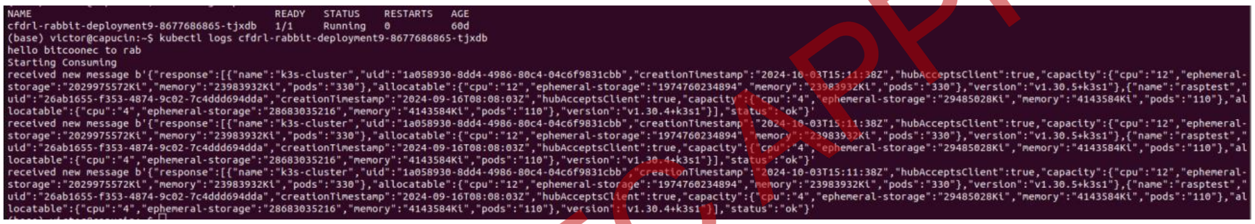
It’s possible to follow over time the variations of the values of the CPU usage and RAM usage with respect to the maximum target values in the following figures. One thing to notice is that the usage values in blue here do not go over the limit target value in orange.







- **Connection with the MO**: The connection with the meta-orchestrator is through RabbitMQ. On the CFDRL side the connection is made in python using the pika library. A queue was set in the RabbitMQ so that CFDRL can query it for general info about static cluster resource/specification information (such as its CPU, RAM, storage, etc), as illustrated in Figure 6 below. CFDRL obtains information about the cluster id, capacity such as CPU and memory.



```
NAME                                      READY   STATUS    RESTARTS   AGE
cfdrl-rabbit-deployment9-8677686865-tjxdb   1/1     Running   0          60d
(base) victor@capucin:~$ kubectl logs cfdrl-rabbit-deployment9-8677686865-tjxdb
hello bitcoonec to rab
Starting Consuming
received new message b'{"response":[{"name":"k3s-cluster","uid":"1a058930-8dd4-4986-80c4-04c6f9831cbb","creationTimestamp":"2024-10-03T15:11:38Z","hubAcceptsClient":true,"capacity":{"cpu":"12","ephemeral-
storage":"2029975572Ki","memory":"23983932Ki","pods":"330"},"allocatable":{"cpu":"12","ephemeral-storage":"197476023489 4","memory":"23983932Ki","pods":"330"},"version":"v1.30.5+k3s1"},{"name":"rasptest","
uid":"26ab1655-f353-4874-9c02-7c4ddd694dda","creationTimestamp":"2024-09-16T08:08:03Z","hubAcceptsClient":true,"capacity":{"cpu":"4","ephemeral-storage":"29485028Ki","memory":"4143584Ki","pods":"110"},"al
locatable":{"cpu":"4","ephemeral-storage":"28683035216","memory":"4143584Ki","pods":"110"},"version":"v1.30.4+k3s1"}],"status":"ok"}'
received new message b'{"response":[{"name":"k3s-cluster","uid":"1a058930-8dd4-4986-80c4-04c6f9831cbb","creationTimestamp":"2024-10-03T15:11:38Z","hubAcceptsClient":true,"capacity":{"cpu":"12","ephemeral-
storage":"2029975572Ki","memory":"23983932Ki","pods":"330"},"allocatable":{"cpu":"12","ephemeral-storage":"197476023489 4","memory":"23983932Ki","pods":"330"},"version":"v1.30.5+k3s1"},{"name":"rasptest","
uid":"26ab1655-f353-4874-9c02-7c4ddd694dda","creationTimestamp":"2024-09-16T08:08:03Z","hubAcceptsClient":true,"capacity":{"cpu":"4","ephemeral-storage":"29485028Ki","memory":"4143584Ki","pods":"110"},"al
locatable":{"cpu":"4","ephemeral-storage":"28683035216","memory":"4143584Ki","pods":"110"},"version":"v1.30.4+k3s1"}],"status":"ok"}'
received new message b'{"response":[{"name":"k3s-cluster","uid":"1a058930-8dd4-4986-80c4-04c6f9831cbb","creationTimestamp":"2024-10-03T15:11:38Z","hubAcceptsClient":true,"capacity":{"cpu":"12","ephemeral-
storage":"2029975572Ki","memory":"23983932Ki","pods":"330"},"allocatable":{"cpu":"12","ephemeral-storage":"197476023489 4","memory":"23983932Ki","pods":"330"},"version":"v1.30.5+k3s1"},{"name":"rasptest","
uid":"26ab1655-f353-4874-9c02-7c4ddd694dda","creationTimestamp":"2024-09-16T08:08:03Z","hubAcceptsClient":true,"capacity":{"cpu":"4","ephemeral-storage":"29485028Ki","memory":"4143584Ki","pods":"110"},"al
locatable":{"cpu":"4","ephemeral-storage":"28683035216","memory":"4143584Ki","pods":"110"},"version":"v1.30.4+k3s1"}],"status":"ok"}'
```

Figure 6 Retrieving the information from the Meta orchestrator through Rabbit MQ

3.3 Conclusion

The CFDRL was uploaded to the Onelab cluster and connected to the Meta Orchestrator and the PPEF so that the RL model could learn from interacting with the monitoring data.

4 Federated meta-Network Cluster Controller (mNCC)

4.1 Overview

The mNCC module is tasked with managing network connectivity and the assessment and exportation of network characteristics within the NEMO system. It serves as an intermediary between NEMO and the underlying physical network topology, providing connectivity and abstracting network services from the deployment specifics. The mNCC enables real-time access to network characteristics, allowing services to autonomously update their network perspective.

Recent updates to the mNCC have significantly enhanced its capabilities and integration within the NEMO ecosystem. The first stable version has been successfully integrated, marking a major milestone in the module's development. The intent-based library has been expanded with additional service libraries, broadening the range of network functionalities that can be expressed through high-level intents. Furthermore, the integration of expected technology adaptors has been completed, enabling the mNCC to interface with a wider array of network technologies seamlessly.

Moreover, the integration of the metric reporting and exposure capabilities has been completed, allowing for more comprehensive network monitoring and analysis. This update enables the mNCC to provide more accurate and detailed network characteristics to other NEMO components. The achievement of automated deployment, supported by the MO and the CI/CD frameworks in NEMO, represents a significant step forward in operational efficiency, streamlining the process of implementing and scaling the mNCC across various network environments.

The interaction of the different modules of the component is represented in Figure 7.

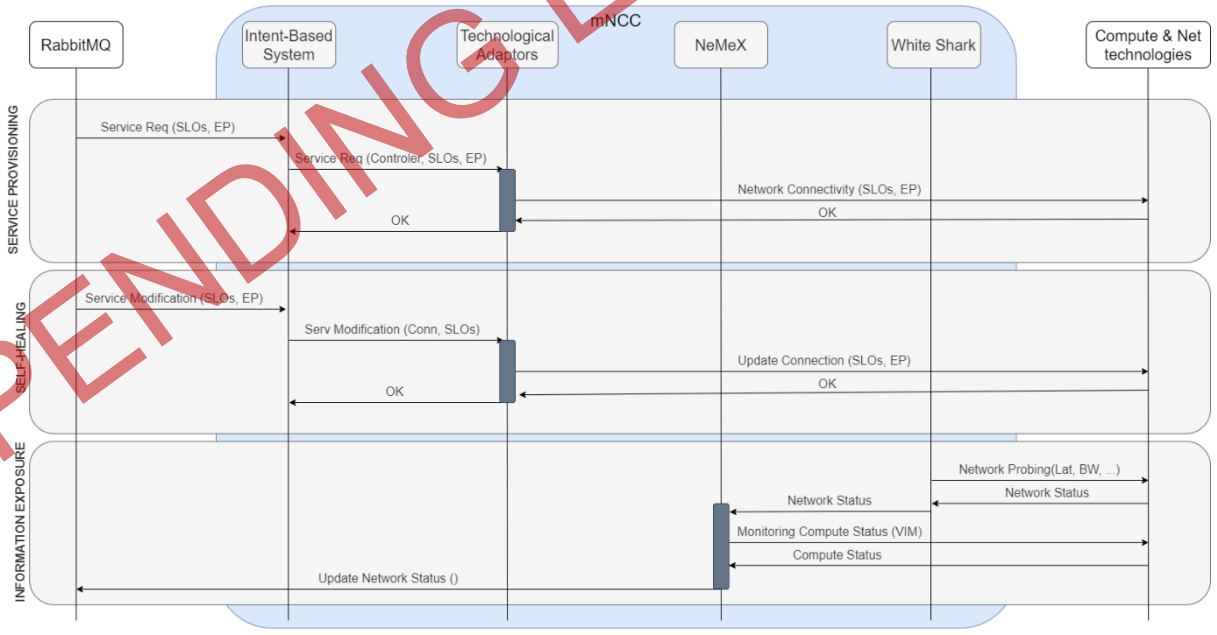

RabbitMQ
mNCC
Intent-Based System
Technological Adaptors
NeMeX
White Shark
Compute & Net technologies
SERVICE PROVISIONING
Service Req (SLOs, EP)
Service Req (Controler, SLOs, EP)
Network Connectivity (SLOs, EP)
OK
OK
SELF-HEALING
Service Modification (SLOs, EP)
Serv Modification (Conn, SLOs)
Update Connection (SLOs, EP)
OK
OK
INFORMATION EXPOSURE
Network Probing(Lat, BW, ...)
Network Status
Network Status
Monitoring Compute Status (VIM)
Compute Status
Update Network Status ()

Figure 7 Retrieving the information from the Meta orchestrator through Rabbit MQ

Document name:	D2.3 Enhancing NEMO Underlying Technology					Page:	20 of 60
Reference:	D2.3	Dissemination:	PU	Version:	1.0	Status:	Final

4.2 Architecture and Approach

In this section, are described the components of the mNCC. These include the Intent-Based System (IBS), which translates network requests into specific technology implementations, and the Network Metrics Exposure (NeMeX), which integrates and exports network topology and performance metrics. The mNCC also incorporates Technology Adaptors, such as the 5G Adapter and Teraflow SDN Controller, which manage specific network technologies. The Connectivity Controller, based on L2SM technology, provides secure link-layer connectivity between microservices within Kubernetes clusters.

4.2.1 Intent-Based System

4.2.1.1 Introduction

The Intent Based System (IBS) is a translator between different technologies, abstracting the specific details of network components from the general directives of higher-level components. This abstract order is known as Intent (an intention). It expresses an expectation of what the underlying technologies should do without knowing "the how". This concept adapts nicely with the paradigm of the mNCC to be able to scale with new Network Adaptors. Just with the addition of a new intent library, the IBS can translate an intention to a new network technology.

In the following section, there is a general overview of recent updates and the current status of the software. Additionally, how the Meta-Orchestrator (MO) utilizes intents to request different types of networks and the work in progress of the integration will be explored. This insight into the system's evolution and functionality will demonstrate how the Intent Based System continues to enhance network management and orchestration capabilities.

4.2.1.2 Intent structure

In this section, the different targets, context and expectations that will trigger the different network adaptors in the mNCC will be described. Notice that the intent structure has not received any update from the version reported in [D2.2], the structure defined the 3GPP specification [3gpp 28.321]. For the sake of simplicity, not all the objects will be described. Also, the targets, contexts and expectations will be divided in tables, although the intent requests must have the complete structure defined in the Gitlab repository[2].

Table 1: objectContext for 5G adapter

objectContexts	objectType	contextAttribute	contextCondition	contextValueRange
	5G_SLICE_FLOW	ip4Address	IS_EQUAL_TO	ip string
		type	IS_EQUAL_TO	'BOTH'
		portNumber	IS_EQUAL_TO	port string
		portType	IS_EQUAL_TO	'UDP' / 'TCP'

[2]Gitlab: https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-infrastructure-management/federated-meta-network-cluster-controller/intent-based-system

Table 2: expectationTarget for 5G adapter

	objectType	targetName	targetCondition	targetValueRange
expectationTargets	5G_SLICE_FLOW	ulCapacity	IS_EQUAL_TO	ip string
		dlCapacity	IS_EQUAL_TO	‘BOTH’

Table 3: targetContext for 5G adapter

	targetName	contextAttribute	contextCondition	contextValueRange
targetContexts	dlCapacity/ulCapacity	profile	IS_EQUAL_TO	profile string (audio4k)

Table 4: expectationContext for 5G adapter

	objectType	contextAttribute	contextCondition	contextValueRange
expectatinContexts	5G_SLICE_FLOW	startTime	IS_EQUAL_TO	RFC 3339 – Date/time format
		stopTime	IS_EQUAL_TO	RFC 3339 – Date/time format
		url	IS_EQUAL_TO	url string

Table 5: intentContext for all adapters

	userLabel	contextAttribute	contextCondition	contextValueRange
intentContexts	Cloud_continuum (NEMO)	NEMO_WORKLOAD	IS_EQUAL_TO	NEMO uuid

Table 6: objecttContext for L2SM adapter

	objectType	contextAttribute	contextCondition	contextValueRange
objectContexts	K8S_L2_NETWORK	name	IS_EQUAL_TO	network string
		providerName	IS_EQUAL_TO	provider string
		uc3m	IS_EQUAL_TO	domain string

Table 7: objrctContext for L2SM adapter

	objectType	contextAttribute	contextCondition	contextValueRange
objectContexts	K8S_CLUSTER_CONFIG	name	IS_EQUAL_TO	network string
		beare_token	IS_EQUAL_TO	token string
		api_key	IS_EQUAL_TO	key string

Table 8: expectationContext for L2SM adapter

	objectType	contextAttribute	contextCondition	contextValueRange
expectationContexts	K8S_CLUSTER_CONFIG	k8s_l2_network	IS_EQUAL_TO	network string
		url	IS_EQUAL_TO	url string

The complete set of possible intents and examples continue in the annexes 9.1 and 9.2.

4.2.1.3 Intent lifecycle

The intent lifecycle starts with the request of the Meta-Orchestrator via the RabbitMQ queue. Once the IBS consumes the intent, it starts the classification phase. In this phase, the classifier module of the IBS compares the key attributes of the intent and matches them with one of the Intent libraries installed [D2.2]. The classification has been updated with a better subdivision of incoming intents into sub-intents. Also, the IBS can process several intents in the same file separated with the usual "yaml" separator "---". This enables in the MO more flexibility and scalability in the requests. In the current version, the IBS can classify an intent in the cloud continuum library and then sub divide this intent to be processed by one of the libraries capable of translating the intent into the specific technological network adaptor.

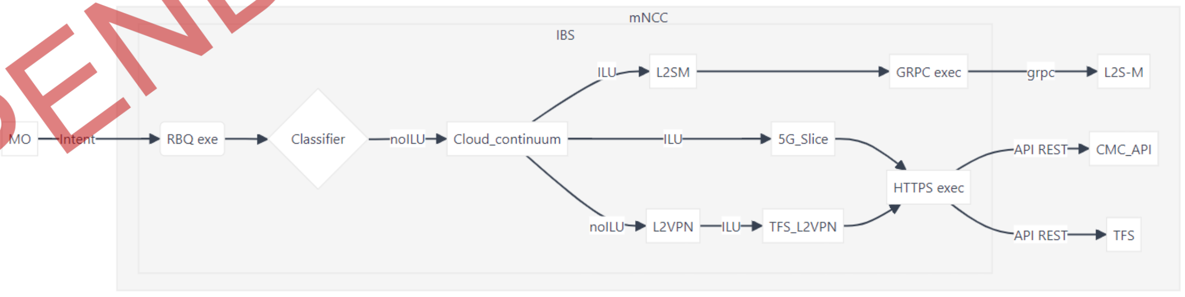


Figure 8 Intent workflow schema

Then, once the intent is translated by the corresponding library, the order is sent to the corresponding executioners. As they are programmed to be interfaces towards the different technologies, if the interfaces are the same, the executioners can be reused, giving the IBS flexibility and independence between the different steps in the lifecycle.

4.2.1.4 Intent model in NEMO cloud continuum paradigm

The implementation of the Intent Based System (IBS) as a micro-kernel architecture significantly enhances the development and integration of new network adaptors. This modular design separates core functionality from specific network adaptor implementations, allowing for seamless integration of new technologies without modifying the central IBS components. As new network technologies emerge, corresponding adaptors can be developed and plugged into the system with minimal disruption to existing operations, ensuring scalability and future proofing. The micro-kernel approach also simplifies maintenance and updates of individual components, including network adaptors. Furthermore, it offers organizations the flexibility to develop or modify network adaptors to suit their specific needs without altering the core IBS functionality. This architectural choice ultimately ensures that the IBS remains highly adaptable, capable of evolving alongside the rapidly changing landscape of network technologies in the cloud continuum paradigm.

4.2.2 Network Metrics Exposure

As defined in previous deliverables, the **Network Metrics Exposure (NeMeX)** is the component serving as the North-Bound Interface (NBI) for **mNCC**. It interacts with the internal subcomponents of mNCC and the RabbitMQ queue to export network topology and performance metrics for the managed technological domains.

NeMeX has two main functions:

1. **Topology view integration**: it gathers information from multiple sources and combines it into a unified, coherent topology view.
2. **Data modeling and performance metrics delivery**: it uses RabbitMQ to share information with other NEMO components, ensuring scalability and interoperability for the integration of technological controllers or add-ons in a standardized manner.

In its initial deployment, NeMeX establishes a connection with NEMO's RabbitMQ broker. While this version is manually configured, future versions aim to automate this process to enhance scalability, availability, and robustness. Once connected, NeMeX maps multi-domain resources to create an integrated view that combines performance values with domain-specific resources.

The process transforms three initial views into a unified topology by assigning common identifiers to the same nodes. Every *D* seconds, updated values are sent to the designated RabbitMQ queue if changes occur.

4.2.2.1 Technological dependencies and requisites

The NeMeX sub-component is based on the next technologies:

- **Python3**[3]. This is the programming language used for almost all code in this component. Python 3 is a high-level interpreted language designed to facilitate code usability, thereby enhancing the potential for code improvements and extensions. Currently version Python3.12 is used.
- **Kubernetes (K8S)**[4]. is used for managing distributed systems, specifically for monitoring computational capacities required for virtualizing network resources. This choice is based on the strong presence of Kubernetes in current systems and its compatibility with other

[3] Python3 official release: https://www.python.org/download/releases/3.0/
[4] Official K8S documentation: https://kubernetes.io/docs/home/

components of the NEMO meta-OS, from Kubernetes API we extract Node information and launch the sub-component. Currently version 1.31 is used.

Internally with NEMO, there are also some dependencies:

- **Underlay Network Probes**. NeMeX requires the deployment of the Underlay Network Probes to obtain the metrics extracted from the network. If probes are not deployed or are failing, there would be no content to expose. This would not cause a crash in the system, but the behavior would not be the expected one for the main workflow.
- **RabbitMQ**. Like what happens with the Underlay Network Probes, if there is no broker to exchange the metrics, NeMeX will be unable to push them, and therefore the behavior will not be the desired one. Also, it would neither kill NeMeX, just stopping the exposition service.

Also, there are some Python libraries also used and needed in this deployment. Although there is no need for a specific version in each of those technologies, it may be interesting to mention them just in case new implementations of those technologies include incompatibilities:

```
requests~=2.25
pika~=1.0
kubernetes~=31.0
```

About the System dependencies, all code has been tested and executed in Linux (Ubuntu 22.04 LTS) but, as the code is deployed using Kubernetes and Python3 (an interpreted language) there is no special dependency on the system to be used. Also, there are neither dependencies with hardware capabilities.

Finally, there are also configuration dependencies. These dependencies are related to the arguments to be provided during the deployment of the component:

- **delay**: Time between updates (default: 3600)
- **cluster**: Cluster id (default: 'cluster-1')
- **rmq_ip**: IP where the RabbitMQ broker is available (default: '132.227.122.23')
- **rmq_port**: Port where the RabbitMQ broker is available (default: '30403')
- **exchange**: Exchange used int the Rabbit MQ (default: ' ')
- **key**: Key used in RabbitMQ (default: 'costs-map')
- **user**: Nemo user for pushing RabbitMQ messages (default: 'nemo-user')
- **passwd**: Password to push RabbitMQ messages (default: '1234')

These arguments should be passed using the Kubernetes manifest available in the NeMeX official repository [5].

4.2.2.2 Internal workflow

When the NeMeX execution starts, this subcomponent evaluates if it has any topology stored locally (for example, in case tests are performed and the function in charge of reading the metrics is launched more than once without deleting the local data). In case there is no topology already stored, one will be loaded natively. Initially this topology was enhanced through a graph management, to provide path

[5] kubefiles/02_nemex-deployment.yaml · main · Eclipse Research Labs / NEMO Project / NEMO Infrastructure Management / Federated Meta-Network Cluster Controller / Network Exposure module · GitLab

Computation capabilities, but due to the lack of demand for this service, we have proceeded to use local dictionaries, which consume a smaller amount of memory and have a faster management.

Once the topology update is initialized, two data sources will be evaluated: Kubernetes to obtain the mapping between the Kubernetes nodes and the IPs that represent them and the *Underlay Network Probes*. From the *Kubernetes API (client.CoreV1Api())* the node information is loaded, extracting the name and IP values from the *metadata.name* and *status.addresses* field.

Afterwards, and having these measurements, the different metrics repositories of the *Underlay Network Probes*, which are present in each node of the cluster through a Prometheus service on port 5001, are analyzed. For each node, the metrics and properties are read and parsed into the format defined for the output. If any node does not have the service available or does not provide these metrics, the error will be detected, and it will be discarded from the process.

Once we have all the metrics processed, they are integrated in the same JSON, including the timestamp of when they have been generated and the cluster from which they come, to allow the identification of these. This resulting JSON is sent to the **RabbitMQ** broker indicated in the deployment and the process is paused until the next iteration.

This process is depicted in Figure 9.

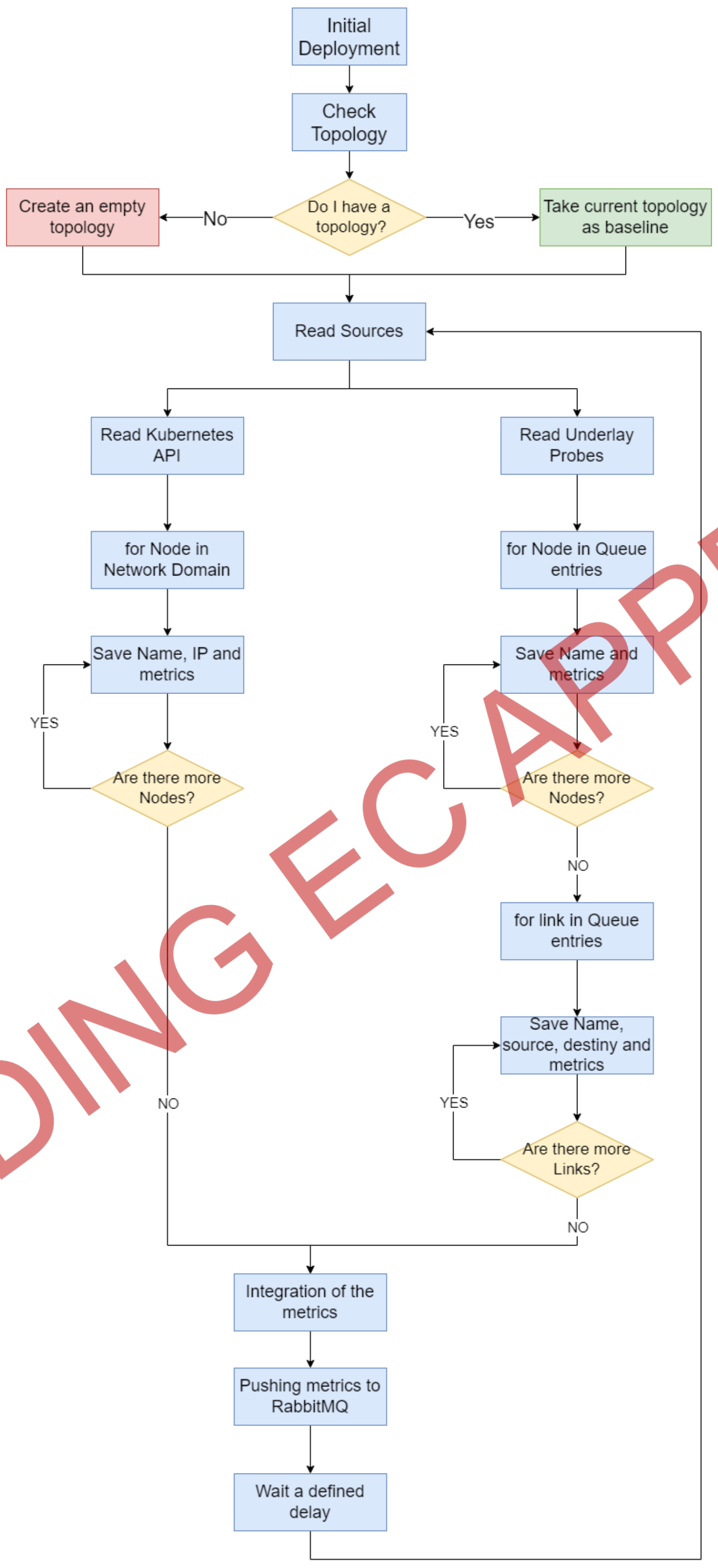


Figure 9 NeMeX internal workflow.

4.2.2.3 Deployment and permissions.

The NeMeX component is designed to be deployed in a Kubernetes environment, in which write permissions are not required but read permissions are required. In the deployment process there are two steps that must be performed sequentially: on the one hand the definition of the *Namespace* where they will be executed (in the case of the Nemo project this is **nemo-net**) and the *Deployment* of the NeMeX resources. Both documents are available in the eclipse repository [6].

Within NeMeX resources, we have 4 necessary deployments:

1. **Deployment**: this is the deployment of the code. Here the image to be instantiated is identified as well as the parameters that will serve as variables. These parameters can be parameterized or hardcoded. The user and password must be managed through Kubernetes secrets, which has not been uploaded to the repository for security reasons.
2. **ServiceAccount**: Creates an identity in Kubernetes for the deployment to manage the necessary resources for monitoring nodes.
3. **ClusterRole**: Role that NeMeX will perform within Kubernetes. In this section, read permissions are requested for the pods at namespace level and for the nodes at cluster level. This component can be replaced by a *Role*, but then it will only be able to monitor the pods at namespace level and not the nodes.
4. **ClusterRoleBinding**: Maps the permissions requested in the *Cluster Role* with the *Service Account* created. If a *Role* is used instead of a *Cluster Role*, a *RoleBinding* should be used instead.

4.2.2.4 Inputs and outputs.

We can identify two main inputs: on the one hand we have the compute information obtained using the **Kubernetes API**; this information will help to match the nodes with their IPs. This input is managed using the Python3 Kubernetes API and the value obtained is a string with IP format per each node.

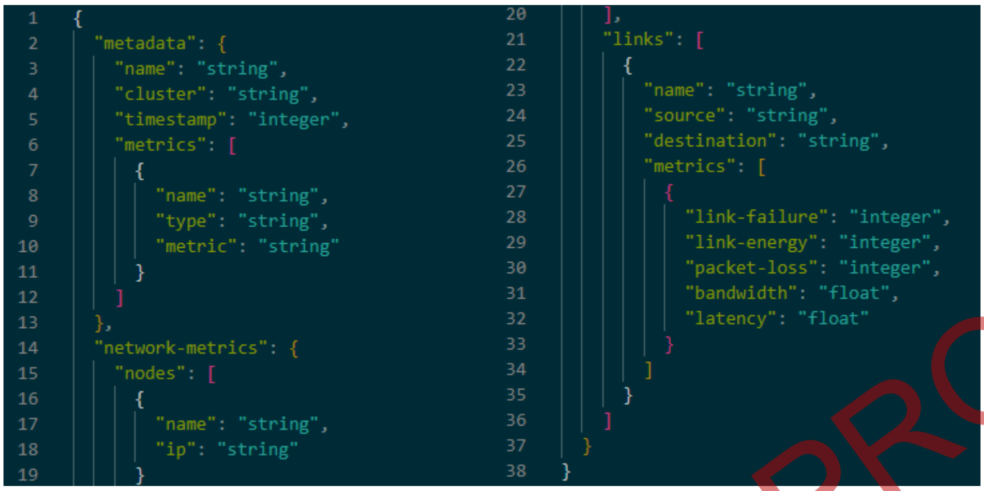
On the other hand, it is also received metric information from the network probes; these metrics will be associated with nodes and links and will be summarized in the following table:

Table 9. NeMeX input metrics.

Name	Description	Type	Valid values
network_metric_latency	Delay between two probes at IP level. Unit: ms	Float	0.01 – N
network_metric_throughput	Data rate available in the link. Unit: Mbps	Float	0.00 – N
network_metric_packet_loss	Packet loss rate since last measurement. Unit: %	Int	0 – 100
network_metric_link_energy	Average power consumption. Unit: W	Int	0 – N
network_metric_link_failure	Number of times the link drops since last measure. Unit: fails	Int	0 – N

[6] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-infrastructure-management/federated-meta-network-cluster-controller/network-exposure-module/-/tree/main/kubefiles

These metrics are processed and formatted to be aligned with the node names and IPs used by the Kubernetes resources to be managed. This integration returns the following data model:



```
{
  "metadata": {
    "name": "string",
    "cluster": "string",
    "timestamp": "integer",
    "metrics": [
      {
        "name": "string",
        "type": "string",
        "metric": "string"
      }
    ]
  },
  "network-metrics": {
    "nodes": [
      {
        "name": "string",
        "ip": "string"
      }
    ],
    "links": [
      {
        "name": "string",
        "source": "string",
        "destination": "string",
        "metrics": [
          {
            "link-failure": "integer",
            "link-energy": "integer",
            "packet-loss": "integer",
            "bandwidth": "float",
            "latency": "float"
          }
        ]
      }
    ]
  }
}
```

Figure 10 mNCC metrics data model.

In Section 6 there will be available an example of the deployment and exposure of metrics, showing a demo scenario with three nodes.

4.2.3 Technology Connectivity Adaptors

The final working network adaptors. There are two types of connectivity adaptors: the ones part of the NEMO project, and the ones external to the project but used as a proof of concept for the versatility and flexibility of the mNCC architecture. In section 6 the demonstration for two cases will be shown.

4.2.3.1 L2SM connectivity solution

Link Layer Secure Microservices (L2SM) serves as a secure connectivity solution within the mNCC, offering advanced network functionalities through an extension of the Kubernetes (K8s) API. It enables the creation, management, and deletion of virtual networks and supports the use of custom overlay networks. By taking advantage of the Software Defined Networking (SDN) capabilities, L2SM provides an API that deploys a customized data plane and enforces rules within Open Virtual Switches (OVS) as part of the overlay network topology.

L2SM has undergone significant advancements to align with the mNCC's evolving requirements. Initially, as presented in D2.2, L2SM handled internal cluster connectivity via overlays and introduced the concept of Network Edge Devices (NEDs) to enable the interconnection of overlay networks across different clusters. This concept has been further developed and implemented, facilitating communication between workloads deployed through NEMO OS using a standardized interface.

L2SM MD

The creation and management of L2SM resources are now handled via the L2SM multi-Domain (L2SM MD) client—a gRPC server that processes requests to create inter-cluster overlay topologies, add or delete clusters, and manage inter-cluster virtual networks atop these overlays. This modular approach simplifies the expansion of L2SM's functionalities, integrating existing concepts of virtual networks and overlay networks with the new NEDs and inter-cluster networking capabilities through a standardized API. This enhances the efficiency of the mNCC's connectivity adaptor.

Specifically, L2SM MD implements the following methods:

- **Create and Delete Networks:** Allows the creation and deletion of virtual networks across clusters, given a list of clusters and the name of the network.
- **Create and Delete Overlays:** Manages overlay network topologies that define the interconnection of clusters, provided with a list of clusters and a name for the overlay.
- **Add and Remove Clusters:** Enables dynamic addition or removal of clusters within an overlay network, specified by the overlay name and the cluster to modify.

To securely manage these operations, L2SM MD requires cluster information, including the K8s API endpoint and a user's bearer token with permissions to manage L2SM Custom Resource Definitions (CRDs), Overlays, NEDs, and L2Networks in the managed clusters. Since the K8s API endpoint uses HTTPS, L2SM MD needs access to the TLS certificate to establish secure communication. Therefore, copies of the public certificates are stored securely within the control plane cluster as Kubernetes secrets.

To facilitate the setup of the gRPC server and ease the management of certificates and resources, L2SM MD provides a Command Line Interface (CLI) with a set of tools:

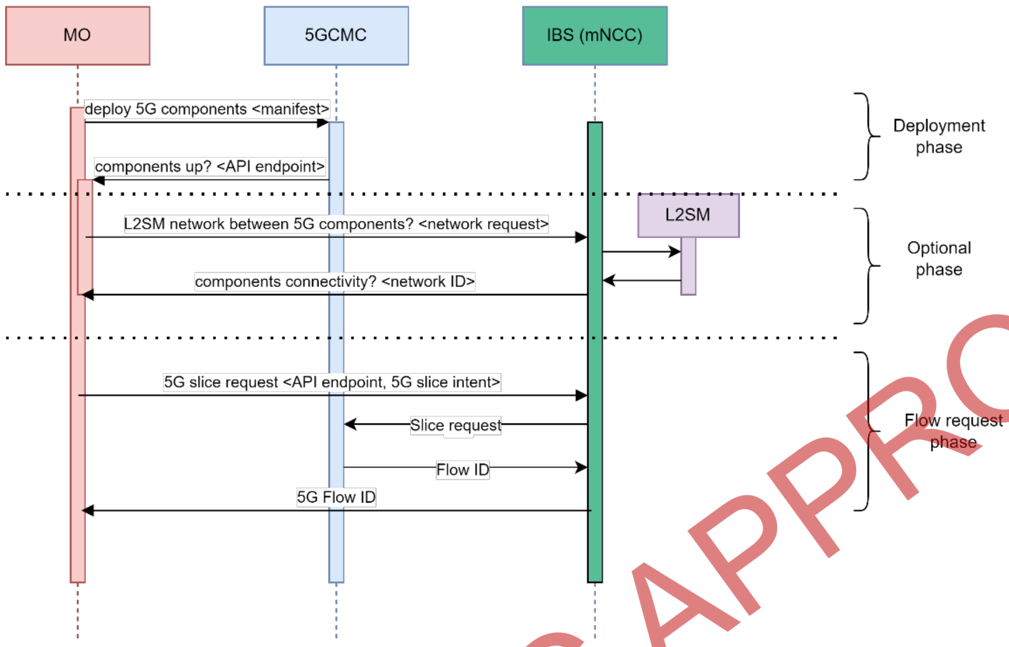
- **Apply-cert:** Takes a .key file and a cluster name as input, containing the public certificate, and stores it as a secret, allowing L2SM MD to access it securely.
- **Generate-cr:** Automates the generation of CRs by receiving input values for each resource. For example, when creating an Overlay, you input the names of clusters and links, and it generates the complete CR, including the necessary NED CRs required when connecting multiple clusters. This CLI tool mirrors the utility of the gRPC server but without actually creating the resources, serving as an alternative for quick fixes in case of errors in the workflow or for testing the adaptor in a resource-constrained environment.

4.2.3.2 5G adapter: functions, capabilities and workflow update

The 5G adapter is responsible for managing data flows and providing monitoring and analytics capabilities within the 5G network. Here are its key functions with an extended definition in D2.2:

Function	Description
Applying Data Flows	Sets up new data transmission paths for efficient routing.
Modifying Data Flows	Adjusts existing data flows due to changing network conditions or to optimize performance.
Deleting Data Flows	Removes unnecessary data flows to free up network resources.
Performance Monitoring	Tracks metrics like latency, throughput, and error rates to ensure slice performance.
Utilization Monitoring	Observes resource usage to identify underutilized or overburdened slices.

Moreover, the deployment and request of 5G network flows has been updated to achieve an automated deployment by the meta-orchestrator and the following API exposure towards the IBS. In such a way that, the MO can request a new network flow using the unified interface provided by the IBS. This workflow is not fully implemented at the date of this deliverable since MO tasks is part of WP3 and ends later in time. The general workflow is depicted on Figure 11.



MO
5GCMC
IBS (mNCC)
deploy 5G components <manifest>
components up? <API endpoint>
Deployment phase
L2SM
L2SM network between 5G components? <network request>
components connectivity? <network ID>
Optional phase
5G slice request <API endpoint, 5G slice intent>
Slice request
Flow ID
5G Flow ID
Flow request phase

Figure 11 Towards integration workflow 5G adapter

In the Figure 11 there are shown three main phases internal interactions the network adapter:

- Deployment phase: is the one orchestrated by the MO, in charge of the deployment of the 5G core components such as the UPF, AMF, SMF...
- Optional phase: This phase has been theoretically described, as there is the possibility of connecting the different components of the core if deployed in different clusters through the L2SM adapter. From the L2SM perspective, this could be treated as a common service being connected through a L2SM network.
- Flow request phase: Once the adapter is deployed, the MO can start requesting new 5G flows through the intent-based interface connected to the RabbitMQ queue specific for the mNCC.

4.2.3.3 IETF TeraFlow SDN-based connectivity adapter. (External)

The ETSI TeraFlowSDN[7] is an innovative, open-source, cloud-native Software-Defined Networking (SDN) controller and orchestrator designed to support smart networks and services for beyond 5G (B5G) and 6G networks. Developed by the ETSI Software Development Group TeraFlowSDN, it employs a micro-services architecture, enabling seamless integration with other ETSI initiatives such as OpenSourceMANO and compliance with standards from bodies like IETF and [2].

TeraFlowSDN is characterized by its high performance, advanced SDN automation, and support for transport network slicing, multi-tenancy, and cyberthreat analysis using machine learning (ML) and deep learning (DL) components. It also incorporates features like distributed ledger technology and smart contracts for secure network management. The controller is highly scalable, allowing for rapid

[7] TFS controller GitLab: https://labs.etsi.org/rep/tfs/controller/-/wikis/home

prototyping and experimentation, and it supports the management of heterogeneous network equipment, including packet optical, IP and microwave networks [3].

The latest release, TeraFlowSDN 2.1, has been used by the mNCC IBS library to create a VPN layer 2 translation. A complete example of a request and the translated version in TFS controller format is shown in Annex 7. Briefly, the following context can be used to build an intent expectation to request a layer 2 VPN.

Table 10: objectContext for L2VPN

	objectType	contextAttribute	contextCondition	contextValueRange
objectContexts	L2VPN	nodeSrc	IS_EQUAL_TO	ip string
		nodeDst	IS_EQUAL_TO	ip string
		endpointSrc	IS_EQUAL_TO	interface string
		vlanId	IS_EQUAL_TO	interface string
		niName	IS_EQUAL_TO	name string

Table 11: expextationTargets for L2VPN

	objectType	targetAttribute	targetCondition	targetValueRange
expectationTargets	L2VPN	bandwidth	IS_EQUAL_TO_OR_GREATER_THAN	float
		latency	IS_LESS_THAN	float

4.2.3.4 Network Slice Controller (External)

The IETF-based Network Slice Controller (NSC) processes end-to-end network slice requests initiated by 5G customers. These requests are coordinated by the 5G end-to-end orchestrator, which configures the Radio Access Network (RAN), and Core Network elements as needed and forwards the request to the NSC for execution. The NSC subsequently collaborates with the relevant network controllers to deploy the network slice within the transport network. [4]

There are some implementations of a network slice controller following the approach of the IETF draft (e.g., in-house Telefónica development in progress, not yet fully release as open source[8]). In the context of the project, the main characteristics that could have an intent for a slice request has been described as followed:

[8] Github: Telefonica/network_slice_controller: Repository created for the Network Slice Controller (NSC) development

	objectType	contextAttribute	contextCondition	contextValueRange
objectContexts	SLICE_SERVICE_SDP	slo-sle-policy	IS_EQUAL_TO	name string
		sdp-id	IS_EQUAL_TO	id string
		node-id	IS_EQUAL_TO	name string
		sdp-ip-address	IS_EQUAL_TO	ip string
		niName	IS_EQUAL_TO	name string

	objectType	contextAttribute	contextCondition	contextValueRange
objectTarget	SLICE_SERVICE_SDP	one-way-bandwidth	IS_EQUAL_TO	bandwidth string
		one-way-delay-maximum	IS_EQUAL_TO	delay string

4.2.3.5 Auto peering API. (External)

Auto peering, as defined in this draft [5], refers to the automated process of establishing and managing peering relationships between different Autonomous Systems (ASes). This involves using an API to exchange necessary information, such as network details, routing policies, and contact information, to facilitate the setup and maintenance of peering connections without the need for manual intervention. The goal of auto peering is to simplify and streamline the peering process, reducing the time and effort required to establish new peering relationships and ensuring more efficient and reliable interconnectivity between networks. Although literally could not be perceived as a network connectivity adaptor, it is very useful towards future adaptors that could involve BGP session to access new routes or link state parameters [6] [7].

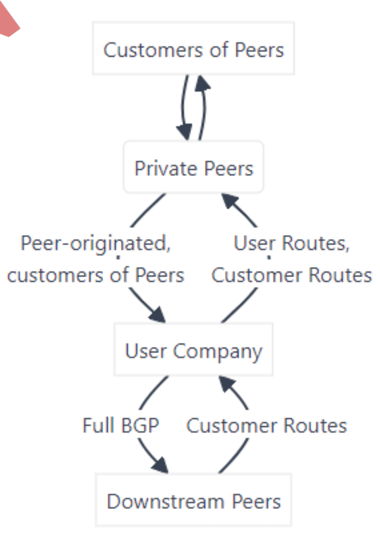


Figure 12: Private peering interactions

There is not a workable implementation of the process but, theoretically, the intents can be described accordingly[9]. The intentExpectation objects to request a new peering could contain mainly the following objectContexts:

Table 12: objectContext for BGP session

	objectType	contextAttribute	contextCondition	contextValueRange
objectContexts	BGP_SESSION	localAsn	IS_EQUAL_TO	ASn integer
		localIp	IS_EQUAL_TO	ip string
		peerAsn	IS_EQUAL_TO	ASn integer
		peerType	IS_EQUAL_TO	private
		md5	IS_EQUAL_TO	key string

Table 13: expectationContext for BGP session

	objectType	contextAttribute	contextCondition	contextValueRange
expectationContexts	BGP_SESSION	location	IS_EQUAL_TO	string

4.2.4 Network Performance Monitoring

The Network Performance Monitoring module within the mNCC provides the basic tools to evaluate real-time network conditions. This component leverages a Python-based network performance probe to measure key metrics across diverse network environments. By utilizing socket communication, it collects granular data on throughput, latency, and packet loss. These metrics are critical for ensuring efficient network operation and supporting the autonomous decision-making capabilities of other NEMO components.

The monitoring probe supports configurable parameters, allowing for fine-tuned evaluations tailored to specific network conditions or service requirements. For example, Users can define test duration, target hosts, and ports to suit their operational needs. The monitoring tool seamlessly integrates with the broader mNCC architecture, feeding network metrics directly into the meta-OS for actionable insights. Its modular design supports deployment via containerized environments, ensuring scalability and compatibility with multi-cluster scenarios. Leveraging the CI/CD framework of NEMO meta-OS, the monitoring tool is deployed and updated automatically, aligning with the intent-based management principles of the mNCC.

[9] In fact, from NEMO Project standardization contributions are being proposed for the definition of interconnection intents, e.g. https://datatracker.ietf.org/doc/draft-contreras-nmrg-interconnection-intents/.

4.3 Conclusion

The Federated Meta-Network Cluster Controller represents a key module in the NEMO meta-operating system, bringing an innovative solution for network management for edge-cloud environments. Throughout its development, the mNCC has evolved into a robust and flexible framework designed to address the challenges of scalability, adaptability and interoperability across diverse network technologies and multi-cluster scenarios.

The technological progression of the mNCC highlights its work as an infrastructure manager that simplifies and abstracts the complexities of network management. Conceptualized to cover the gap between the NEMO meta-Orchestrator and underlying network infrastructures, the mNCC has deployed its capabilities through the integration of intent-based networking and advanced metric exposure mechanisms with a secure solution for network connectivity, that works as the main technological adaptor, having also the capability to use up to five different technological adaptors according to the needed scenario (KPI 3.3, at least 5 technological adaptors). These developments have allowed the mNCC to not only provide seamless connectivity but also enable the capability of NEMO meta-OS to make dynamic and data-driven decisions.

The mNCC's architecture is built around the principles of abstraction, interoperability, and automation, enabling it to efficiently manage and monitor network operations across diverse environments. By leveraging an intent-based system, the mNCC translates high-level service expectations into detailed network configurations, allowing services to operate with greater autonomy and reduced reliance on manual intervention. This capability significantly enhances scalability, a critical requirement in distributed and multi-cluster settings, making the deployment of 50 Network Segments in 3.531s (KPI 3.1, < 5ms). Additionally, the integration of advanced metric collection and exposure mechanisms provides real-time insights into network performance. This feature supports network optimization and facilitates seamless interoperability between the mNCC and other NEMO components, such as the Meta-Orchestrator.

Looking forward, several areas for improvement and expansion have been identified to ensure the continued relevance and effectiveness of the mNCC. One priority is the <u>integration of advanced security features</u>, such as Zero Trust solutions, which will strengthen the module's defense mechanisms in distributed network scenarios. Additionally, <u>optimizing resource allocation</u> in dynamic and resource-constrained environments is an ongoing focus. Addressing these challenges, with the <u>potential introduction of AI-based engines</u>, is one of the main paths to follow during the next steps of the component.

In conclusion, the <u>mNCC has completed the main goals</u> expected for the component, even it is still at an early Technology Readiness Level (TRL), the proofs of concept done show a potential solution for multi-environment scenarios, allowing quick reactions and an abstraction level that reduces the complexity to integrate it with external modules, being the first step for seamless operation in next-generation network ecosystems.

5 Time Sensitive Networking

5.1 Overview

Time Sensitive Networking (TSN) features enable synchronizing mobile network UEs. TSN can be used to synchronize mobile network with existing IT infrastructure.

The industrial LAN may also consist of TSN-enabled Ethernet bridges. The latest release of 5G specification (Rel. 18) supports the fully centralized TSN configuration model, where a central controller should be able to configure both Ethernet and 5GS bridges as a unified network. The 5GS supports the whole industrial network, both Medium Access Control (MAC) learning and flooding based forwarding as well as the static forwarding configured by the central controller need to be supported. 3GPP has defined that a 5GS can be modelled as one or more virtual TSN bridges.

Within the context of NEMO, TSN will be a complement in the sense that the edge-cloud continuum infrastructure becomes extended with TSN domains, where the control functional entities of the 5G network, as well as other functions (e.g., content endpoints, application workloads, etc) are instantiated leveraging on NEMO infrastructure and stack.

5.2 Architecture and Approach

TSN implementation ensures centralized, precise time distribution to UEs for ultra-reliable, low-latency communication as shown in the Figure 13 below.

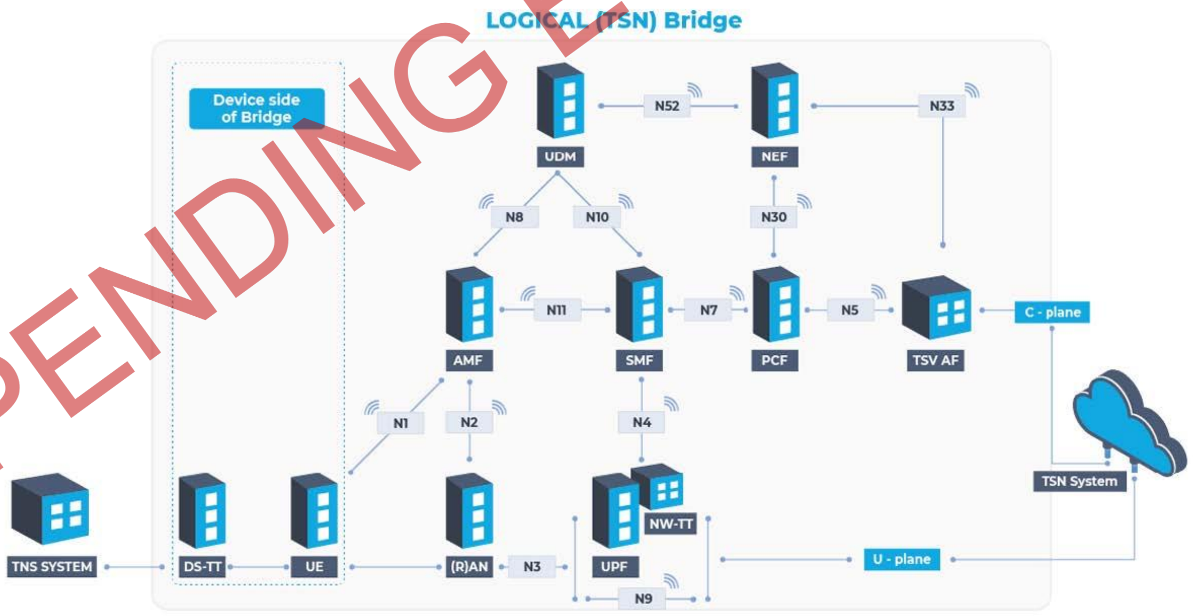


Figure 13: TSN Architecture

- CNC and TSN AF Integration: The TSN Centralized Network Controller (CNC) is responsible for configuring bridges within the TSN-enabled network, allowing timesynchronized data transfer from the Grand Master to each UE. The TSN Application Function (AF) interfaces with the 5G control plane, translating TN parameters to 5GS-specific settings and reporting the capabilities of the 5GS bridge (e.g., delay specifications and topology details).

- 5G QoS Mapping: Ethernet and TSN traffic flows are mapped to 5G QoS flows to maintain service quality. This mapping allows the CNC to optimize flows' handling across the network, supporting per-stream filtering, traffic policing, and VLAN configuration in compliance with IEEE 802.1Q.
- Seamless Redundancy and Preemption: TSN framework offers Seamless Redundancy for ultra-reliable communication and Frame Preemption to prioritize time-critical information.
- Time Synchronization: Through the TSN Translator (DS-TT) on UEs, precise time synchronization is achieved using gPTP messages, adhering to 3GPP specifications (24.535).

5.3 Internal and External Interfacing

TSN AF is part of 5GC and provides the control plane translator functionality for the integration of the 5GS with a TSN network, e.g. the interactions with the CNC. The TSN AF interfaces towards the CNC for the PSFP (IEEE Std 802.1Q) managed objects that correspond to the PSFP functionality implemented by the DS-TT and the NW-TT. Thus, when PSFP information is provided by the CNC, the TSN AF may extract relevant parameters from the PSFP configuration. The TSN AF calculates traffic pattern parameters (such as burst arrival time with reference to the ingress port and periodicity). TSN AF also obtains the flow direction. TSN AF is responsible for forwarding these parameters in TSC Assistance Information (TSCAI) Container to the SMF (via PCF). TSN AF may enable aggregation of TSN streams if the TSN streams belong to the same traffic class, terminate in the same egress port and have the same periodicity and compatible Burst arrival time. One set of parameters and one container are calculated by the TSN AF for multiple TSN streams to enable aggregation of TSN streams to the same QoS Flow.

NW TT (Network TSN Translator): supports link layer connectivity discovery and reporting as defined in IEEE Std 802.1AB for discovery of Ethernet devices attached to NW-TT. When integrating normal devices, we cannot assume that would be a DS-TT does not support link layer connectivity discovery and reporting, then NW-TT performs link layer connectivity discovery and reporting as defined in IEEE Std 802.1AB for discovery of Ethernet devices attached to DS-TT on behalf of DS-TT. If NW-TT performs link layer connectivity discovery and reporting on behalf of DS-TT, it is assumed that LLDP frames are transmitted between NW-TT and UE on the QoS Flow with the default QoS rule. Alternatively, SMF can establish a dedicated QoS Flow matching on the Ethertype defined for LLDP (IEEE Std 802.1AB).

DS-TT (Device Side TSN translator): translate the TSN 802.1 protocols on top of 5G networks but cannot be assumed that would be a DS-TT capable UEs.

5.4 Conclusion

As it points out, TSN main characteristics are performed: Seamless Redundancy that permits ultra-reliability due to the duplicity of buffers, Frame Pre-emption that includes a priority of the information that produces, and time critical communication. In addition to the Time Aware Shaper that allows scheduling of the outputs and Time Synchronization.

6 Proof of Concepts

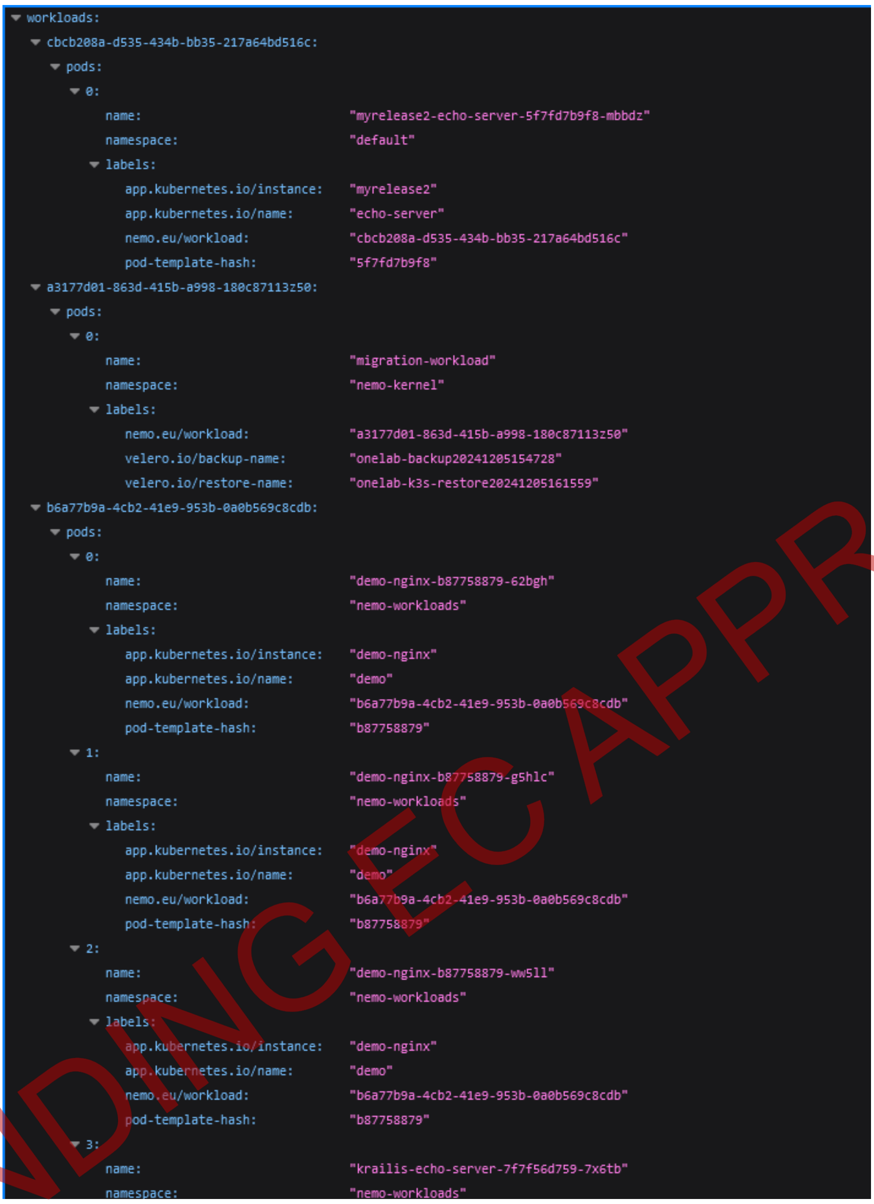
The Proof of Concepts chapter demonstrates the practical implementation and functionality of the key components developed within the NEMO project. This section showcases the integration and performance of the Cybersecure Micro-services Digital Twins (CMDT), Cybersecure Federated Deep Reinforcement Learning (CF-DRL), and the meta-Network Cluster Controller (mNCC) in various scenarios. Through detailed demonstrations and analyses, we evaluate the components against their defined Key Performance Indicators (KPIs) and illustrate their interactions within the project ecosystem. The following subsections provide in-depth insights into each component's deployment, operational workflows, and achieved results, highlighting the advancements made from the initial versions.

6.1 CMDT first stable version

6.1.1 Detailed description

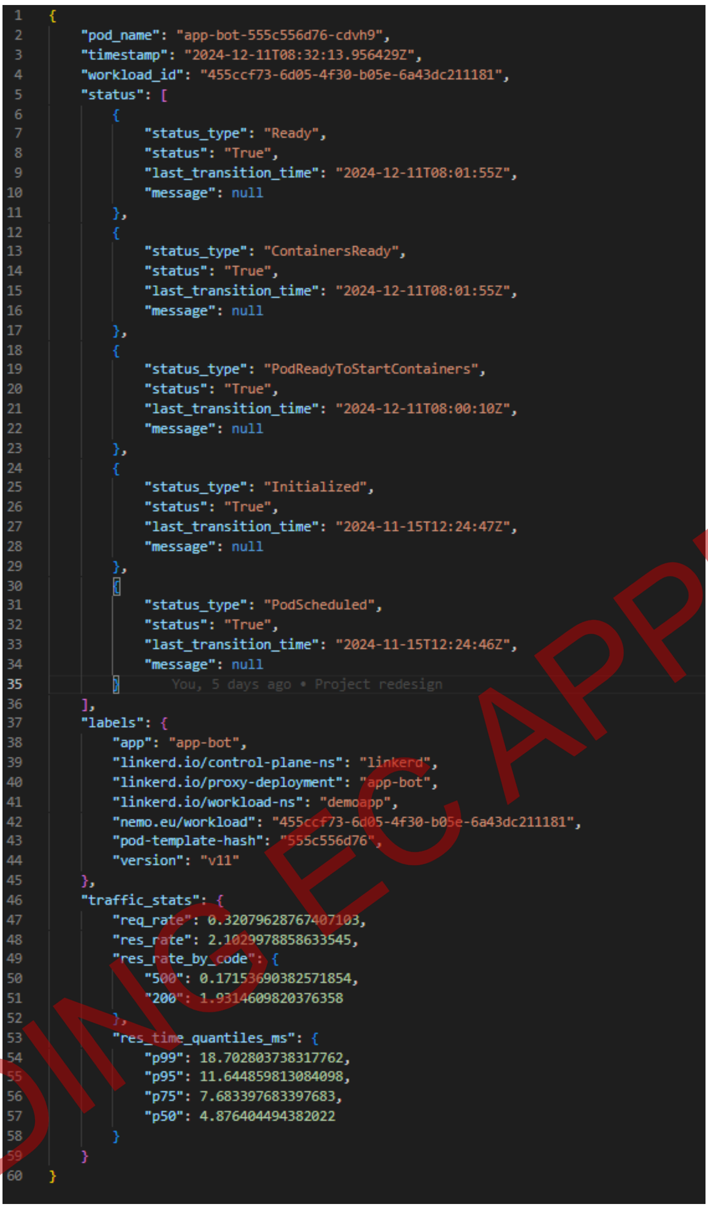
The easiest way of demonstrating the functionality of CMDT is to showcase it on a real example. For this reason, we took an application developed for NEMO Meta-OS, deploy it as digital twin and present the CMDT actions in live environment. For the app we selected the isolated ASM Terni deployment of PMU devices and applications that are controlling it. This will be our Digital Twin for this section. The simplified workload includes PMU devices deployed in the network, edge device for pre-processing the data locally on the site and cloud service, which is used to aggregate the data from edge devices and initiate actions towards the devices on the field. The application allows reviewing the status, gathering data and Firmware update over the air (FOTA). The digital twin of the workload comprises all mentioned device logs that can describe application footprint and behavior.

One crucial message that a user would request from the CMDT would be a list of services (Digital twin components) that represent a digital twin, and a corresponding status. A list of all workloads that CMDT can get from NEMO infrastructure is presented in Figure 14.



```
▼ workloads:
  ▼ cbcb208a-d535-434b-bb35-217a64bd516c:
    ▼ pods:
      ▼ 0:
          name:                           "myrelease2-echo-server-5f7fd7b9f8-mbbdz"
          namespace:                      "default"
        ▼ labels:
            app.kubernetes.io/instance:   "myrelease2"
            app.kubernetes.io/name:       "echo-server"
            nemo.eu/workload:             "cbcb208a-d535-434b-bb35-217a64bd516c"
            pod-template-hash:            "5f7fd7b9f8"
  ▼ a3177d01-863d-415b-a998-180c87113z50:
    ▼ pods:
      ▼ 0:
          name:                           "migration-workload"
          namespace:                      "nemo-kernel"
        ▼ labels:
            nemo.eu/workload:             "a3177d01-863d-415b-a998-180c87113z50"
            velero.io/backup-name:        "onelab-backup20241205154728"
            velero.io/restore-name:       "onelab-k3s-restore20241205161559"
  ▼ b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb:
    ▼ pods:
      ▼ 0:
          name:                           "demo-nginx-b87758879-62bgh"
          namespace:                      "nemo-workloads"
        ▼ labels:
            app.kubernetes.io/instance:   "demo-nginx"
            app.kubernetes.io/name:       "demo"
            nemo.eu/workload:             "b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb"
            pod-template-hash:            "b87758879"
      ▼ 1:
          name:                           "demo-nginx-b87758879-g5hlc"
          namespace:                      "nemo-workloads"
        ▼ labels:
            app.kubernetes.io/instance:   "demo-nginx"
            app.kubernetes.io/name:       "demo"
            nemo.eu/workload:             "b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb"
            pod-template-hash:            "b87758879"
      ▼ 2:
          name:                           "demo-nginx-b87758879-ww5ll"
          namespace:                      "nemo-workloads"
        ▼ labels:
            app.kubernetes.io/instance:   "demo-nginx"
            app.kubernetes.io/name:       "demo"
            nemo.eu/workload:             "b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb"
            pod-template-hash:            "b87758879"
      ▼ 3:
          name:                           "krailis-echo-server-7f7f56d759-7x6tb"
          namespace:                      "nemo-workloads"
```

Figure 14: List of resources per workload.



```
{
    "pod_name": "app-bot-555c556d76-cdvh9",
    "timestamp": "2024-12-11T08:32:13.956429Z",
    "workload_id": "455ccf73-6d05-4f30-b05e-6a43dc211181",
    "status": [
        {
            "status_type": "Ready",
            "status": "True",
            "last_transition_time": "2024-12-11T08:01:55Z",
            "message": null
        },
        {
            "status_type": "ContainersReady",
            "status": "True",
            "last_transition_time": "2024-12-11T08:01:55Z",
            "message": null
        },
        {
            "status_type": "PodReadyToStartContainers",
            "status": "True",
            "last_transition_time": "2024-12-11T08:00:10Z",
            "message": null
        },
        {
            "status_type": "Initialized",
            "status": "True",
            "last_transition_time": "2024-11-15T12:24:47Z",
            "message": null
        },
        {
            "status_type": "PodScheduled",
            "status": "True",
            "last_transition_time": "2024-11-15T12:24:46Z",
            "message": null
        }        You, 5 days ago • Project redesign
    ],
    "labels": {
        "app": "app-bot",
        "linkerd.io/control-plane-ns": "linkerd",
        "linkerd.io/proxy-deployment": "app-bot",
        "linkerd.io/workload-ns": "demoapp",
        "nemo.eu/workload": "455ccf73-6d05-4f30-b05e-6a43dc211181",
        "pod-template-hash": "555c556d76",
        "version": "v11"
    },
    "traffic_stats": {
        "req_rate": 0.32079628767407103,
        "res_rate": 2.1029978858633545,
        "res_rate_by_code": {
            "500": 0.17153690382571854,
            "200": 1.9314609820376358
        },
        "res_time_quantiles_ms": {
            "p99": 18.702803738317762,
            "p95": 11.644859813084098,
            "p75": 7.683397683397683,
            "p50": 4.876404494382022
        }
    }
}
```

Figure 15 The CMDT message example which is transported over RabbitMQ

The CMDT monitors the status of each workload component continuously and distributes this information accordingly. This represents workloads DT status, which is timestamped for each event and transmitted the LCM and LCM – UI service. This notification path is crucial to notify the user about the status of any component in the workload. The messages that are passed over the RabbitMQ are sent for each component on the #CMDT channel. Example of such CMDT message structure (Figure 15) includes all information that was already explained and defined in the Deliverable D2.2, e.i. rates of times in quantiles and rates of response codes, e.g. 200 for OK and 500 for Server Error. From "res_rate_by_code" parameter (Figure 15) user can find that our demo application two requests per minute with return the content successfully (code 200) and once per 6 minutes application responded with server error (Code 500).

An additional improvement from the last version of CMDT is the introduction of the pod status block. For higher reliability and avoiding missing the information when some services are restarted, the block includes last N status changes attributed with the timestamps. As already presented in this document, the recipient of this data is LCM UI, which creates graphics and continuously updates user with the status of his digital twin workload living on NEMO infrastructure.

In this proof of concept, we presented the CMDT operation on real-world example that is deployed in NEMO. The workload status is constantly monitored, processed, collected and sent to the LCM UI. With the demonstrated example we showed the effectiveness of the CMDT service and outline the progress from the previous deliverable D2.2.

6.2 CF-DRL first stable version

We demonstrate the run of the CFDRL for the scaling the replicas of the workloads in the Meta-Orchestrator.

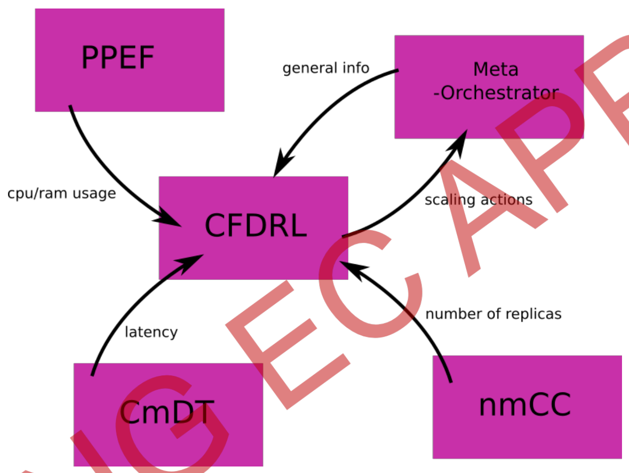


Figure 16 CFDRL communication architecture for scaling the replicas of the workloads in the Meta-Orchestrator

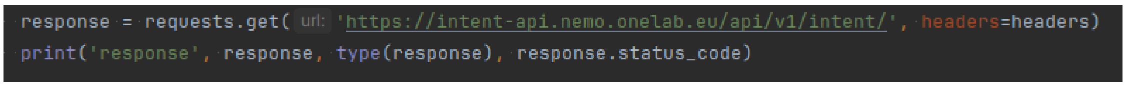
The first step is to deploy the CFDRL docker container to the Onelab cluster. Once the code is containerized. It is sent to the cluster using the commands:

```
        export KUBECONFIG="/home/victor/nemo/victor.gabillon-kubeconfig.yaml"
kubectl apply -f  -n nemo-ai
```

Then the run of the CFDRL can be monitored using the kubernetes functions:

```
        kubectl get pods -n nemo-ai
kubectl  logs cfdrl-rabbit-deployment9-8677686865-tjxdb
```

The CFDRL automatically connects to the Meta Orchestrator with RabbitMQ and with the PPEF through the intent-based API.

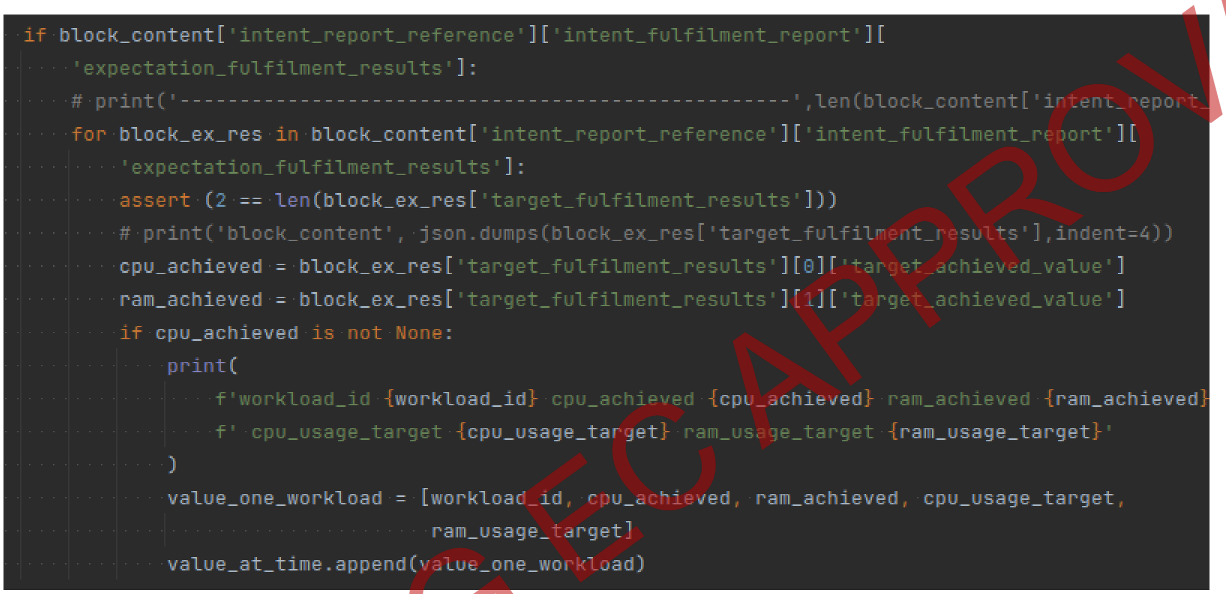


```python
response = requests.get( url: 'https://intent-api.nemo.onelab.eu/api/v1/intent/', headers=headers)
print('response', response, type(response), response.status_code)
```

Figure 17: Code snippet for request

In the code the connection in Python to the intent-based API is made witha request to https://intent-api.nemo.onelab.eu/api/v1/intent/

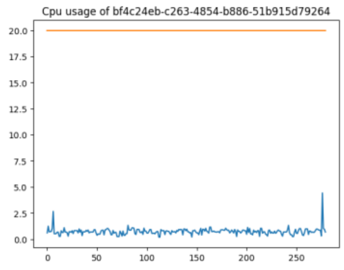
The connection to the intent-based API requires an identification token (bearer token) that expires every 2 hours. A curl request periodically is sent to the NEMO Identity Management component to re-initiate the token.



```python
if block_content['intent_report_reference']['intent_fulfilment_report'][
    'expectation_fulfilment_results']:
    # print('----------------------------------------------------',len(block_content['intent_report_
    for block_ex_res in block_content['intent_report_reference']['intent_fulfilment_report'][
        'expectation_fulfilment_results']:
        assert (2 == len(block_ex_res['target_fulfilment_results']))
        # print('block_content', json.dumps(block_ex_res['target_fulfilment_results'],indent=4))
        cpu_achieved = block_ex_res['target_fulfilment_results'][0]['target_achieved_value']
        ram_achieved = block_ex_res['target_fulfilment_results'][1]['target_achieved_value']
        if cpu_achieved is not None:
            print(
                f'workload_id {workload_id} cpu_achieved {cpu_achieved} ram_achieved {ram_achieved}
                f' cpu_usage_target {cpu_usage_target} ram_usage_target {ram_usage_target}'
            )
            value_one_workload = [workload_id, cpu_achieved, ram_achieved, cpu_usage_target,
                                  ram_usage_target]
            value_at_time.append(value_one_workload)
```

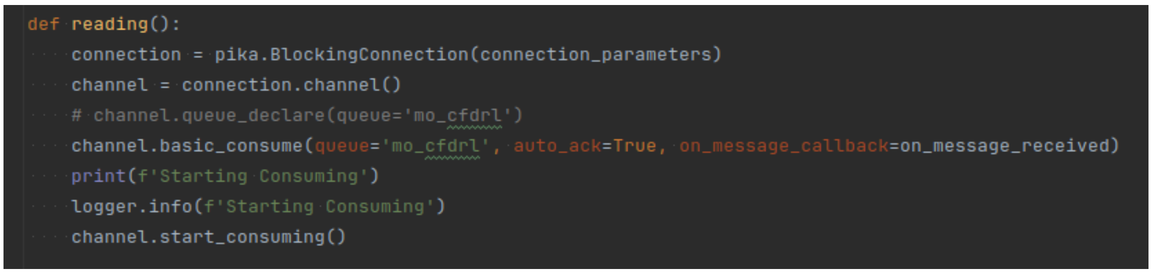
The information received from the PPEF (through the intent-based API) is filtered in order to focus on the CPU and RAM of each workloads.

This allow us to display the change of the CPU and RAM though time:



<u>Meta Orchestrator connection:</u>

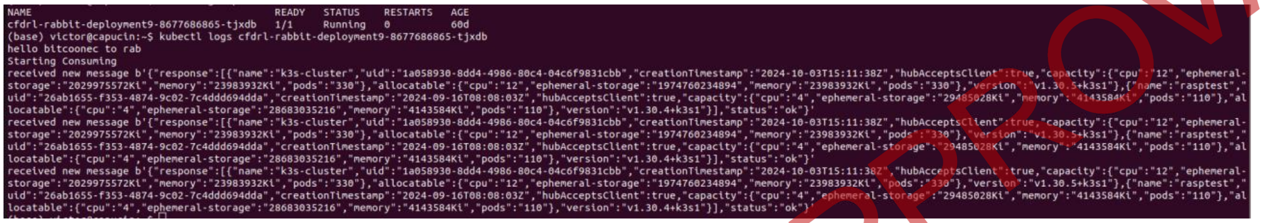
The connection with the meta-orchestrator is through RabbitMQ. On the CFDRL side the connection is made in python using the pika library.



```
def reading():
    connection = pika.BlockingConnection(connection_parameters)
    channel = connection.channel()
    # channel.queue_declare(queue='mo_cfdrl')
    channel.basic_consume(queue='mo_cfdrl', auto_ack=True, on_message_callback=on_message_received)
    print(f'Starting Consuming')
    logger.info(f'Starting Consuming')
    channel.start_consuming()
```

Figure 18: RabbitMQ connectiviy snipped

A queue was set in the RabbitMq so that CFDRL can query it for general info about the network as illustrated in the Figure below. CFDRL obtains information about the cluster id, capacity such as CPU and memory.



```
NAME                                     READY   STATUS    RESTARTS   AGE
cfdrl-rabbit-deployment9-8677686865-tjxdb   1/1     Running   0          60d
(base) victor@capucin:~$ kubectl logs cfdrl-rabbit-deployment9-8677686865-tjxdb
hello bitcoonec to rab
Starting Consuming
received new message b'{"response":[{"name":"k3s-cluster","uid":"1a058930-8dd4-4986-80c4-04c6f9831cbb","creationTimestamp":"2024-10-03T15:11:38Z","hubAcceptsClient":true,"capacity":{"cpu":"12","ephemeral-
storage":"2029975572Ki","memory":"23983932Ki","pods":"330"},"allocatable":{"cpu":"12","ephemeral-storage":"197476023489 4","memory":"23983932Ki","pods":"330"},"version":"v1.30.5+k3s1"},{"name":"rasptest","
uid":"26ab1655-f353-4874-9c02-7c4ddd694dda","creationTimestamp":"2024-09-16T08:08:03Z","hubAcceptsClient":true,"capacity":{"cpu":"4","ephemeral-storage":"29485028Ki","memory":"4143584Ki","pods":"110"},"al
locatable":{"cpu":"4","ephemeral-storage":"28683035216","memory":"4143584Ki","pods":"110"},"version":"v1.30.4+k3s1"}],"status":"ok"}'
received new message b'{"response":[{"name":"k3s-cluster","uid":"1a058930-8dd4-4986-80c4-04c6f9831cbb","creationTimestamp":"2024-10-03T15:11:38Z","hubAcceptsClient":true,"capacity":{"cpu":"12","ephemeral-
storage":"2029975572Ki","memory":"23983932Ki","pods":"330"},"allocatable":{"cpu":"12","ephemeral-storage":"197476023489 4","memory":"23983932Ki","pods":"330"},"version":"v1.30.5+k3s1"},{"name":"rasptest","
uid":"26ab1655-f353-4874-9c02-7c4ddd694dda","creationTimestamp":"2024-09-16T08:08:03Z","hubAcceptsClient":true,"capacity":{"cpu":"4","ephemeral-storage":"29485028Ki","memory":"4143584Ki","pods":"110"},"al
locatable":{"cpu":"4","ephemeral-storage":"28683035216","memory":"4143584Ki","pods":"110"},"version":"v1.30.4+k3s1"}],"status":"ok"}'
received new message b'{"response":[{"name":"k3s-cluster","uid":"1a058930-8dd4-4986-80c4-04c6f9831cbb","creationTimestamp":"2024-10-03T15:11:38Z","hubAcceptsClient":true,"capacity":{"cpu":"12","ephemeral-
storage":"2029975572Ki","memory":"23983932Ki","pods":"330"},"allocatable":{"cpu":"12","ephemeral-storage":"197476023489 4","memory":"23983932Ki","pods":"330"},"version":"v1.30.5+k3s1"},{"name":"rasptest","
uid":"26ab1655-f353-4874-9c02-7c4ddd694dda","creationTimestamp":"2024-09-16T08:08:03Z","hubAcceptsClient":true,"capacity":{"cpu":"4","ephemeral-storage":"29485028Ki","memory":"4143584Ki","pods":"110"},"al
locatable":{"cpu":"4","ephemeral-storage":"28683035216","memory":"4143584Ki","pods":"110"},"version":"v1.30.4+k3s1"}],"status":"ok"}'
```

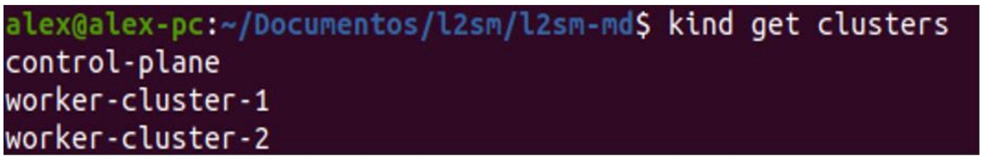
Figure 19: Metric retrieving

6.3 mNCC hybrid clusters and intent networking

6.3.1 Single Cluster Connectivity throughout L2SM overlay adaptor

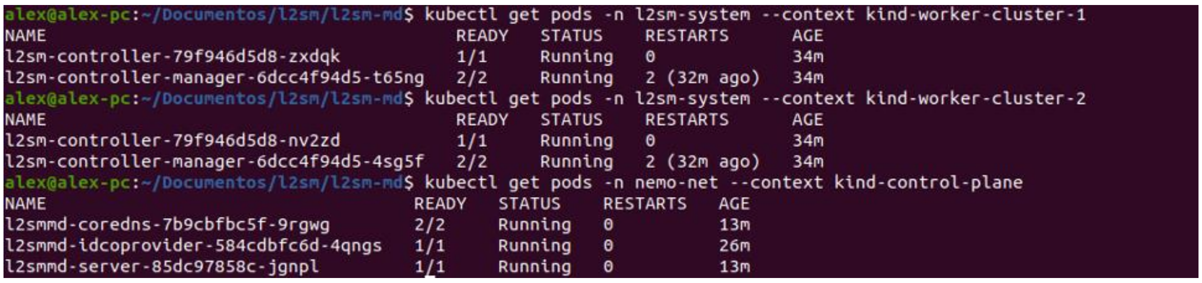
In order to demonstrate the advances of the L2S-M connectivity adaptor in the NEMO project, this new PoC showcases the automatic setup of the adaptor in a NEMO inter-domain cluster environment in order to enable the deployment of virtualized workloads within the infrastructure of the project. Since the internal components that enabled this connectivity were previously shown in D.2.2, this new PoC will focus on showcasing the ability of the new L2S-M-MD module (described in Section 4) to properly configure new k8s clusters to the NEMO project, ensuring their ability to create inter-domain virtual networks that enable the connectivity of network functions in multiple K8s clusters.

Particularly, this setup consists of three separate K8s clusters: one controller cluster and two worker clusters, all following the guidelines of the NEMO project hierarchy and its characteristics. In this regard, for the K8s control plane cluster, the L2SM-MD-server, the IDCO provider and multidomain DNS components were installed, while the L2S-M operator and its components were installed in the L2S-M namespace of the K8s worker clusters. This setup can be seen in Figure 20: K8s clusters of this PoC.



```
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kind get clusters
control-plane
worker-cluster-1
worker-cluster-2
```

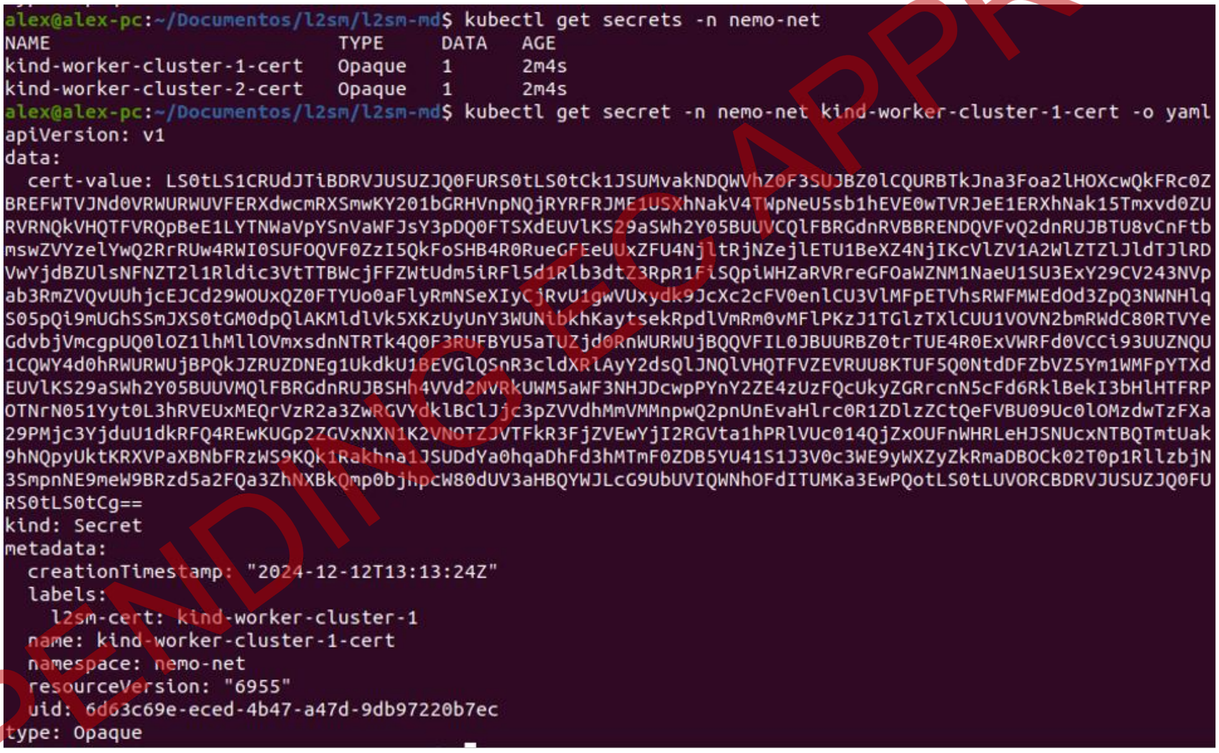
Figure 20 K8s clusters of this PoC



```
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get pods -n l2sm-system --context kind-worker-cluster-1
NAME                                       READY   STATUS    RESTARTS      AGE
l2sm-controller-79f946d5d8-zxdqk           1/1     Running   0             34m
l2sm-controller-manager-6dcc4f94d5-t65ng   2/2     Running   2 (32m ago)   34m
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get pods -n l2sm-system --context kind-worker-cluster-2
NAME                                       READY   STATUS    RESTARTS      AGE
l2sm-controller-79f946d5d8-nv2zd           1/1     Running   0             34m
l2sm-controller-manager-6dcc4f94d5-4sg5f   2/2     Running   2 (32m ago)   34m
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get pods -n nemo-net --context kind-control-plane
NAME                                   READY   STATUS    RESTARTS   AGE
l2smmd-coredns-7b9cbfbc5f-9rgwg        2/2     Running   0          13m
l2smmd-idcoprovider-584cdbfc6d-4qngs   1/1     Running   0          26m
l2smmd-server-85dc97858c-jgnpl         1/1     Running   0          13m
```

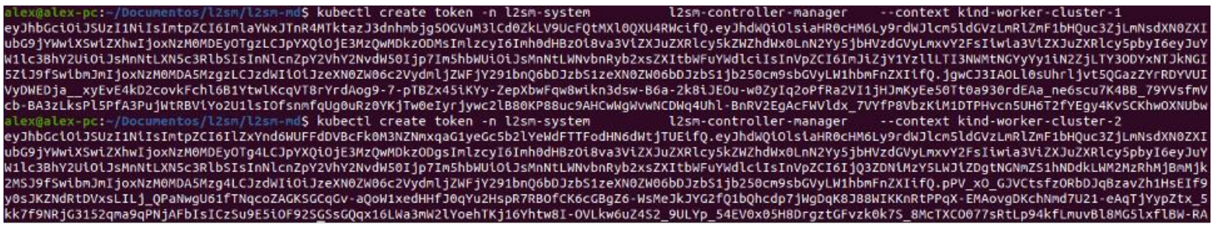
Figure 21 Resources present in the clusters in the PoC

Since we are interested in enabling the management cluster to configure and deploy new components and K8s resources in the worker clusters, it is necessary to generate the public certificates of their respective server APIs and the tokens that enable the management for K8s resources in each cluster, which can be done using the K8s CLI as seen in Figure 22 and Figure 23.



```
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get secrets -n nemo-net
NAME                          TYPE     DATA   AGE
kind-worker-cluster-1-cert    Opaque   1      2m4s
kind-worker-cluster-2-cert    Opaque   1      2m4s
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get secret -n nemo-net kind-worker-cluster-1-cert -o yaml
apiVersion: v1
data:
  cert-value: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUMvakNDQWVhZ0F3SUJBZ0lCQURBTkJna3Foa2lHOXcwQkFRc0Z
BREFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRJME1USXhNakV4TWpNeU5sb1hEVE0wTVRJeE1ERXhNak15Tmxvd0ZU
RVRNQkVHQTFVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBRENDQVFvQ2dnRUJBTU8vCnFtb
mswZVYzelYwQ2RrRUw4RWI0SUFOQVF0ZzI5QkFoSHB4R0RueGFEeUUxZFU4NjltRjNZejlETU1BeXZ4NjIKcVlZV1A2WlZTZlJldTJlRD
VwYjdBZUlsNFNZT2l1Rldic3VtTTBWcjFFZWtUdm5iRFl5d1Rlb3dtZ3RpR1FiSQpiWHZaRVRreGFOaWZNM1NaeU1SU3ExY29CV243NVp
ab3RmZVQvUUhjcEJCd29WOUxQZ0FTYUo0aFlyRmNSeXIyCjRvU1gwVUxydk9JcXc2cFV0enlCU3VlMFpETVhsRWFMWEdOd3ZpQ3NWNHlq
S05pQi9mUGhSSmJXS0tGM0dpQlAKMldlVk5XKzUyUnY3WUNibkhKaytsekRpdlVmRm0vMFlPKzJ1TGlzTXlCUU1VOVN2bmRWdC80RTVYe
GdvbjVmcgpUQ0lOZ1lhMllOVmxsdnNTRTk4Q0F3RUFBYU5aTUZjd0RnWURWUjBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU
1CQWY4d0hRWURWUjBPQkJZRUZDNEg1UkdkU1BEVGlQSnR3cldXRlAyY2dsQlJNQlVHQTFVZEVRUU8KTUF5Q0NtdDFZbVZ5Ym1WMFpYTXd
EUVlKS29aSWh2Y05BUUVMQlFBRGdnRUJBSHh4VVd2NVRkUWM5aWF3NHJDcwpPYnY2ZE4zUzFQcUkyZGRrcnN5cFd6RklBekI3bHlHTFRP
OTNrN051Yyt0L3hRVEUxMEQrVzR2a3ZwRGVYdklBClJjc3pZVVdhMmVMMnpwQ2pnUnEvaHlrc0R1ZDlzZCtQeFVBU09Uc0lOMzdwTzFXa
29PMjc3YjduU1dkRFQ4REwKUGp2ZGVxNXN1K2VNOTZJVTFkR3FjZVEwYjI2RGVta1hPRlVUc014QjZxOUFnWHRLeHJSNUcxNTBQTmtUak
9hNQpyUktKRXVPaXBNbFRzWS9KQk1Rakhna1JSUDdYa0hqaDFd3hMTmF0ZDB5YU41S1J3V0c3WE9yWXZyZkRmaDBOCk02T0p1RllzbjN
3SmpnNE9meW9BRzd5a2FQa3ZhNXBkQmp0bjhpcW80dUV3aHBQYWJLcG9UbUVIQWNhOFdITUKa3EwPQotLS0tLUVORCBDRVJUSUZJQ0FU
RS0tLS0tCg==
kind: Secret
metadata:
  creationTimestamp: "2024-12-12T13:13:24Z"
  labels:
    l2sm-cert: kind-worker-cluster-1
  name: kind-worker-cluster-1-cert
  namespace: nemo-net
  resourceVersion: "6955"
  uid: 6d63c69e-eced-4b47-a47d-9db97220b7ec
type: Opaque
```

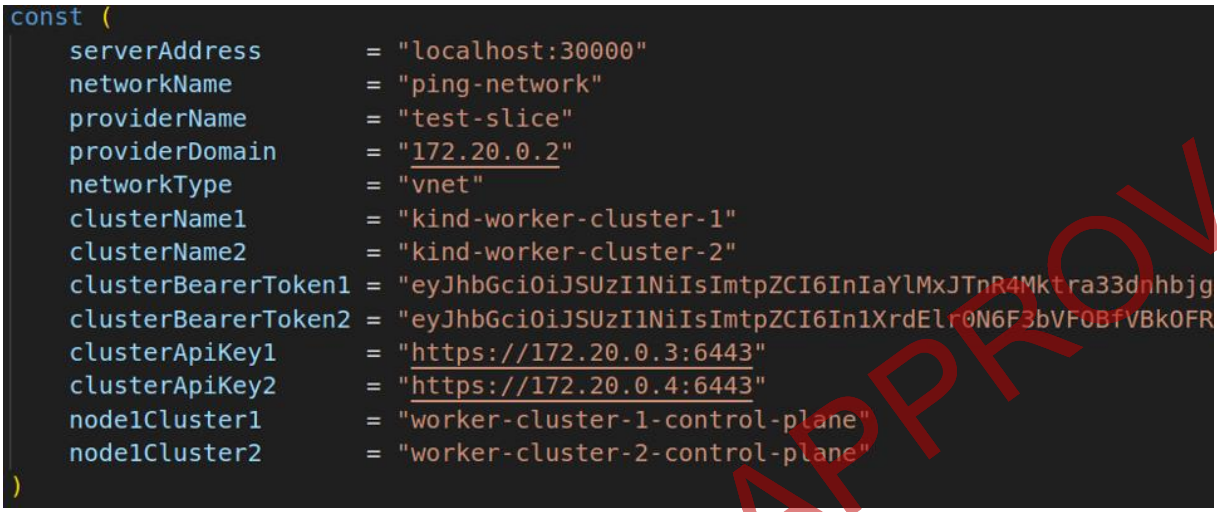
Figure 22 API endpoint generation



```
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl create token -n l2sm-system          l2sm-controller-manager    --context kind-worker-cluster-1
eyJhbGciOiJSUzI1NiIsImtpZCI6ImlaYWxJTnR4MTktazJ3dnhmbjg5OGVuM3lCd0ZkLV9UcFQtMXl0QXU4RWcifQ.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXI
ubG9jYWwiXSwiZXhwIjoxNzM0MDEyOTgzLCJpYXQiOjE3MzQwMDkzODMsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuY
W1lc3BhY2UiOiJsMnNtLXN5c3RlbSIsInNlcnZpY2VhY2NvdW50Ijp7Im5hbWUiOiJsMnNtLWNvbnRyb2xsZXItbWFuYWdlciIsInVpZCI6ImJiZjY1YzllLTI3NWMtNGYyYy1iN2ZjLTY3ODYxNTJkNGI
5ZiJ9fSwibmJmIjoxNzM0MDA5MzgzLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6bDJzbS1zeXN0ZW06bDJzbS1jb250cm9sbGVyLW1hbmFnZXIifQ.jgwCJ3IAOLl0sUhrljvt5QGazZYrRDYVUI
VyDWEDja__xyEvE4kD2covkFchl6B1YtwlKcqVT8rYrdAog9-7-pTBZx45iKYy-ZepXbwFqw8wikn3dsw-B6a-2k8iJEOu-w0ZyIq2oPfRa2VI1jHJmKyEe50Tt0a930rdEAa_ne6scu7K4BB_79YVsfmV
cb-BA3zLksPl5PfA3PujWtRBViYo2U1lsIOfsnmfqUg0uRz0YKjTw0eIyrjywc2lB80KP88uc9AHCwWgWvwNCDWq4Uhl-BnRV2EgAcFWVldx_7VYfP8VbzKiM1DTPHvcn5UH6T2fYEgy4KvSCKhwOXNUbw
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl create token -n l2sm-system          l2sm-controller-manager    --context kind-worker-cluster-2
eyJhbGciOiJSUzI1NiIsImtpZCI6IlZxYnd6WUFFdDVBcFk0M3NZNmxqaG1yeGc5b2lYeWdFTTFodHN6dWtjTUEifQ.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXI
ubG9jYWwiXSwiZXhwIjoxNzM0MDEyOTg4LCJpYXQiOjE3MzQwMDkzODgsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuY
W1lc3BhY2UiOiJsMnNtLXN5c3RlbSIsInNlcnZpY2VhY2NvdW50Ijp7Im5hbWUiOiJsMnNtLWNvbnRyb2xsZXItbWFuYWdlciIsInVpZCI6IjQ3ZDNiMzY5LWJiZDgtNGNmZS1hNDdkLWM2MzRhMjBmMjk
2MSJ9fSwibmJmIjoxNzM0MDA5Mzg4LCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6bDJzbS1zeXN0ZW06bDJzbS1jb250cm9sbGVyLW1hbmFnZXIifQ.pPV_xO_GJVCtsfzORbDJqBzavZh1HsEIf9
y0sJKZNdRtDVxsLILj_QPaNwgU61fTNqcoZAGKSGCqGv-aQoW1xedHHfJ0qYu2HspR7RBOfCK6cGBgZ6-WsMeJkJYG2fQ1bQhcdp7jWgDqK8J88WIKKnRtPPqX-EMAovgDKchNmd7U21-eAqTjYypZtx_5
kk7f9NRjG3152qma9qPNjAFbIsICzSu9E5iOF92SGSsGQqx16LWa3mW2lYoehTKj16Yhtw8I-OVLkw6uZ4S2_9ULYp_54EV0x05H8DrgztGFvzk0k7S_8McTXCO077sRtLp94kfLmuvBl8MG5lxflBW-RA
```

Figure 23 K8s cluster token generation

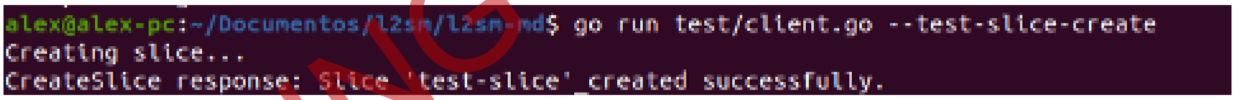
Once all the elements have been created to enable the management of the K8s resources in the clusters that are going to be managed by the mNCC component, the system administrator can proceed to create a slice (i.e., inter cluster overlay that will logically connect both clusters) between both clusters. To do so, it is necessary to introduce the API endpoints and the generated tokens to perform a gRPC call, as it can be seen in Figure 24:



```
const (
    serverAddress       = "localhost:30000"
    networkName         = "ping-network"
    providerName        = "test-slice"
    providerDomain      = "172.20.0.2"
    networkType         = "vnet"
    clusterName1        = "kind-worker-cluster-1"
    clusterName2        = "kind-worker-cluster-2"
    clusterBearerToken1 = "eyJhbGciOiJSUzI1NiIsImtpZCI6InIaYlMxJTnR4Mktra33dnhbjg
    clusterBearerToken2 = "eyJhbGciOiJSUzI1NiIsImtpZCI6In1XrdElrON6F3bVFOBfVBkOFR
    clusterApiKey1      = "https://172.20.0.3:6443"
    clusterApiKey2      = "https://172.20.0.4:6443"
    node1Cluster1       = "worker-cluster-1-control-plane"
    node1Cluster2       = "worker-cluster-2-control-plane"
)
```

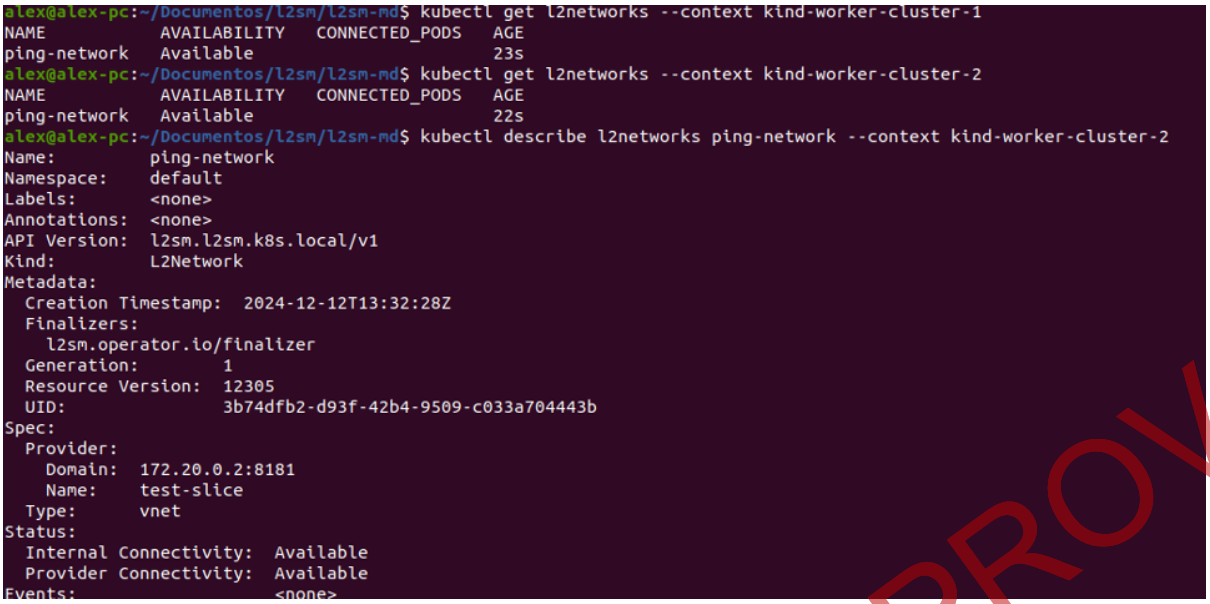
Figure 24 Configuration file used to create a new slice with the L2S-M-md component

With these values, the administrator can proceed with the creation using the available API in the manager cluster, which will install both NEDs and the overlay network (slice) in the managed K8s clusters as their own resources, as seen in Figure 25:



```
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ go run test/client.go --test-slice-create
Creating slice...
CreateSlice response: Slice 'test-slice' created successfully.
```

Figure 25 CLI used to create the slice between the clusters



```
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get l2networks --context kind-worker-cluster-1
NAME           AVAILABILITY   CONNECTED_PODS   AGE
ping-network   Available                       23s
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl get l2networks --context kind-worker-cluster-2
NAME           AVAILABILITY   CONNECTED_PODS   AGE
ping-network   Available                       22s
alex@alex-pc:~/Documentos/l2sm/l2sm-md$ kubectl describe l2networks ping-network --context kind-worker-cluster-2
Name:         ping-network
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  l2sm.l2sm.k8s.local/v1
Kind:         L2Network
Metadata:
  Creation Timestamp:  2024-12-12T13:32:28Z
  Finalizers:
    l2sm.operator.io/finalizer
  Generation:        1
  Resource Version:  12305
  UID:               3b74dfb2-d93f-42b4-9509-c033a704443b
Spec:
  Provider:
    Domain:  172.20.0.2:8181
    Name:    test-slice
  Type:      vnet
Status:
  Internal Connectivity:  Available
  Provider Connectivity:  Available
Events:                   <none>
```

Figure 26 L2S-M resources successfully installed in both clusters

After this installation, the mNCC can interconnect NEMO workloads in these managed K8s clusters, which showcases how the new L2SM-MD component can greatly assist the procedure of setting up the inter-cluster communication between K8s clusters belonging to a NEMO infrastructure.

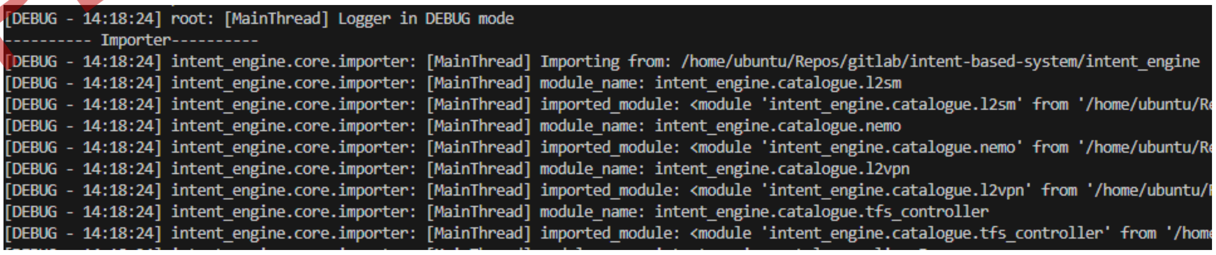
6.3.2 Intent-Based System

In order to showcase the advancements in the Intent-Based System, two different cases will be described. The purpose of the first one will be to describe the full intent lifecycle that the IBS follows to translate an Intent to the specific network technology. In this case it will be using the TFS adaptor. The second one, will show the integration with the L2SM component and making use of the cloud continuum intents for intercluster connectivity.

6.3.2.1 Intent lifecycle. TFS L2VPN adaptor.

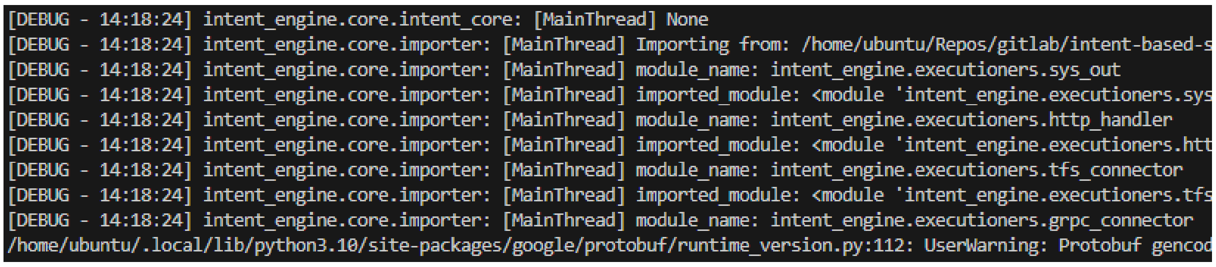
As explained in previous sections and deliverables (4.2.1, [1]), the Intent lifecycle is divided mainly in classification, translation and execution. The next logs snippets are part of an execution of the IBS managing the request of a new L2VPN creation in the intent format.

At the very beginning, the IBS detects all the intent libraries installed and loads all of them to later be able to translate to the different underlying technologies. This phase is divided in two parts: the one importing libraries Figure 27, and the one importing the executioners Figure 28.



```
[DEBUG - 14:18:24] root: [MainThread] Logger in DEBUG mode
---------- Importer----------
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] Importing from: /home/ubuntu/Repos/gitlab/intent-based-system/intent_engine
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.catalogue.l2sm
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.catalogue.l2sm' from '/home/ubuntu/Re
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.catalogue.nemo
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.catalogue.nemo' from '/home/ubuntu/Re
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.catalogue.l2vpn
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.catalogue.l2vpn' from '/home/ubuntu/
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.catalogue.tfs_controller
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.catalogue.tfs_controller' from '/home
```

Figure 27 IBS importing libraries

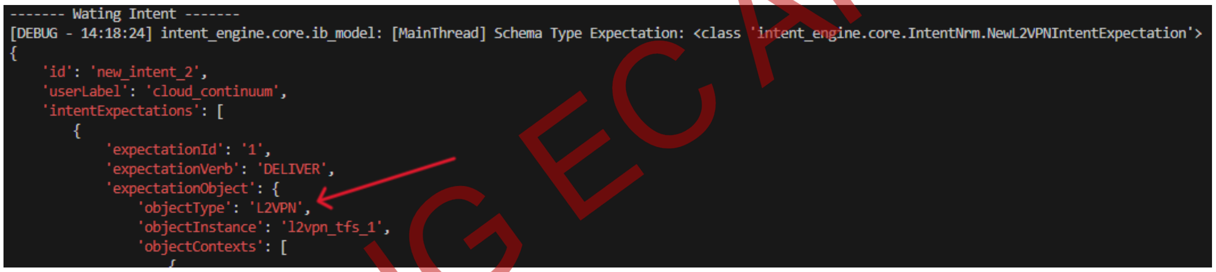


```
[DEBUG - 14:18:24] intent_engine.core.intent_core: [MainThread] None
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] Importing from: /home/ubuntu/Repos/gitlab/intent-based-s
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.executioners.sys_out
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.executioners.sys
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.executioners.http_handler
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.executioners.htt
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.executioners.tfs_connector
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] imported_module: <module 'intent_engine.executioners.tfs
[DEBUG - 14:18:24] intent_engine.core.importer: [MainThread] module_name: intent_engine.executioners.grpc_connector
/home/ubuntu/.local/lib/python3.10/site-packages/google/protobuf/runtime_version.py:112: UserWarning: Protobuf gencod
```

Figure 28 IBS importing executioners

All the executioners depicted in the previous figure are later available for the different libraries in case they are required. There is a specific executioner created for debugging purposes called "sys_out", used to show the output of the intent translation. Any of them can be used at the same time, so at the end, the one handling TFS and the one handling the system out print will be activated. The fact that this is possible is showcasing also the flexibility of the architecture and the easy integration with different executioners and libraries depending on the needs of the user.

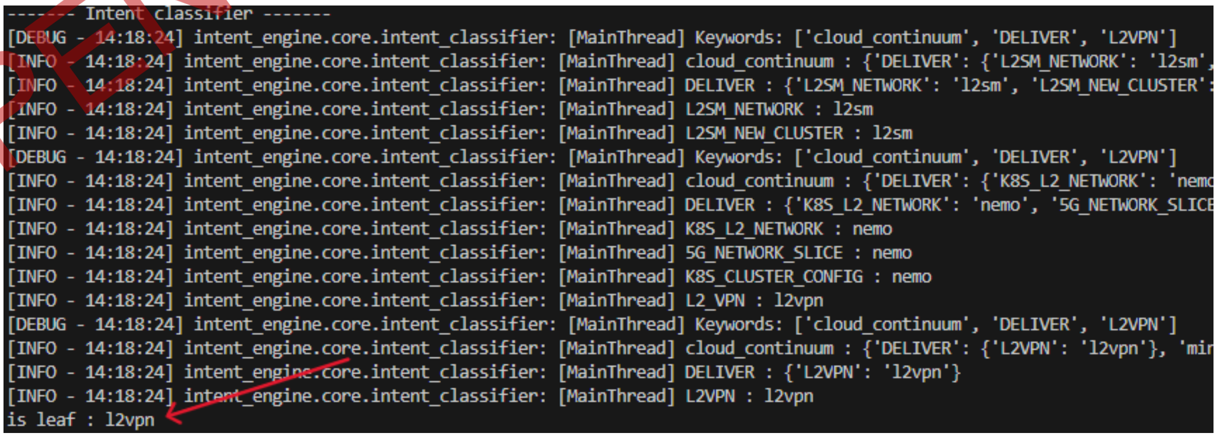
The next screenshot shows the intent incoming and having a first format check. In Figure 29, an arrow is showing the objectType of the intent, in this case "L2VPN". It is important to remark this because the intent will be understood as a layer two VPN but not yet as a TFS L2VPN.



```
------- Wating Intent -------
[DEBUG - 14:18:24] intent_engine.core.ib_model: [MainThread] Schema Type Expectation: <class 'intent_engine.core.IntentNrm.NewL2VPNIntentExpectation'>
{
    'id': 'new_intent_2',
    'userLabel': 'cloud_continuum',
    'intentExpectations': [
        {
            'expectationId': '1',
            'expectationVerb': 'DELIVER',
            'expectationObject': {
                'objectType': 'L2VPN',
                'objectInstance': 'l2vpn_tfs_1',
                'objectContexts': [
```

Figure 29 IBS input intent

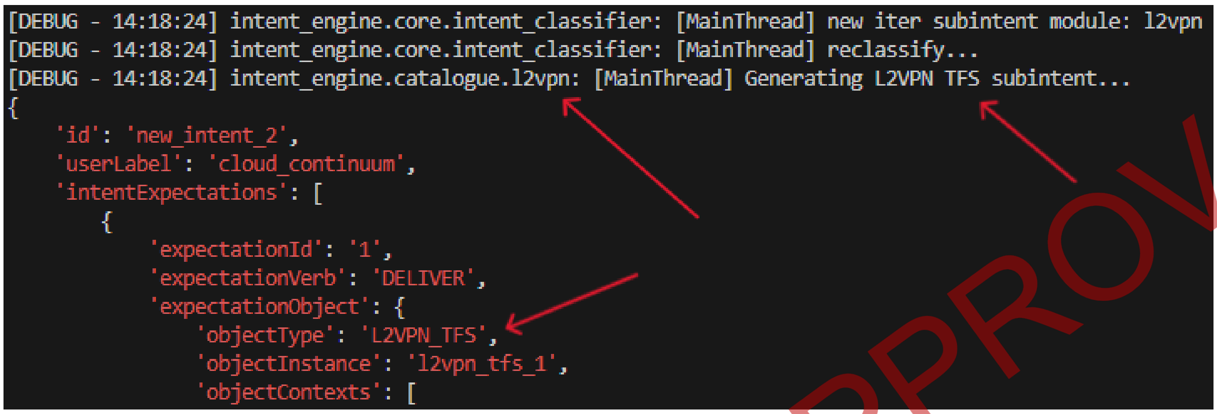
In the Figure 30, it is showed how the comparison between the different libraries is traversed, and at the end, how the IBS is able to classify the intent as "L2VPN". This is due to the fact that, in the classification tree used, when the system reaches a leave, it means it fully matches all the keywords for the library.



```
------- Intent classifier -------
[DEBUG - 14:18:24] intent_engine.core.intent_classifier: [MainThread] Keywords: ['cloud_continuum', 'DELIVER', 'L2VPN']
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] cloud_continuum : {'DELIVER': {'L2SM_NETWORK': 'l2sm',
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] DELIVER : {'L2SM_NETWORK': 'l2sm', 'L2SM_NEW_CLUSTER':
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] L2SM_NETWORK : l2sm
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] L2SM_NEW_CLUSTER : l2sm
[DEBUG - 14:18:24] intent_engine.core.intent_classifier: [MainThread] Keywords: ['cloud_continuum', 'DELIVER', 'L2VPN']
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] cloud_continuum : {'DELIVER': {'K8S_L2_NETWORK': 'nemo
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] DELIVER : {'K8S_L2_NETWORK': 'nemo', '5G_NETWORK_SLICE
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] K8S_L2_NETWORK : nemo
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] 5G_NETWORK_SLICE : nemo
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] K8S_CLUSTER_CONFIG : nemo
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] L2_VPN : l2vpn
[DEBUG - 14:18:24] intent_engine.core.intent_classifier: [MainThread] Keywords: ['cloud_continuum', 'DELIVER', 'L2VPN']
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] cloud_continuum : {'DELIVER': {'L2VPN': 'l2vpn'}, 'mir
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] DELIVER : {'L2VPN': 'l2vpn'}
[INFO - 14:18:24] intent_engine.core.intent_classifier: [MainThread] L2VPN : l2vpn
is leaf : l2vpn
```

Figure 30 IBS first classification

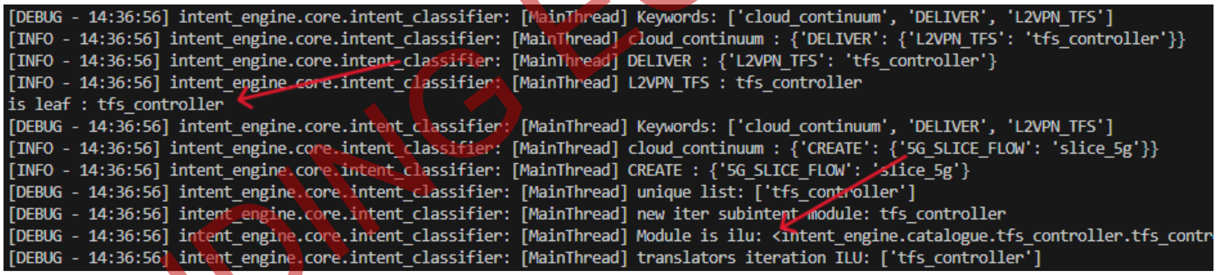
The shows the IBS passing the intent to the L2VPN library. As this library doesn't generate ILUs (Intent Logic Units), the library reclassifies the intent (now called sub-intent), into a less abstracted library the TFS controller one. This step is important to abstract less abstracted libraries from higher ones, this means that the TFS controller library has no need to be aware of more abstracted L2VPN request, and the L2VPN must be aware of the specific technology library but not of the full translation to the technology. The step of reclassifying the intent is done until a library capable of generating an ILU is executed.



```
[DEBUG - 14:18:24] intent_engine.core.intent_classifier: [MainThread] new iter subintent module: l2vpn
[DEBUG - 14:18:24] intent_engine.core.intent_classifier: [MainThread] reclassify...
[DEBUG - 14:18:24] intent_engine.catalogue.l2vpn: [MainThread] Generating L2VPN TFS subintent...
{
    'id': 'new_intent_2',
    'userLabel': 'cloud_continuum',
    'intentExpectations': [
        {
            'expectationId': '1',
            'expectationVerb': 'DELIVER',
            'expectationObject': {
                'objectType': 'L2VPN_TFS',
                'objectInstance': 'l2vpn_tfs_1',
                'objectContexts': [
```

Figure 31: IBS subintent generation.

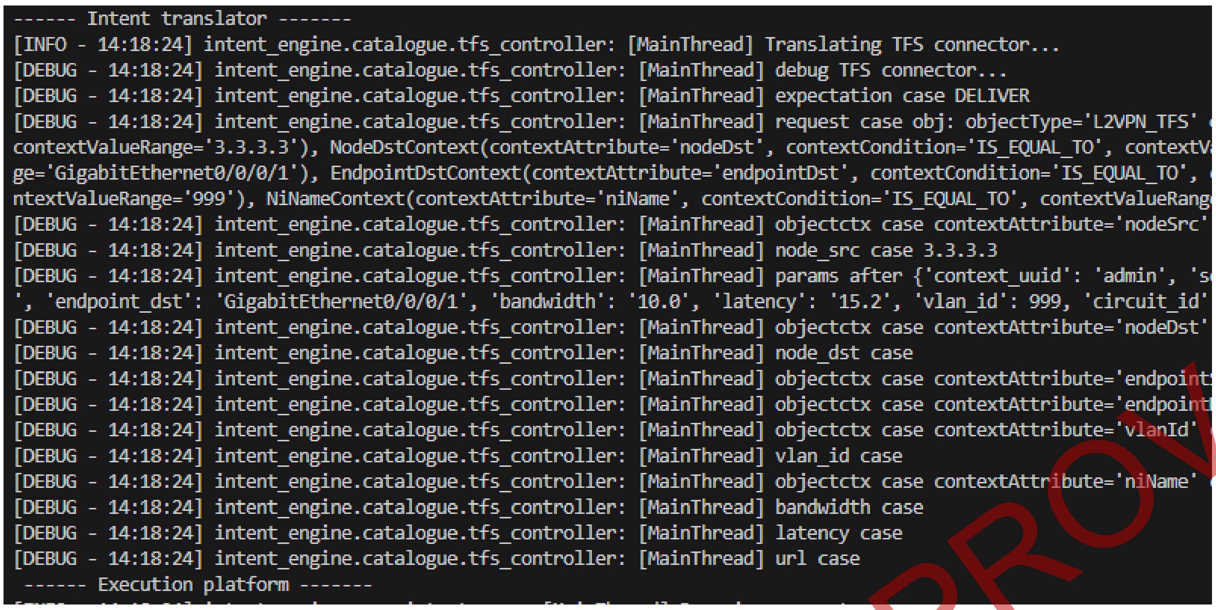
Also, in the reclassification step, the library can make adjustments to the intent so that the following library is able to detect the intent as own. As shown in Figure 31 the "*objectType*" of the intent has changed. Now the sub-intent is passed through the classification again and detected as "*tfs_controller*" intent.



```
[DEBUG - 14:36:56] intent_engine.core.intent_classifier: [MainThread] Keywords: ['cloud_continuum', 'DELIVER', 'L2VPN_TFS']
[INFO - 14:36:56] intent_engine.core.intent_classifier: [MainThread] cloud_continuum : {'DELIVER': {'L2VPN_TFS': 'tfs_controller'}}
[INFO - 14:36:56] intent_engine.core.intent_classifier: [MainThread] DELIVER : {'L2VPN_TFS': 'tfs_controller'}
[INFO - 14:36:56] intent_engine.core.intent_classifier: [MainThread] L2VPN_TFS : tfs_controller
is leaf : tfs_controller
[DEBUG - 14:36:56] intent_engine.core.intent_classifier: [MainThread] Keywords: ['cloud_continuum', 'DELIVER', 'L2VPN_TFS']
[INFO - 14:36:56] intent_engine.core.intent_classifier: [MainThread] cloud_continuum : {'CREATE': {'5G_SLICE_FLOW': 'slice_5g'}}
[INFO - 14:36:56] intent_engine.core.intent_classifier: [MainThread] CREATE : {'5G_SLICE_FLOW': 'slice_5g'}
[DEBUG - 14:36:56] intent_engine.core.intent_classifier: [MainThread] unique list: ['tfs_controller']
[DEBUG - 14:36:56] intent_engine.core.intent_classifier: [MainThread] new iter subintent module: tfs_controller
[DEBUG - 14:36:56] intent_engine.core.intent_classifier: [MainThread] Module is ilu: <intent_engine.catalogue.tfs_controller.tfs_contr
[DEBUG - 14:36:56] intent_engine.core.intent_classifier: [MainThread] translators iteration ILU: ['tfs_controller']
```

Figure 32 IBS second classification

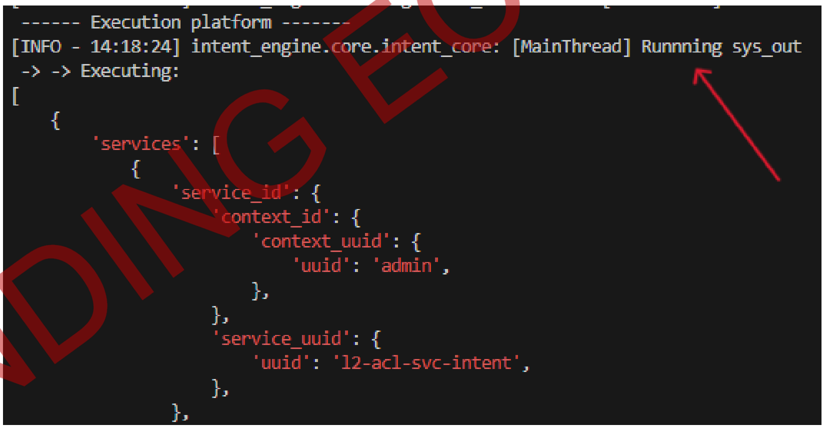
In the Figure 32, it is also shown how now the intent is classified as ILU. From this point, it starts the translation phase. In the Figure 33, it is shown the parsing of the intent, and how it traverses all the contexts, targets and expectations.



```
------ Intent translator -------
[INFO - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] Translating TFS connector...
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] debug TFS connector...
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] expectation case DELIVER
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] request case obj: objectType='L2VPN_TFS'
contextValueRange='3.3.3.3'), NodeDstContext(contextAttribute='nodeDst', contextCondition='IS_EQUAL_TO', contextV
ge='GigabitEthernet0/0/0/1'), EndpointDstContext(contextAttribute='endpointDst', contextCondition='IS_EQUAL_TO',
ntextValueRange='999'), NiNameContext(contextAttribute='niName', contextCondition='IS_EQUAL_TO', contextValueRang
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] objectctx case contextAttribute='nodeSrc'
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] node_src case 3.3.3.3
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] params after {'context_uuid': 'admin', 's
', 'endpoint_dst': 'GigabitEthernet0/0/0/1', 'bandwidth': '10.0', 'latency': '15.2', 'vlan_id': 999, 'circuit_id'
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] objectctx case contextAttribute='nodeDst'
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] node_dst case
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] objectctx case contextAttribute='endpoint
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] objectctx case contextAttribute='endpoint
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] objectctx case contextAttribute='vlanId'
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] vlan_id case
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] objectctx case contextAttribute='niName'
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] bandwidth case
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] latency case
[DEBUG - 14:18:24] intent_engine.catalogue.tfs_controller: [MainThread] url case
 ------ Execution platform -------
```

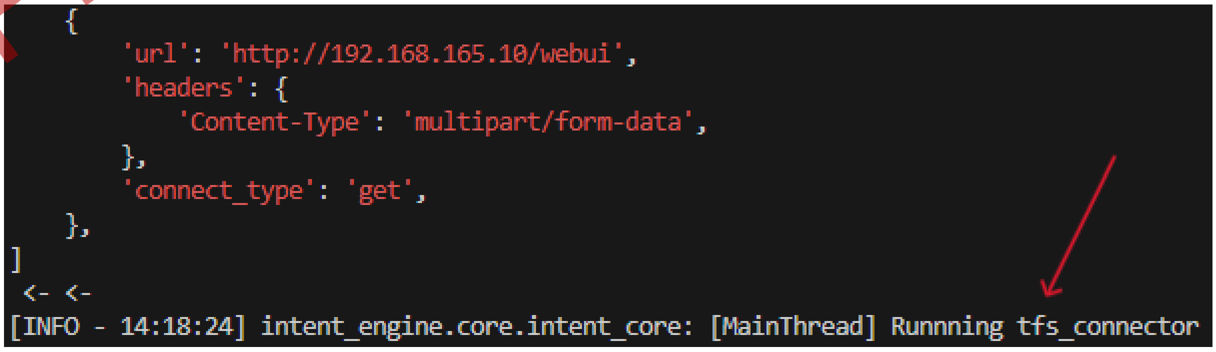
Figure 33 IBS tfs_l2vpn translation

Finally, when the intent is translated, the library in charge of the translation calls the executioners needed to execute the intent. As explained earlier this case shows a first one calling "*sys_out*" () and a second one to the real "*tfs_connector*" ().



```
 ------ Execution platform -------
[INFO - 14:18:24] intent_engine.core.intent_core: [MainThread] Runnning sys_out
 -> -> Executing:
[
    {
        'services': [
            {
                'service_id': {
                    'context_id': {
                        'context_uuid': {
                            'uuid': 'admin',
                        },
                    },
                    'service_uuid': {
                        'uuid': 'l2-acl-svc-intent',
                    },
                },
```

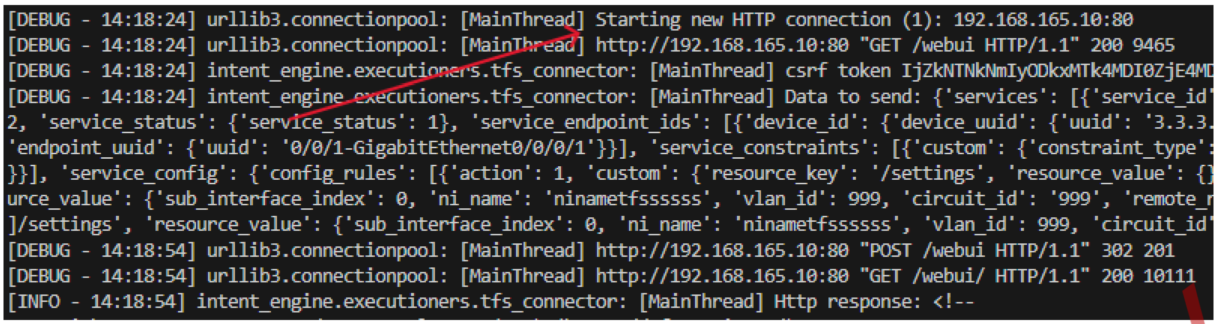
Figure 34 IBS "sys_out" executioner



```
    {
        'url': 'http://192.168.165.10/webui',
        'headers': {
            'Content-Type': 'multipart/form-data',
        },
        'connect_type': 'get',
    },
]
 <- <-
[INFO - 14:18:24] intent_engine.core.intent_core: [MainThread] Runnning tfs_connector
```

Figure 35 IBS "tfs_connector" executioner

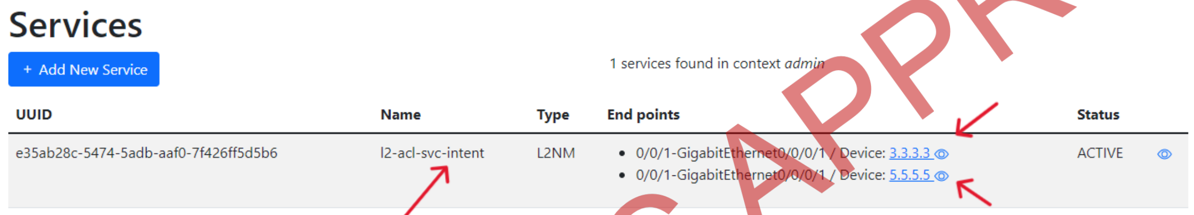
Now, the executioner starts the proper protocol to create a connection to TFS north bound interface and makes a request for a new L2VPN connection.



```
[DEBUG - 14:18:24] urllib3.connectionpool: [MainThread] Starting new HTTP connection (1): 192.168.165.10:80
[DEBUG - 14:18:24] urllib3.connectionpool: [MainThread] http://192.168.165.10:80 "GET /webui HTTP/1.1" 200 9465
[DEBUG - 14:18:24] intent_engine.executioners.tfs_connector: [MainThread] csrf token IjZkNTNkNmIyODkxMTk4MDI0ZjE4MD
[DEBUG - 14:18:24] intent_engine.executioners.tfs_connector: [MainThread] Data to send: {'services': [{'service_id'
2, 'service_status': {'service_status': 1}, 'service_endpoint_ids': [{'device_id': {'device_uuid': {'uuid': '3.3.3.
'endpoint_uuid': {'uuid': '0/0/1-GigabitEthernet0/0/0/1'}}], 'service_constraints': [{'custom': {'constraint_type':
}}], 'service_config': {'config_rules': [{'action': 1, 'custom': {'resource_key': '/settings', 'resource_value': {}
urce_value': {'sub_interface_index': 0, 'ni_name': 'ninametfssssss', 'vlan_id': 999, 'circuit_id': '999', 'remote_r
]/settings', 'resource_value': {'sub_interface_index': 0, 'ni_name': 'ninametfssssss', 'vlan_id': 999, 'circuit_id'
[DEBUG - 14:18:54] urllib3.connectionpool: [MainThread] http://192.168.165.10:80 "POST /webui HTTP/1.1" 302 201
[DEBUG - 14:18:54] urllib3.connectionpool: [MainThread] http://192.168.165.10:80 "GET /webui/ HTTP/1.1" 200 10111
[INFO - 14:18:54] intent_engine.executioners.tfs_connector: [MainThread] Http response: <!--
```

Figure 36 IBS http request to TFS

As a result of the process, a VPN service in TFS controller is created. In Figure 37, notice how the service is active on the nodes requested, and in Figure 38, are shown all the details translated from the *intentContext* and the *intentTargets* specified in the input intent.

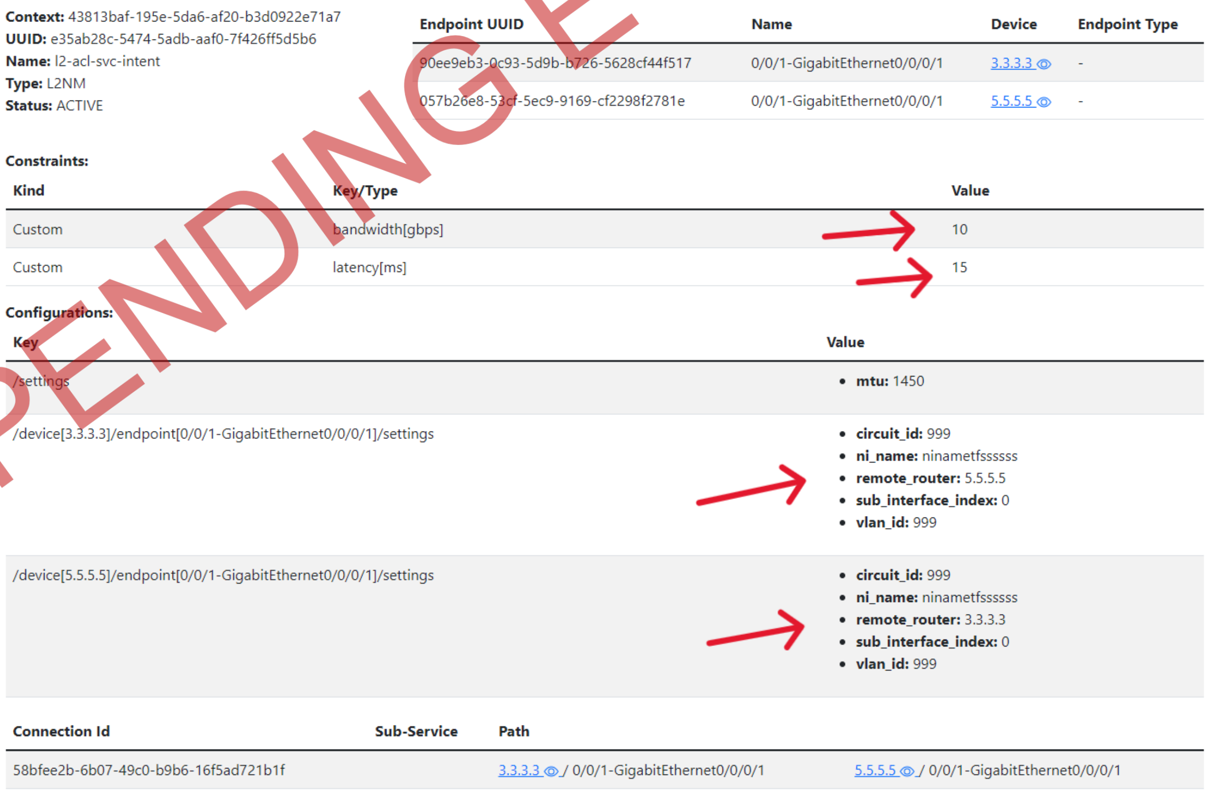


Services

1 services found in context *admin*

UUID	Name	Type	End points	Status
e35ab28c-5474-5adb-aaf0-7f426ff5d5b6	l2-acl-svc-intent	L2NM	• 0/0/1-GigabitEthernet0/0/0/1 / Device: 3.3.3.3 • 0/0/1-GigabitEthernet0/0/0/1 / Device: 5.5.5.5	ACTIVE

Figure 37 TFS webui L2VPN service status



Context: 43813baf-195e-5da6-af20-b3d0922e71a7
UUID: e35ab28c-5474-5adb-aaf0-7f426ff5d5b6
Name: l2-acl-svc-intent
Type: L2NM
Status: ACTIVE

Endpoint UUID	Name	Device	Endpoint Type
90ee9eb3-0c93-5d9b-b726-5628cf44f517	0/0/1-GigabitEthernet0/0/0/1	3.3.3.3	-
057b26e8-53cf-5ec9-9169-cf2298f2781e	0/0/1-GigabitEthernet0/0/0/1	5.5.5.5	-

Constraints:

Kind	Key/Type	Value
Custom	bandwidth[gbps]	10
Custom	latency[ms]	15

Configurations:

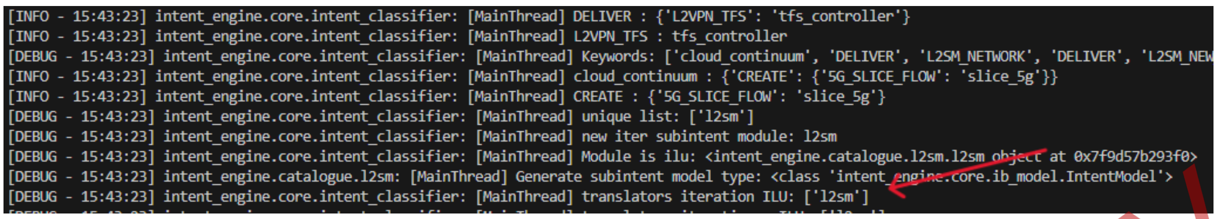
Key	Value
/settings	• **mtu:** 1450
/device[3.3.3.3]/endpoint[0/0/1-GigabitEthernet0/0/0/1]/settings	• **circuit_id:** 999 • **ni_name:** ninametfssssss • **remote_router:** 5.5.5.5 • **sub_interface_index:** 0 • **vlan_id:** 999
/device[5.5.5.5]/endpoint[0/0/1-GigabitEthernet0/0/0/1]/settings	• **circuit_id:** 999 • **ni_name:** ninametfssssss • **remote_router:** 3.3.3.3 • **sub_interface_index:** 0 • **vlan_id:** 999

Connection Id	Sub-Service	Path	
58bfee2b-6b07-49c0-b9b6-16f5ad721b1f		3.3.3.3 / 0/0/1-GigabitEthernet0/0/0/1	5.5.5.5 / 0/0/1-GigabitEthernet0/0/0/1

Figure 38 TFS webui L2VPN service details

6.3.2.2 Cloud continuum Intent. L2SM MD connectivity

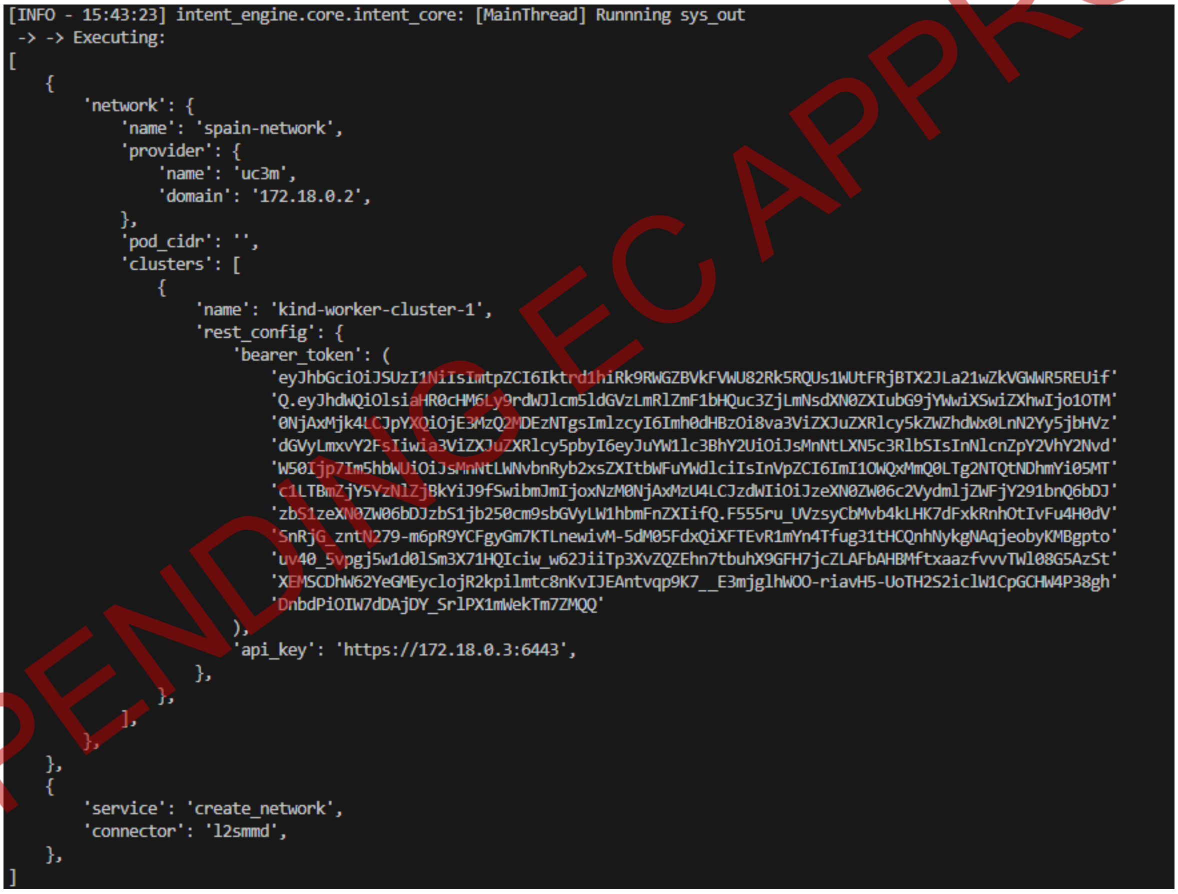
This section will show how an L2SM network is deployed through an intent coming to the IBS (full intent on annex 9.2). In the Figure 39, it is shown how the classification step detects the L2SM intent and calls the L2SM library.



```
[INFO - 15:43:23] intent_engine.core.intent_classifier: [MainThread] DELIVER : {'L2VPN_TFS': 'tfs_controller'}
[INFO - 15:43:23] intent_engine.core.intent_classifier: [MainThread] L2VPN_TFS : tfs_controller
[DEBUG - 15:43:23] intent_engine.core.intent_classifier: [MainThread] Keywords: ['cloud_continuum', 'DELIVER', 'L2SM_NETWORK', 'DELIVER', 'L2SM_NEW
[INFO - 15:43:23] intent_engine.core.intent_classifier: [MainThread] cloud_continuum : {'CREATE': {'5G_SLICE_FLOW': 'slice_5g'}}
[INFO - 15:43:23] intent_engine.core.intent_classifier: [MainThread] CREATE : {'5G_SLICE_FLOW': 'slice_5g'}
[DEBUG - 15:43:23] intent_engine.core.intent_classifier: [MainThread] unique list: ['l2sm']
[DEBUG - 15:43:23] intent_engine.core.intent_classifier: [MainThread] new iter subintent module: l2sm
[DEBUG - 15:43:23] intent_engine.core.intent_classifier: [MainThread] Module is ilu: <intent_engine.catalogue.l2sm.l2sm object at 0x7f9d57b293f0>
[DEBUG - 15:43:23] intent_engine.catalogue.l2sm: [MainThread] Generate subintent model type: <class 'intent_engine.core.ib_model.IntentModel'>
[DEBUG - 15:43:23] intent_engine.core.intent_classifier: [MainThread] translators iteration ILU: ['l2sm']
```

Figure 39: Intent classification. L2SM case.

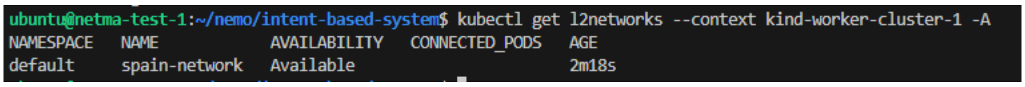
From this point, the L2SM library is in charge of translating the intent to the corresponding interface that L2SM expects. The translation is depicted in next Figure 40.



```
[INFO - 15:43:23] intent_engine.core.intent_core: [MainThread] Runnning sys_out
 -> -> Executing:
[
    {
        'network': {
            'name': 'spain-network',
            'provider': {
                'name': 'uc3m',
                'domain': '172.18.0.2',
            },
            'pod_cidr': '',
            'clusters': [
                {
                    'name': 'kind-worker-cluster-1',
                    'rest_config': {
                        'bearer_token': (
                            'eyJhbGciOiJSUzI1NiIsImtpZCI6Iktrd1hiRk9RWGZBVkFVWU82Rk5RQUs1WUtFRjBTX2JLa21wZkVGWWR5REUif'
                            'Q.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiXSwiZXhwIjo1OTM'
                            '0NjAxMjk4LCJpYXQiOjE3MzQ2MDEzNTgsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVz'
                            'dGVyLmxvY2FsIiwia3ViZXJuZXRlcy5pbyI6eyJuYW1lc3BhY2UiOiJsMnNtLXN5c3RlbSIsInNlcnZpY2VhY2Nvd'
                            'W50Ijp7Im5hbWUiOiJsMnNtLWNvbnRyb2xsZXItbWFuYWdlciIsInVpZCI6ImI1OWQxMmQ0LTg2NTQtNDhmYi05MT'
                            'c1LTBmZjY5YzNlZjBkYiJ9fSwibmJmIjoxNzM0NjAxMzU4LCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6bDJ'
                            'zbS1zeXN0ZW06bDJzbS1jb250cm9sbGVyLW1hbmFnZXIifQ.F555ru_UVzsyCbMvb4kLHK7dFxkRnhOtIvFu4H0dV'
                            'SnRjG_zntN279-m6pR9YCFgyGm7KTLnewivM-5dM05FdxQiXFTEvR1mYn4Tfug31tHCQnhNykgNAqjeobyKMBgpto'
                            'uv40_5vpgj5w1d0lSm3X71HQIciw_w62JiiTp3XvZQZEhn7tbuhX9GFH7jcZLAFbAHBMftxaazfvvvTWl08G5AzSt'
                            'XEMSCDhW62YeGMEyclojR2kpilmtc8nKvIJEAntvqp9K7__E3mjglhWOO-riavH5-UoTH2S2iclW1CpGCHW4P38gh'
                            'DnbdPiOIW7dDAjDY_SrlPX1mWekTm7ZMQQ'
                        ),
                        'api_key': 'https://172.18.0.3:6443',
                    },
                },
            ],
        },
    },
    {
        'service': 'create_network',
        'connector': 'l2smmd',
    },
]
```

Figure 40: Final L2SM translation.

Finally, once the translated intent is executed in L2SM, it creates a new Kubernetes intercluster network as shown in Figure 41, where a Kubernetes command is executed to verify that everything works as expected.



```
ubuntu@netma-test-1:~/nemo/intent-based-system$ kubectl get l2networks --context kind-worker-cluster-1 -A
NAMESPACE   NAME            AVAILABILITY   CONNECTED_PODS   AGE
default     spain-network   Available                       2m18s
```

Figure 41: L2SM intercluster network active.

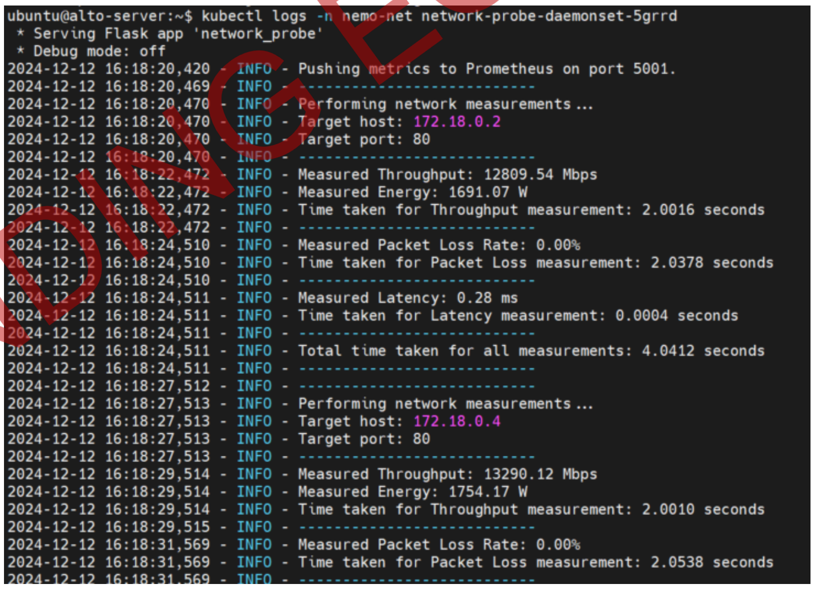
6.3.3 Multi-Cluster monitoring and exposure

In order to perform the PoC of the multi-cluster monitoring and exposure, the mNCC components in charge of monitoring and exposure of the mNCC have been deployed in a scenario with 3 nodes. The mNCC monitoring and exposure process is explained in section 4, so we will not extend with this explanation, but it is worth remembering that the monitoring is based on the communication between pods deployed in each node and then shared with the exposure module through a local Prometheus. Once the exposure module has those metrics, it formats them and indicates to which cluster they belong and at what time they have been generated.

For these tests, we have reduced the monitoring and exposure time to 2 minutes (120 seconds) to show how the RabbitMQ queue is being updated, although in the joint environment these measurements would be hourly (3600s) to avoid saturation in the network and in the metrics queues.

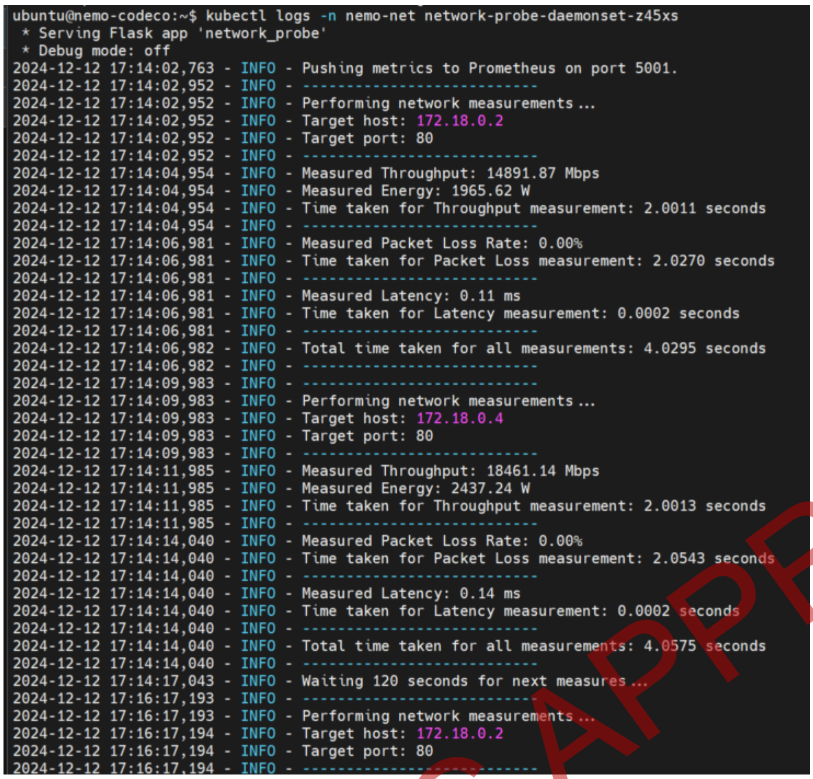
In the process of evaluating the mNCC metrics exposure we must start with the deployment of the probes and the exposure module (Nemex for short) in each cluster, counting on a pod of probes per worker node and a Nemex per cluster. For this proof of concept, we will perform the evaluation on 2 clusters in parallel.

In Figure 42 and Figure 43 we can see a test run of the monitoring pods in which the logs of one of the probes of each cluster are obtained. These logs show the measurements taken periodically and shared with the Nemex through an internal Prometheus (Figure 44 shows the format of these logs).



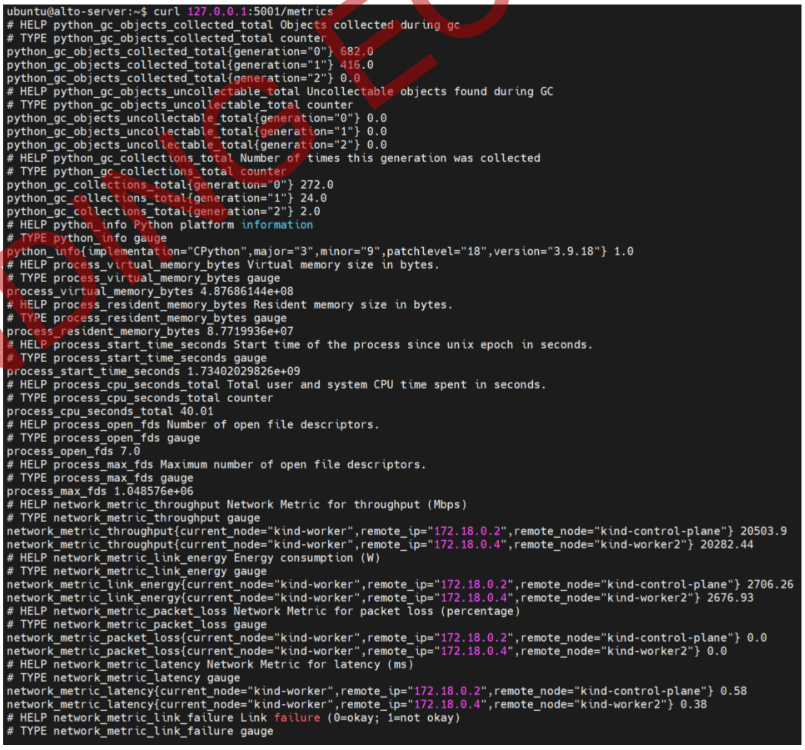
```
ubuntu@alto-server:~$ kubectl logs -n nemo-net network-probe-daemonset-5grrd
 * Serving Flask app 'network_probe'
 * Debug mode: off
2024-12-12 16:18:20,420 - INFO - Pushing metrics to Prometheus on port 5001.
2024-12-12 16:18:20,469 - INFO - ---------------------------
2024-12-12 16:18:20,470 - INFO - Performing network measurements...
2024-12-12 16:18:20,470 - INFO - Target host: 172.18.0.2
2024-12-12 16:18:20,470 - INFO - Target port: 80
2024-12-12 16:18:20,470 - INFO - ---------------------------
2024-12-12 16:18:22,472 - INFO - Measured Throughput: 12809.54 Mbps
2024-12-12 16:18:22,472 - INFO - Measured Energy: 1691.07 W
2024-12-12 16:18:22,472 - INFO - Time taken for Throughput measurement: 2.0016 seconds
2024-12-12 16:18:22,472 - INFO - ---------------------------
2024-12-12 16:18:24,510 - INFO - Measured Packet Loss Rate: 0.00%
2024-12-12 16:18:24,510 - INFO - Time taken for Packet Loss measurement: 2.0378 seconds
2024-12-12 16:18:24,510 - INFO - ---------------------------
2024-12-12 16:18:24,511 - INFO - Measured Latency: 0.28 ms
2024-12-12 16:18:24,511 - INFO - Time taken for Latency measurement: 0.0004 seconds
2024-12-12 16:18:24,511 - INFO - ---------------------------
2024-12-12 16:18:24,511 - INFO - Total time taken for all measurements: 4.0412 seconds
2024-12-12 16:18:24,511 - INFO - ---------------------------
2024-12-12 16:18:27,512 - INFO - ---------------------------
2024-12-12 16:18:27,513 - INFO - Performing network measurements...
2024-12-12 16:18:27,513 - INFO - Target host: 172.18.0.4
2024-12-12 16:18:27,513 - INFO - Target port: 80
2024-12-12 16:18:27,513 - INFO - ---------------------------
2024-12-12 16:18:29,514 - INFO - Measured Throughput: 13290.12 Mbps
2024-12-12 16:18:29,514 - INFO - Measured Energy: 1754.17 W
2024-12-12 16:18:29,514 - INFO - Time taken for Throughput measurement: 2.0010 seconds
2024-12-12 16:18:29,515 - INFO - ---------------------------
2024-12-12 16:18:31,569 - INFO - Measured Packet Loss Rate: 0.00%
2024-12-12 16:18:31,569 - INFO - Time taken for Packet Loss measurement: 2.0538 seconds
2024-12-12 16:18:31,569 - INFO - ---------------------------
```

Figure 42 Logs from the network probes in the first cluster.



```
ubuntu@nemo-codeco:~$ kubectl logs -n nemo-net network-probe-daemonset-z45xs
 * Serving Flask app 'network_probe'
 * Debug mode: off
2024-12-12 17:14:02,763 - INFO - Pushing metrics to Prometheus on port 5001.
2024-12-12 17:14:02,952 - INFO - ----------------------------
2024-12-12 17:14:02,952 - INFO - Performing network measurements...
2024-12-12 17:14:02,952 - INFO - Target host: 172.18.0.2
2024-12-12 17:14:02,952 - INFO - Target port: 80
2024-12-12 17:14:02,952 - INFO - ----------------------------
2024-12-12 17:14:04,954 - INFO - Measured Throughput: 14891.87 Mbps
2024-12-12 17:14:04,954 - INFO - Measured Energy: 1965.62 W
2024-12-12 17:14:04,954 - INFO - Time taken for Throughput measurement: 2.0011 seconds
2024-12-12 17:14:04,954 - INFO - ----------------------------
2024-12-12 17:14:06,981 - INFO - Measured Packet Loss Rate: 0.00%
2024-12-12 17:14:06,981 - INFO - Time taken for Packet Loss measurement: 2.0270 seconds
2024-12-12 17:14:06,981 - INFO - ----------------------------
2024-12-12 17:14:06,981 - INFO - Measured Latency: 0.11 ms
2024-12-12 17:14:06,981 - INFO - Time taken for Latency measurement: 0.0002 seconds
2024-12-12 17:14:06,981 - INFO - ----------------------------
2024-12-12 17:14:06,982 - INFO - Total time taken for all measurements: 4.0295 seconds
2024-12-12 17:14:06,982 - INFO - ----------------------------
2024-12-12 17:14:09,983 - INFO - ----------------------------
2024-12-12 17:14:09,983 - INFO - Performing network measurements...
2024-12-12 17:14:09,983 - INFO - Target host: 172.18.0.4
2024-12-12 17:14:09,983 - INFO - Target port: 80
2024-12-12 17:14:09,983 - INFO - ----------------------------
2024-12-12 17:14:11,985 - INFO - Measured Throughput: 18461.14 Mbps
2024-12-12 17:14:11,985 - INFO - Measured Energy: 2437.24 W
2024-12-12 17:14:11,985 - INFO - Time taken for Throughput measurement: 2.0013 seconds
2024-12-12 17:14:11,985 - INFO - ----------------------------
2024-12-12 17:14:14,040 - INFO - Measured Packet Loss Rate: 0.00%
2024-12-12 17:14:14,040 - INFO - Time taken for Packet Loss measurement: 2.0543 seconds
2024-12-12 17:14:14,040 - INFO - ----------------------------
2024-12-12 17:14:14,040 - INFO - Measured Latency: 0.14 ms
2024-12-12 17:14:14,040 - INFO - Time taken for Latency measurement: 0.0002 seconds
2024-12-12 17:14:14,040 - INFO - ----------------------------
2024-12-12 17:14:14,040 - INFO - Total time taken for all measurements: 4.0575 seconds
2024-12-12 17:14:14,040 - INFO - ----------------------------
2024-12-12 17:14:17,043 - INFO - Waiting 120 seconds for next measures...
2024-12-12 17:16:17,193 - INFO - ----------------------------
2024-12-12 17:16:17,193 - INFO - Performing network measurements...
2024-12-12 17:16:17,194 - INFO - Target host: 172.18.0.2
2024-12-12 17:16:17,194 - INFO - Target port: 80
2024-12-12 17:16:17,194 - INFO - ----------------------------
```

Figure 43 Logs from the network probes in the second cluster.



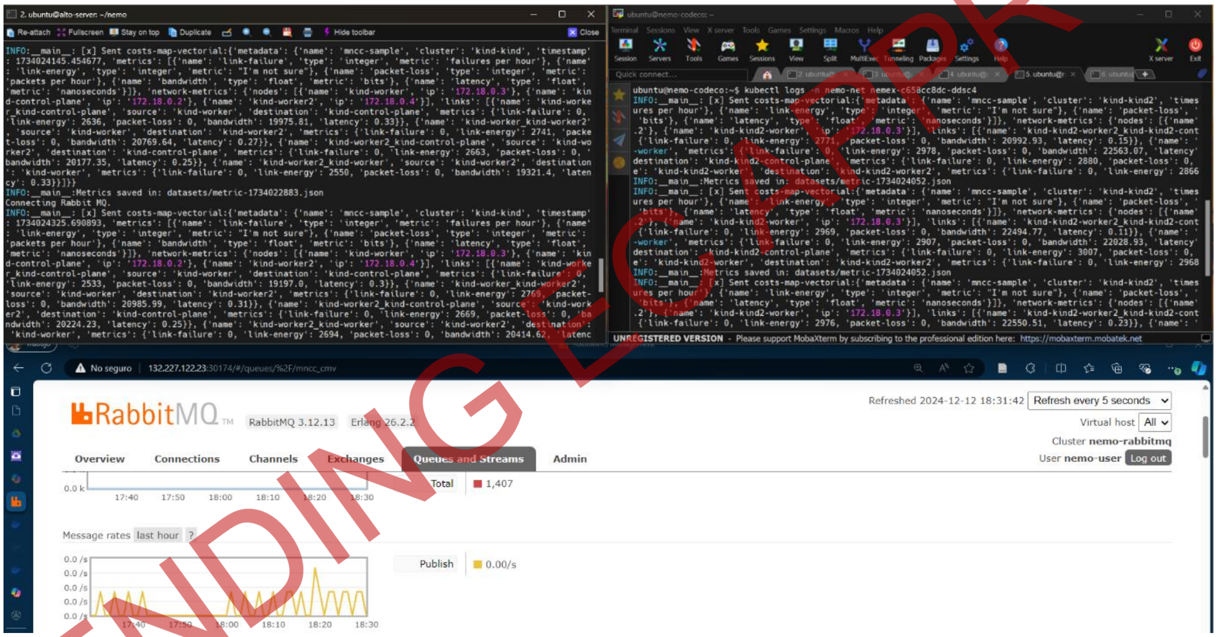
```
ubuntu@alto-server:~$ curl 127.0.0.1:5001/metrics
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 682.0
python_gc_objects_collected_total{generation="1"} 416.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 272.0
python_gc_collections_total{generation="1"} 24.0
python_gc_collections_total{generation="2"} 2.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="9",patchlevel="18",version="3.9.18"} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 4.87686144e+08
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 8.7719936e+07
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.73402029826e+09
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 40.01
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 7.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.048576e+06
# HELP network_metric_throughput Network Metric for throughput (Mbps)
# TYPE network_metric_throughput gauge
network_metric_throughput{current_node="kind-worker",remote_ip="172.18.0.2",remote_node="kind-control-plane"} 20503.9
network_metric_throughput{current_node="kind-worker",remote_ip="172.18.0.4",remote_node="kind-worker2"} 20282.44
# HELP network_metric_link_energy Energy consumption (W)
# TYPE network_metric_link_energy gauge
network_metric_link_energy{current_node="kind-worker",remote_ip="172.18.0.2",remote_node="kind-control-plane"} 2706.26
network_metric_link_energy{current_node="kind-worker",remote_ip="172.18.0.4",remote_node="kind-worker2"} 2676.93
# HELP network_metric_packet_loss Network Metric for packet loss (percentage)
# TYPE network_metric_packet_loss gauge
network_metric_packet_loss{current_node="kind-worker",remote_ip="172.18.0.2",remote_node="kind-control-plane"} 0.0
network_metric_packet_loss{current_node="kind-worker",remote_ip="172.18.0.4",remote_node="kind-worker2"} 0.0
# HELP network_metric_latency Network Metric for latency (ms)
# TYPE network_metric_latency gauge
network_metric_latency{current_node="kind-worker",remote_ip="172.18.0.2",remote_node="kind-control-plane"} 0.58
network_metric_latency{current_node="kind-worker",remote_ip="172.18.0.4",remote_node="kind-worker2"} 0.38
# HELP network_metric_link_failure Link failure (0=okay; 1=not okay)
# TYPE network_metric_link_failure gauge
```

Figure 44 Metrics saved in the internal Prometheus in the first cluster.

In order to show this last capture, we have performed a port-forwarding to the host system, since this service is only exposed within the nemo-net network, so that other systems or users cannot access them autonomously.

In parallel, the Nemex component performs periodic queries to this Prometheus, and integrates the metrics together with the information of the cluster and the time when the exposure is performed. In this way, the components that require this information will be able to access it.

In Figure 45 we can see how from RabbitMQ we can see the pushes to the mncc_cmv queue (mNCC Cost Map Vectorial), where at the beginning we have a series of short peaks and after a while more elongated peaks appear. This is due to the fact that, at the beginning, only one cluster was launched, but when testing with several clusters, they started to send the metrics slightly out of sync. Also, in the upper part of the figure, the logs are shown from the two Nemex pods. Reading the first lines it is possible to check that the upper-left window shows the "kind-kind" cluster and the upper-right the "kind-kind-2" cluster. These clusters were launched in two different devices in the same lab, in order to check the correct behavior of the multi-cluster monitoring and exposure functionality.



```
INFO:__main__: [x] Sent costs-map-vectorial:{'metadata': {'name': 'mncc-sample', 'cluster': 'kind-kind', 'timestamp'
: 1734024145.454677, 'metrics': [{'name': 'link-failure', 'type': 'integer', 'metric': 'failures per hour'}, {'name'
: 'link-energy', 'type': 'integer', 'metric': "I'm not sure"}, {'name': 'packet-loss', 'type': 'integer', 'metric':
'packets per hour'}, {'name': 'bandwidth', 'type': 'float', 'metric': 'bits'}, {'name': 'latency', 'type': 'float',
'metric': 'nanoseconds'}]}, 'network-metrics': {'nodes': [{'name': 'kind-worker', 'ip': '172.18.0.3'}, {'name': 'kin
d-control-plane', 'ip': '172.18.0.2'}, {'name': 'kind-worker2', 'ip': '172.18.0.4'}], 'links': [{'name': 'kind-worke
r_kind-control-plane', 'source': 'kind-worker', 'destination': 'kind-control-plane', 'metrics': {'link-failure': 0,
'link-energy': 2636, 'packet-loss': 0, 'bandwidth': 19975.81, 'latency': 0.33}}, {'name': 'kind-worker_kind-worker2'
, 'source': 'kind-worker', 'destination': 'kind-worker2', 'metrics': {'link-failure': 0, 'link-energy': 2741, 'packe
t-loss': 0, 'bandwidth': 20769.64, 'latency': 0.27}}, {'name': 'kind-worker2_kind-control-plane', 'source': 'kind-wo
rker2', 'destination': 'kind-control-plane', 'metrics': {'link-failure': 0, 'link-energy': 2663, 'packet-loss': 0, '
bandwidth': 20177.35, 'latency': 0.25}}, {'name': 'kind-worker2_kind-worker', 'source': 'kind-worker2', 'destination
': 'kind-worker', 'metrics': {'link-failure': 0, 'link-energy': 2550, 'packet-loss': 0, 'bandwidth': 19321.4, 'laten
cy': 0.33}}]}}
INFO:__main__:Metrics saved in: datasets/metric-1734022883.json
Connecting Rabbit MQ.
INFO:__main__: [x] Sent costs-map-vectorial:{'metadata': {'name': 'mncc-sample', 'cluster': 'kind-kind', 'timestamp'
: 1734024325.690893, 'metrics': [{'name': 'link-failure', 'type': 'integer', 'metric': 'failures per hour'}, {'name'
: 'link-energy', 'type': 'integer', 'metric': "I'm not sure"}, {'name': 'packet-loss', 'type': 'integer', 'metric':
'packets per hour'}, {'name': 'bandwidth', 'type': 'float', 'metric': 'bits'}, {'name': 'latency', 'type': 'float',
'metric': 'nanoseconds'}]}, 'network-metrics': {'nodes': [{'name': 'kind-worker', 'ip': '172.18.0.3'}, {'name': 'kin
d-control-plane', 'ip': '172.18.0.2'}, {'name': 'kind-worker2', 'ip': '172.18.0.4'}], 'links': [{'name': 'kind-worke
r_kind-control-plane', 'source': 'kind-worker', 'destination': 'kind-control-plane', 'metrics': {'link-failure': 0,
'link-energy': 2533, 'packet-loss': 0, 'bandwidth': 19197.0, 'latency': 0.3}}, {'name': 'kind-worker_kind-worker2',
'source': 'kind-worker', 'destination': 'kind-worker2', 'metrics': {'link-failure': 0, 'link-energy': 2769, 'packet-
loss': 0, 'bandwidth': 20985.99, 'latency': 0.31}}, {'name': 'kind-worker2_kind-control-plane', 'source': 'kind-work
er2', 'destination': 'kind-control-plane', 'metrics': {'link-failure': 0, 'link-energy': 2669, 'packet-loss': 0, 'ba
ndwidth': 20224.23, 'latency': 0.25}}, {'name': 'kind-worker2_kind-worker', 'source': 'kind-worker2', 'destination':
 'kind-worker', 'metrics': {'link-failure': 0, 'link-energy': 2694, 'packet-loss': 0, 'bandwidth': 20414.62, 'latenc
```

ubuntu@nemo-codeco: ~
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help X server Exit
Quick connect...
5. ubuntu@r

```
ubuntu@nemo-codeco:~$ kubectl logs -n nemo-net nemex-c658cc8dc-ddsc4
INFO:__main__: [x] Sent costs-map-vectorial:{'metadata': {'name': 'mncc-sample', 'cluster': 'kind-kind2', 'times
ures per hour'}, {'name': 'link-energy', 'type': 'integer', 'metric': "I'm not sure"}, {'name': 'packet-loss', '
 'bits'}, {'name': 'latency', 'type': 'float', 'metric': 'nanoseconds'}]}, 'network-metrics': {'nodes': [{'name'
.2'}, {'name': 'kind-kind2-worker', 'ip': '172.18.0.3'}], 'links': [{'name': 'kind-kind2-worker2_kind-kind2-cont
 {'link-failure': 0, 'link-energy': 2771, 'packet-loss': 0, 'bandwidth': 20992.93, 'latency': 0.15}}, {'name': '
-worker', 'metrics': {'link-failure': 0, 'link-energy': 2978, 'packet-loss': 0, 'bandwidth': 22563.07, 'latency'
destination': 'kind-kind2-control-plane', 'metrics': {'link-failure': 0, 'link-energy': 2880, 'packet-loss': 0,
e': 'kind-kind2-worker', 'destination': 'kind-kind2-worker2', 'metrics': {'link-failure': 0, 'link-energy': 2866
INFO:__main__:Metrics saved in: datasets/metric-1734024052.json
INFO:__main__: [x] Sent costs-map-vectorial:{'metadata': {'name': 'mncc-sample', 'cluster': 'kind-kind2', 'times
ures per hour'}, {'name': 'link-energy', 'type': 'integer', 'metric': "I'm not sure"}, {'name': 'packet-loss', '
 'bits'}, {'name': 'latency', 'type': 'float', 'metric': 'nanoseconds'}]}, 'network-metrics': {'nodes': [{'name'
.2'}, {'name': 'kind-kind2-worker', 'ip': '172.18.0.3'}], 'links': [{'name': 'kind-kind2-worker2_kind-kind2-cont
 {'link-failure': 0, 'link-energy': 2969, 'packet-loss': 0, 'bandwidth': 22494.77, 'latency': 0.11}}, {'name': '
-worker', 'metrics': {'link-failure': 0, 'link-energy': 2907, 'packet-loss': 0, 'bandwidth': 22028.93, 'latency'
destination': 'kind-kind2-control-plane', 'metrics': {'link-failure': 0, 'link-energy': 3007, 'packet-loss': 0,
e': 'kind-kind2-worker', 'destination': 'kind-kind2-worker2', 'metrics': {'link-failure': 0, 'link-energy': 2968
INFO:__main__:Metrics saved in: datasets/metric-1734024052.json
INFO:__main__: [x] Sent costs-map-vectorial:{'metadata': {'name': 'mncc-sample', 'cluster': 'kind-kind2', 'times
ures per hour'}, {'name': 'link-energy', 'type': 'integer', 'metric': "I'm not sure"}, {'name': 'packet-loss', '
 'bits'}, {'name': 'latency', 'type': 'float', 'metric': 'nanoseconds'}]}, 'network-metrics': {'nodes': [{'name'
.2'}, {'name': 'kind-kind2-worker', 'ip': '172.18.0.3'}], 'links': [{'name': 'kind-kind2-worker2_kind-kind2-cont
 {'link-failure': 0, 'link-energy': 2976, 'packet-loss': 0, 'bandwidth': 22550.51, 'latency': 0.23}}, {'name': '
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
```

No seguro | 132.227.122.23:30174/#/queues/%2F/mncc_cmv

RabbitMQ 3.12.13 Erlang 26.2.2

Refreshed 2024-12-12 18:31:42 Refresh every 5 seconds
Virtual host All
Cluster nemo-rabbitmq
User nemo-user

Figure 45 On the top, the logs of two Nemex components deployed in two clusters. In the bottom, the RabbitMQ updates.

These metrics in turn are accessible by the Meta Orchestrator and CF-DRL components to make decisions having knowledge of the state of the monitored network.

7 Conclusions

This document reports the final outcomes of the components developed in Work Package 2, namely the CMDT, the CF-DRL and the mNCC. It also includes details on the TSN capabilities integrated on the NEMO edge-cloud continuum for satisfying specific use cases requiring time-sensitive behavior.

This deliverable summarizes the progress of the developments, the integration aspects with respect other components of the NEMO stack, as well as provides exemplary details of the functional behavior of the components.

Final integration efforts will be reported in deliverables from other work packages when required.

8 References

[1] N. w. partners, "NEMO_D2.2-Enhancing-NEMO-Underlying-Technology_v1.1.pdf," [Online]. Available: https://meta-os.eu/wp-content/uploads/2024/09/NEMO_D2.2-Enhancing-NEMO-Underlying-Technology_v1.1.pdf. [Accessed 15 10 2024].

[2] "ETSI TeraFlowSDN Alignment With TIP OOPT MUST Requirements | TeraFlow," [Online]. Available: https://www.teraflow-h2020.eu/publications/etsi-teraflowsdn-alignment-tip-oopt-must-requirements. [Accessed 1 12 2024].

[3] R. V. (. R. M. (. Pol Alemany (CTTC), "D4.1_Preliminary evaluation of TeraFlow security and B5G network integration," [Online]. Available: https://www.teraflow-h2020.eu/sites/teraflow/files/public/content-files/deliverables/D4.1_Preliminary%20evaluation%20of%20TeraFlow%20security%20and%20B5G%20network%20integration.pdf. [Accessed 5 11 2024].

[4] B. W. a. D. D. a. R. R. a. T. S. a. J. Mullooly, "A YANG Data Model for the RFC 9543 Network Slice Service," *IETF,* no. draft-ietf-teas-ietf-network-slice-nbi-yang-17, 2024.

[5] J. Ramseyer, "Peering API," *IETF draft,* no. draft-ramseyer-grow-peering-api-00, 2024.

[6] T. L. S. H. Y. Rekhter, "A Border Gateway Protocol 4 (BGP-4)," *Draft standard,* no. rfc4271, 2006.

[7] G. D. a. C. F. a. K. T. a. M. C. a. D. B. a. B. Decraene, "Border Gateway Protocol - Link State (BGP-LS) Extensions for Segment Routing over IPv6 (SRv6)," *RFC editor,* no. RFC 9514, 2023.

[8] "ETSI Software Development Group TeraFlowSDN - SDG TFS," [Online]. Available: https://tfs.etsi.org/news/20230720_teraflowsdn_release_21/. [Accessed 1 12 2024].

9 Annexes

9.1 TFS interface and example intent

TeraFlowSDN controller input (python function) for descriptor file. L2VPN creation request.

```
        vpn_descriptor={
                    "services": [
                       {"service_id": {
                            "context_id": {"context_uuid": {"uuid": self.__params['context_uuid']}},
                            "service_uuid": {"uuid": self.__params['service_uuid']} },
                         "service_type": 2,
                         "service_status": {"service_status": 1},
                         "service_endpoint_ids": [
                                        {"device_id": {"device_uuid": {"uuid": self.__params['node_src']}}, "endpoint_uuid": {"uuid": "0/0/1-
"+self.__params['endpoint_src']}},
                                        {"device_id": {"device_uuid": {"uuid": self.__params['node_dst']}}, "endpoint_uuid": {"uuid": "0/0/1-
"+self.__params['endpoint_dst']}}
                         ],
                         "service_constraints": [
                            {"custom": {"constraint_type": "bandwidth[gbps]", "constraint_value": self.__params['bandwidth']}},
                            {"custom": {"constraint_type": "latency[ms]", "constraint_value": self.__params['latency']}}
                         ],
                         "service_config": {"config_rules": [
                            {"action": 1, "custom": {"resource_key": "/settings", "resource_value": {
                            }}},
                                             {"action": 1, "custom": {"resource_key": "/device["+self.__params['node_src']+"]/endpoint[0/0/1-
"+self.__params['endpoint_src']+"]/settings", "resource_value": {
                              "sub_interface_index": 0,
                              "ni_name":self.__params['ni_name'],
                              "vlan_id": int(self.__params['vlan_id']),
                              "circuit_id": self.__params['circuit_id'],
                              "remote_router":self.__params['node_dst']
                            }}},
                                             {"action": 1, "custom": {"resource_key": "/device["+self.__params['node_dst']+"]/endpoint[0/0/1-
"+self.__params['endpoint_dst']+"]/settings", "resource_value": {
                              "sub_interface_index": 0,
                              "ni_name":self.__params['ni_name'],
                              "vlan_id": int(self.__params['vlan_id']),
                              "circuit_id": self.__params['circuit_id'],
                              "remote_router":self.__params['node_src']
                            }}}
                         ]}
                       }
                    ]
                }
```

mNCC Intent: L2VPN request.

```
Intent:
  id: 'new_intent_2'
  userLabel: 'cloud_continuum'
  intentExpectations:
    - expectationId: '1'
      expectationVerb: 'DELIVER'
      expectationObject:
        objectType: 'L2VPN'
        objectInstance: 'l2vpn_tfs_1'
        objectContexts:
          - contextAttribute: 'nodeSrc'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: '3.3.3.3'
          - contextAttribute: 'nodeDst'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: '5.5.5.5'
          - contextAttribute: 'endpointSrc'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'GigabitEthernet0/0/0/1'
          - contextAttribute: 'endpointDst'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'GigabitEthernet0/0/0/1'
          - contextAttribute: 'vlanId'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: '999'
          - contextAttribute: 'niName'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'ninametfs'
      expectationTargets:
        - targetName: 'bandwidth'
          targetCondition: 'IS_EQUAL_TO_OR_GREATER_THAN'
          targetValueRange: 10.0
        - targetName: 'latency'
          targetCondition: 'IS_LESS_THAN'
          targetValueRange: 15.2
      expectationContexts:
        - contextAttribute: 'url'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'http://192.168.165.10/restconf/data/ietf-l2vpn-svc:l2vpn-svc/vpn-services'
  intentContexts:
    - contextAttribute: 'NEMO_WORKLOAD'
      contextCondition: 'IS_EQUAL_TO'
      contextValueRange: 'cbcb208a-d535-434b-bb35-217a64bd516b'
  intentPriority: 1
  observationPeriod: 60
  intentAdminState: 'ACTIVATED'
```

9.2 L2SM example intent

mNCC Intent: L2SM network request.

```
Intent:
  id: 'mncc_l2sm_1'
  userLabel: 'cloud_continuum'
  intentExpectations:
    - expectationId: '1'
      expectationVerb: 'DELIVER'
      expectationObject:
        objectType: 'K8S_L2_NETWORK'
        objectInstance: 'spain_network'
        objectContexts:
          - contextAttribute: 'name'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'spain_network'
          - contextAttribute: 'providerName'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'uc3m'
          - contextAttribute: 'domain'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: '172.18.0.2'
      expectationContexts:
        - contextAttribute: 'url'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'http://192.168.165.168:8080'
    - expectationId: '2'
      expectationVerb: 'DELIVER'
      expectationObject:
        objectType: 'K8S_CLUSTER_CONFIG'
        objectInstance: 'spain-network'
        objectContexts:
          - contextAttribute: 'name'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'kind-worker-cluster-1'
          - contextAttribute: 'bearer_token'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'eyJhbGciOiJSUzI1NiIsImtpZCI6Iktrd1hiRk9RWGZBVkFVWU82Rk5…'
          - contextAttribute: 'api_key'
            contextCondition: 'IS_EQUAL_TO'
            contextValueRange: 'https://172.18.0.3:6443'
      expectationContexts:
        - contextAttribute: 'k8s_l2_network'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'spain-network'
        - contextAttribute: 'url'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'http://192.168.165.168:8080'
  intentContexts:
    - contextAttribute: 'NEMO_WORKLOAD'
      contextCondition: 'IS_EQUAL_TO'
      contextValueRange: 'cbcb208a-d535-434b-bb35-217a64bd516b'
```

```
intentPriority: 1
observationPeriod: 60
intentAdminState: 'ACTIVATED'
```