Master of Science in Computer Engineering

Master's Degree Thesis

Developing the Context Discovery Actuator Profile for OpenC2 language

Supervisors:
Prof. Fulvio Valenza
Prof. Matteo Repetto
Dott. Daniele Bringhenti

Candidate:
Dott. Silvio Tanzarella

Academic Year 2023-2024

Abstract

The OpenC2 language standard provides a unified syntax and structure to send instructions to various security tools developed by different vendors that are not able to talk to each other using a common language. In this way, OpenC2 reduces the dependency on custom integrations and allows the organizations to build flexible, scalable, and vendor-agnostic security frameworks.

The openc2lib library implements the standard language described in the OpenC2 normative specifications and it can be extended by new encoders and transfers protocols. In this way, the library is suitable to create custom openC2 stacks by a minimal effort, allowing the Producer to efficiently send commands to a Consumer and receive responses. Also, the library can be extended by new Actuator Profiles, which define the semantic constraints and language extensions for specific cyber-defence functions.

The main contribution of this work is the development of a new Actuator Profile for the OpenC2 language, called Context Discovery (CTXD). Now, if the Consumers implement the CTXD Actuator Profile, the Producer can identify the services running on the digital resources, the security features they implement, and the connections among different services.

To achieve this, the architecture of the CTXD was defined, and a data model was introduced to represent the information gathered by this profile. New data types, absent from the original OpenC2 specifications, were created and implemented within the library to support the profile's functions. Additionally, conformance clauses were added to regulate the profile's behaviour and to standardize its usage.

Then, a use case was implemented where a Producer asks each Consumer for the security functions it provides and its connections to other Consumers. This approach generated a complete map of the network, providing the Producer with full visibility into the entire system. The key achievement is that the Producer only needs to know how to connect to the first Consumer and, starting from the data collected, can obtain the information about connecting to other Consumers linked with the first one. The discovery process results in a directed graph where nodes represent services and edges indicate the connections between them.

In the final part of the thesis, tests were conducted to verify the correct behaviour of the new Actuator Profile. Also, it was evaluated the ability of the CTXD to detect changes when failures happen in the system. Semantic tests were implemented to ensure the correctness of the newly introduced data types.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an increasingly interconnected world, it's crucial to protect against cyberattacks to prevent threats that could compromise the integrity of information systems. In addition to increasing in volume, attacks have also become more complex over time, adopting more sophisticated techniques to avoid the defences implemented by even the most advanced systems. This evolution of attacks has led to their automation, becoming faster and independent from human beings.

In a rapidly evolving context such as the one mentioned above, the techniques dedicated to the defence of computer networks must be equally efficient, changeable, and reactive. The interaction between the different components of the defensive system can ensure a rapid response to the attacks by coordinating the monitoring, detection, and mitigation actions of threats. However, products from different vendors may not be able to communicate effectively, often requiring proprietary languages to bridge the gap. Adopting standard solutions can help mitigate this issue.

1.1 Active Cyber Defense (ACD)

Active Cyber Defense [1] focuses on combining and automating a range of services and systems to enable rapid response actions in cyber-related situations. It consists of six functional components: Sensing, Sense-Making, Decision-Making, Acting, Messaging/Control and ACD Mission Management. These functional areas describe the expected behaviour that a cyber-defence system should have to protect the IT infrastructure. It starts from the collection and analysis of the data (Sensing and Sense

Making) that guide decision-makers in selecting and implementing the optimal solution from the available options (Decision-Making and Acting). Messaging/Control involves establishing communication channels between resources and coordinating response actions. Finally, ACD Mission Management ensures internal control over workflow within the specific operating environment.

1.2 Security Operation Centers (SOCs)

Security Operations Centers (SOCs) follows the steps established by the ACD functional areas. Their primary responsibility is to identify, assess, and address cybersecurity threats and incidents through the integration of skilled personnel and advanced technologies [2]. Defining SOCs is challenging because every organization has the flexibility to construct its own infrastructure tailored to its needs. Therefore, there is no clear model or detailed guide for setting up a SOC [3].

However, some components are common to multiple SOCs, such as SIEM, which fulfil the requirements specified in the first two functional areas of ACD (Sensing and Sense-Making) by offering security analytics capabilities through the use of log events. SIEM systems can identify ongoing attacks and anomalies, but they implement rigid solutions due to the need to locate security appliances at specific points within the infrastructure and the need for redeployment and reconfiguration when there are changes in physical topology [4]. Indeed, managing all system resources can be challenging because of the insufficient visibility into the network infrastructure [5].

1.3 GUARD framework

The GUARD framework is as an innovative approach for deploying detection and analytics processes across digital service chains [6]. Its goal is to facilitate the development and the integration of these processes in dynamic environments by automating the configuration of security analytics pipelines, so the GUARD Platform serves as a bridge, connecting security agents with detection and analytics services. The security analytics pipeline is a processing chain that begins with collecting and processing security data available in digital resources and ends with the analysis of this data to detect potential threats and to determine the appropriate response.

The Context Broker Manager abstracts the deployment of security agents and communication protocols, providing a logical representation of the entire service sequence. The main security properties of each digital resource and the kind of relationship with other resources are retrieved by the CB-Manager. Communication between the CB-Manager and digital resources can be implemented using an application interface (API), but it must consider the various types of resources provided by different vendors, so multiple interfaces are needed. A solution can be the creation of standard interfaces to security functions like the OpenC2 language.

Chapter 2

OpenC2

2.1 Introduction to OpenC2

In a heterogeneous environment where different vendors provide various security functions, a standard language is crucial for coordinating diverse technologies that would otherwise be unable to work together. The aim of OpenC2 is to facilitate synchronized cyber defence in real time among separate components that carry out cybersecurity tasks, thereby implementing the “Acting” phase of the Active Cyber Defense (ACD) framework.

It is a standardized language for machine-to-machine communication, used to issue commands and receive responses, and it is designed to be technologically agnostic to allow interoperability between different products, so it is not tied to any specific technology. This language can be extended by new cyber defence technologies, so it is not static, but it is expected to evolve in the future.

OpenC2 is an open-source project defined across multiple specifications developed by the “Organization for the Advancement of Structured Information Standards” (OASIS), a nonprofit international consortium. All the specifications and detailed information about this project can be found on the official website openc2.org.

2.2 OpenC2 Architecture

The OpenC2 Architecture [7] defines two different entities that are involved in the communication: Producer and Consumer. The Producer generates a Command with an Action that the Consumer will receive and carry out. Finally, the Consumer creates a Response and sends it back to the Producer. This schema is represented in the figure below.

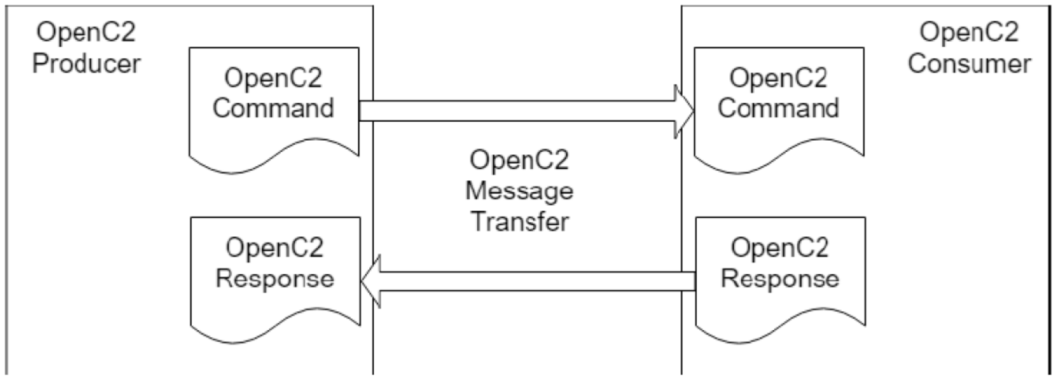


Fig. 2.1 OpenC2 communication model [7]

OpenC2 Command is composed of four fields:

- Action (required): the operation to be carried out.
- Target (required): the object upon which the action is carried out.
- Arguments (optional): it specifies how the Command should be executed.
- Actuator (optional): the entity responsible for performing the action.

The Response provides information about the results of carrying out a Command. It is composed of three fields:

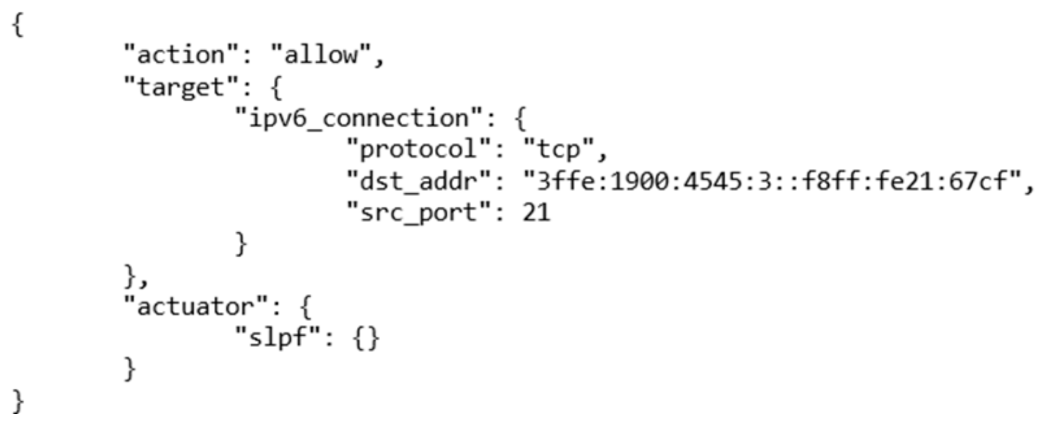
- Status (required): integer status code.
- Status text (optional): description of the Response status.
- Results (optional): results derived from the executed Command.

Command fields follow typical language patterns, allowing for the identification of a subject (Actuator), a verb (Action), an object (Target), and complements (Arguments), with each command requiring one Action and one type of Target. OpenC2 Language Specification [8] defines a set of actions, and adding new ones is not permitted to ensure compatibility with different systems.

However, it is possible to define new Targets beyond those in the specification by creating a new Actuator Profile. Actuator Profiles establish semantic constraints and extend the language for specific cyber-defence functions. This is necessary because not all actions are suitable for every use case, and certain constraints are required to more accurately model real-world scenarios. The Actuator field allows selecting the profile that will execute the Action.

All OpenC2 messages must follow the conformance clauses defined in the specifications, which explain how the language should be used.

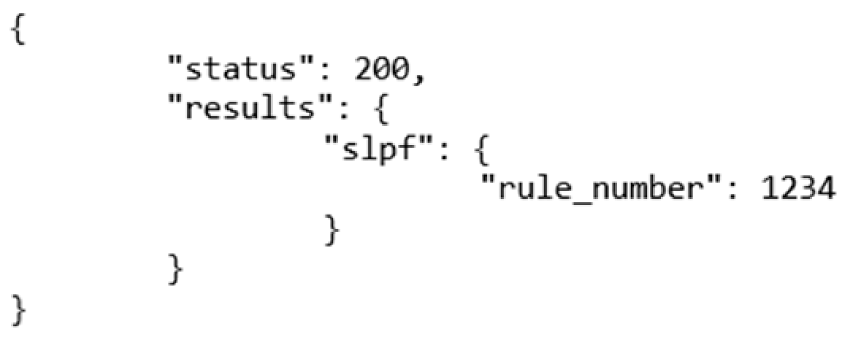
The following example illustrates an OpenC2 command in which the Producer requests that the Consumer, responsible for implementing the StateLess Packet Filter Profile, permit an IPv6 connection. As seen in the figure, the command indicates the source port, destination address, and allowed protocol.



```
{
        "action": "allow",
        "target": {
                "ipv6_connection": {
                        "protocol": "tcp",
                        "dst_addr": "3ffe:1900:4545:3::f8ff:fe21:67cf",
                        "src_port": 21
                }
        },
        "actuator": {
                "slpf": {}
        }
}
```

Fig. 2.2 example of OpenC2 Command [7]

Then, the Consumer returns, in the OpenC2 Response, a rule number associated with the allowed interaction.



```
{
        "status": 200,
        "results": {
                "slpf": {
                        "rule_number": 1234
                }
        }
}
```

Fig. 2.3 example of OpenC2 Response [7]

2.3 Implementation

OpenC2 language can be integrated into a layer model, such as the one in the figure below, where different standards and protocols are used. The OpenC2 Language Specification [8] describes the meaning of the main elements of the language while, OpenC2 Actuator Profiles [9] outline the specific parts of the language applicable to the functions of different actuators.

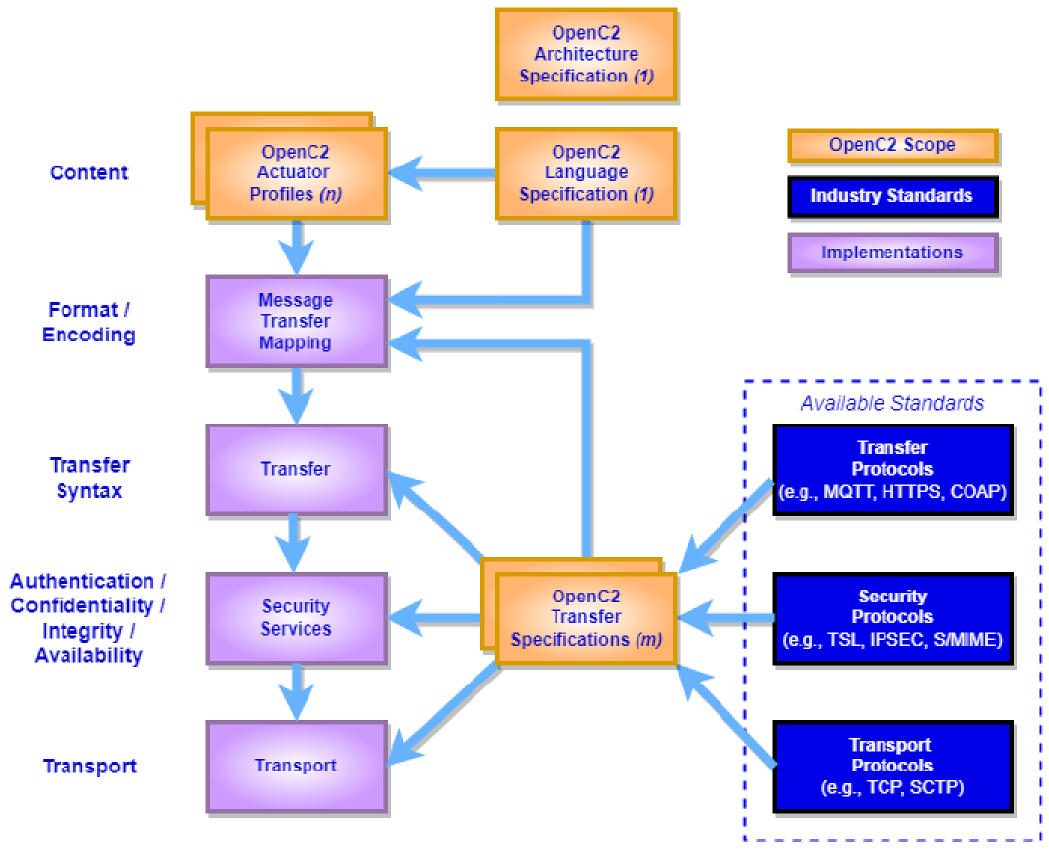

OpenC2 Architecture Specification (1)
Content
OpenC2 Actuator Profiles (n)
OpenC2 Language Specification (1)
OpenC2 Scope
Industry Standards
Implementations
Format / Encoding
Message Transfer Mapping
Available Standards
Transfer Syntax
Transfer
Transfer Protocols (e.g., MQTT, HTTPS, COAP)
Authentication / Confidentiality / Integrity / Availability
Security Services
OpenC2 Transfer Specifications (m)
Security Protocols (e.g., TSL, IPSEC, S/MIME)
Transport
Transport
Transport Protocols (e.g., TCP, SCTP)

Fig. 2.4 OpenC2 Architecture [7]

The OpenC2 language does not mandate a specific message encoding, although JSON is the default solution. Additionally, metadata such as content type, version, sender, timestamp, and correlation ID are typically included in the messages. OpenC2 Transfer Specifications outline how messages can be transmitted using transfer protocols such as HTTPS [10] or MQTT [11]. Security features can be applied using standard protocols such as TLS, IPSEC, or S/MIME to guarantee the authentication of endpoints and confidentiality and integrity of messages. Finally, transport protocols like TCP or SCTP are used to send packets over the network.

For example, OpenC2 messages can be transmitted using the Hypertext Transfer Protocol (HTTP) layered over Transport Layer Security (TLS). To achieve this, the Producer must first establish a TCP connection with the Consumer, followed by initiating a TLS session, which ensures endpoint authentication and provides enhanced security. TLS provides encryption to protect data in transit, integrity to detect tampering, and authentication to verify the identities of the communicating parties. Once the session is established, endpoints can issue OpenC2 Commands and Responses by sending HTTP requests via the POST method. OpenC2 messages are required to use a specific Uniform Resource Identifier (URI) path, which must be set to “/.well-known/openc2”. The “/.well-known/” prefix is commonly used for well-known locations.

The code provided in the figure 2.5 illustrates an example of an HTTP message carrying an OpenC2 Command. In particular, the HTTP method used is always POST because it is required by the General Requirements of the HTTP specification [10]. Also, the URI is a “/.well-known/” path. The “Content-type” indicates that the HTTP message body contains OpenC2 messages in JSON format, following by the version 1.0 of the OpenC2. The extension header "X-Request-ID" serves the purpose of uniquely identifying commands and responses, allowing for the correlation of activity between them. The Producer must include a unique identifier in the "X-Request-ID" HTTP header or in the "request_id" OpenC2 message header. In this way, each command and its corresponding response can be linked. The header of the OpenC2 Command has the original “request_id” value, which is then copied into the HTTP header. The body is the content of the OpenC2 Command. Both the HTTP and OpenC2 header of the Response message contain the same fields of the Command.

```
POST /.well-known/openc2 HTTP/1.1
Content-type: application/openc2+json;version=1.0
Date: Wed, 19 Dec 2018 22:15:00 GMT
X-Request-ID: d1ac0489-ed51-4345-9175-f3078f30afe5

{
        "headers": {
                "request_id": "d1ac0489-ed51-4345-9175-f3078f30afe5",
                "created": 1545257700000,
                "from": "oc2producer.company.net",
                "to": [
                        "oc2consumer.company.net"
                ]
        },
        "body": {
                "openc2": {
                        "request": {
                                "action": "...",
                                "target": "...",
                                "args": "..."
                        }
                }
        }
}
```

Fig. 2.5 HTTP message carrying an OpenC2 Command [10]

The Response to this OpenC2 Command is the following [10]:

```
HTTP/1.1 200 OK
Date: Wed, 19 Dec 2018 22:15:10 GMT
Content-type: application/openc2+json;version=1.0
X-Request-ID: d1ac0489-ed51-4345-9175-f3078f30afe5
{
        "headers": {
                "request_id": "d1ac0489-ed51-4345-9175-f3078f30afe5",
                "created": 1545257710000,
                "from": "oc2consumer.company.net",
                "to": [
                        "oc2producer.company.net"
                ]
        },
        "body": {
                "openc2": {
                        "response": {
                                "status": 200,
                                "status_text": "...",
                                "results": "..."
                        }
                }
        }
}
```

Fig. 2.6 HTTP message carrying an OpenC2 Response [10]

2.4 Base Components and Structures

The abstract terminology used to specify OpenC2 data types is unaffected by how they are implemented in particular scenarios. By defining data types in this abstract way, OpenC2 ensures flexibility and interoperability, allowing the same commands to be represented and communicated across different systems and platforms, regardless of the underlying formats or protocols. OpenC2 data can be divided into two different types: primitive and structures.

Primitive types are the most basic forms of data in programming languages (e.g., binary, boolean, integer). Structures, on the other hand, are collections of multiple data elements grouped together under a single unit. Some examples of structures used in OpenC2 include:

- ArrayOf(vtype);
- Choice;

- Enumerated;
- Map;
- Record.

2.5 Actuator Profile

An OpenC2 Actuator Profile describes the security functions performed by an entity within the network by defining a subset of the OpenC2 language. To achieve this, a Profile has the possibility to extend the OpenC2 language with new targets, Command Arguments and Actuator Specifiers.

There is currently only one Actuator Profile (AP) that has been approved by the OASIS Technical Committee, which is the "Stateless Packet Filter" [12], identified by the namespace identifier (nsid) "slpf". The purpose of the Stateless Packet Filter is to allow or block network traffic based on fixed criteria, such as the source address, destination address, and port numbers. SLPF cannot introduce new Actions due to the conformance clause, but it defines a new Target and four Command Arguments beyond those specified in the Language Specification [8]. Also, it is possible to specify specific actuator specifiers to address the SLPF. Two matrices are specified to highlight which pairs of Action-Target and Action-Command Arguments are permitted. Additionally, new conformance clauses are introduced to standardize the implementation of the SLPF.

Chapter 3

OpenC2 library

3.1 Introduction to OpenC2 library

An OpenC2 library, called “openc2lib”, was created to implement the functions defined by the OpenC2 initiative, and it was written in Python because this language can be easily used in different software environments. Also, Python supports reflective programming paradigms, which is the ability to examine, introspect, and modify its own structure and behaviour while executing, so the user can add new profiles, transport protocols, and encoding formats without changing the core of the library.

The goal of the library is to create a dynamic stack for transferring Commands and Responses as defined in OpenC2 specifications. With the library, it is possible to implement both the Consumer and the Producer and define the full stack for each one. Furthermore, the library provides a flexible and extensible mechanism for serialization using Python dictionaries as intermediary data abstraction. Python dictionaries are ordered and changeable collections that do not allow duplicates and they are used to store data values in key:value pairs. With the intermediary dictionary representation, OpenC2 objects can be serialized without the need to create custom and repetitive methods for each of them, because the library can easily be extended with additional serialization formats, such as XML, YAML and JSON (only the latter is currently implemented).

The overall architecture and workflow of the OpenC2 library are shown in the figure below, where it is possible to note that the encoding workflow of the Producer is the same as the decoding one of the Consumer, but they operate in different directions. The Security Controller is an external software which uses the library to send and receive OpenC2 messages. The JSONEncoder permits the serialization of messages in JSON format, while HTTPSTransfer enables message transfer via the HTTPS protocol. Furthermore, the Consumer can select the right actuator specified in the Command that has just been received.

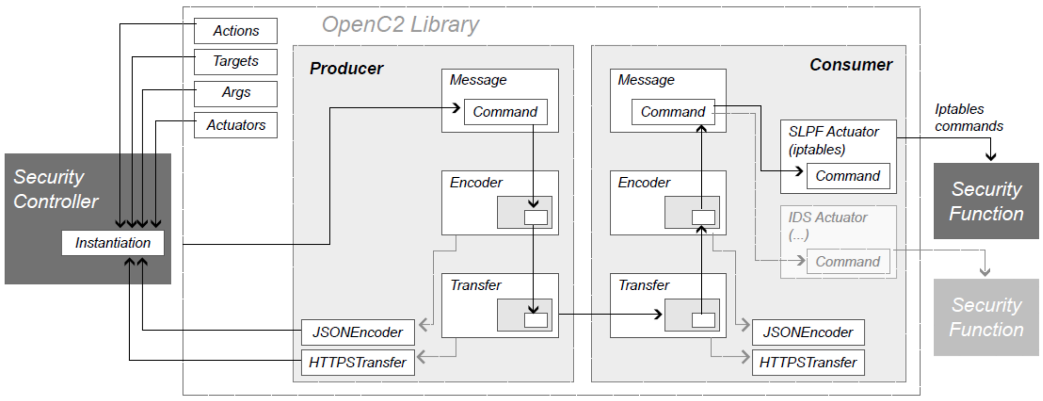


Fig. 3.1 OpenC2 library architecture

3.2 The openc2lib Architecture

The openc2lib architecture can be divided into different modules, with each one implementing a different part of the specifications.

- Data types and structures are implemented in the "types" folder which contains three subfolders:
  - Base: base types (structures) in the Language Specification [8] (section 3.1.1). Each OpenC2 object must derive from these classes, which affects serialization operations. This folder contains the implementations of the *Choice*, *ArrayOf*, *Record*, and other classes of OpenC2 structures. Every base type must implement *todict* and *fromdict* method to transform the object into a Python dictionary and vice versa.
  - Data: data types defined in the Language Specification [8] (section 3.4.2). The classes are named according to the definitions provided by the specification. Example of data types are *IPv4Addr*, *L4Protocol*

and *Nsid*. A data type class must be a child class that inherits all the attributes and methods of a parent class, which is a base type, in order to be automatically serialized.

  - Targets: Targets defined in the Language Specification [8] (section 3.4.1). A target class must be a child class that inherits all the attributes and methods of a base type. An example of target is the *Features* class, which is defined as *ArrayOf(Feature)*, where *ArrayOf* is the base type and *Feature* is the data type.
  -

- The folder named "core" contains the classes that are useful to implement the OpenC2 standard language starting from the data already defined. The modules can be divided based on their function. The first group defines the fields of the OpenC2 Commands and Responses:
  - Actions: it outlines the list of Actions defined in the Language Specification [8] (section 3.3.1.1).
  - Actuator: it describes the "Actuator" element used in Commands. It does not include any element concerning the concrete implementation of Actuators for specific security functions. Its only attribute is the Profile namespace identifier (nsid).
  - Args: it details the Command Arguments as specified in the Language Specification [8] (section 3.3.1.4). It is structured as a Map because new instances can be added when implementing new profiles.
  - Command: this module describes the OpenC2 Command structure as defined in the Language Specification [8] (section 3.2).
  - StatusCode: it indicates the status of processing an OpenC2 Command.
  - Results: it defines the basic structure for Results carried in a Response. This class can be extended by profiles definitions with additional fields. It implements the definition of Result in the Language Specification [8] (section 3.3.2.2).
  - TargetRegister: this class registers all available Targets, both provided by the openc2lib and by Profiles.

  - Profile: it is the openc2lib interpretation of the Profile concept. It basically defines a Profile namespace and the language extensions that are defined for that message. A Profile is fully transparent to concrete implementation for controlling specific security functions.
  - Register: this class registers all available elements, both provided by the openc2lib and by Profiles. Profiles may fill in with additional definitions, to make their classes and names available to the core system for encoding/decoding purposes.

  The second group defines the Producer and the Consumer:

  - Producer: it is used to create an OpenC2 stack with an “Encoder” and a “Transfer” protocol. It implements the function *sendcmd* that sends an openc2lib Command and internally create the ‘Message’ metadata that will be encoded and transferred.
  - Consumer: it implements the expected behaviour of an OpenC2 Consumer server that dispatches OpenC2 Commands to the Actuators. It creates the OpenC2 stack to process a Message. The function *run* allows the Consumer to receive message in the network while the function *dispatch* scans the Actuator Profile carried in the Command and select one or more Actuators that will process the Command.

  The third group defines how a message must be transmitted:

  - Encoder: this module provides base encoding functions to translate openc2lib objects into an intermediary dictionary-based representation and vice versa.
  - Transfer: it is the interface that defines the basic behaviour of the Transfer protocols.
  - Message: it defines the OpenC2 Message structure, as defined in Language Specification [8] (section 3.2).

- In the folder “encoders” all possible encoders are specified:

  - JSONEncoder: this class implements the “Encoder” interface for the JSON format. It leverages the intermediary dictionary representation. It can be used to create an OpenC2 stack in Consumer and Producer.

- In the folder "transfers" all possible transfers protocols are specified, but only HTTP/HTTPS protocol is implemented in "http" folder with two modules:
  - HTTPTransfer: it provides an implementation of the HTTP Specification [10]. it can be used to create an OpenC2 stack in the Consumer and Producer. The function *send* can transmit an HTTP message over the network while the function *receive* allows to listen and receive OpenC2 messages.
  - Message: this class implements the HTTP-specific representation of the OpenC2 Message metadata defined in HTTP Specification [10] (section 3.3.2).
- The "actuators" folder contains all the actuators for all profiles. Each actuator is a distinct class, and its behaviour is defined according to the profile it implements.
- The "profiles" folder contains the definition of profiles provided with openc2lib. Within this folder, the Stateless Packet Filter Profile (SLPF) is defined, including all its specifiers, command arguments, and result types. Additionally, the SLPF specifies a unique target for this profile (RuleID) and the data types that extend the command arguments, which are not included in the Language Specification [8] (DropProcess and Direction) are present in this folder. A validation module establishes the constraints for usable Actions and Command Arguments within the SLPF.

3.3 Serialization and deserialization processes of a message

When a Command or Response is ready, it must be encoded or decoded into a serialization format. While JSON is the most common encoding format for OpenC2 messages, other formats are also permitted. The choice of the encoder is influenced by the environment in which OpenC2 is applied, as well as the capabilities and limitations of the selected transfer protocol.

Python dictionaries cannot serialize directly all OpenC2 data because OpenC2 defines its own data types and structures in the Language Specification [8] so, to overcome this issue, it is necessary to define two functions for the base types: the first function creates a dictionary from OpenC2 element, and the second one creates OpenC2 elements from a dictionary.

The deserialization process is similar to the serialization one but operates in a different direction. It takes as input the JSON message to decode and the openc2lib class to convert the JSON to. First, the JSON message is deserialized into a dictionary, and then the dictionary is transformed into the openc2lib class.

The deserialization and serialization processes work recursively because if the object cannot be transformed directly into a dictionary, these processes keep calling the Encoder class until the object is successfully converted.

3.4 Message Transfer and Reception via HTTP

Once an OpenC2 Command has been encoded in the appropriate serialization format, it must be sent to the Consumer. The transfer of a message is based on a common module, the Transfer class, which provides the base methods to send and receive messages. For each protocol, a child class must be implemented which inherits the attributes and functions of the Transfer class and defines the appropriate format of the message. For HTTP communication, the child class is the HTTPTransfer, which implements the interface for the HTTP and HTTPS protocols. It is built on Flask, which simplifies HTTP communication for development and testing.

A new transfer message is created using the Message class from the 'transfers/http' folder, with the encoded OpenC2 Command as its body. This class defines the appropriate format of the HTTP message. It is important to note that the "request_id" field in the header serves as a unique identifier generated by the Producer and is carried over by the Consumer into all Responses, allowing for reference to a specific Command, transaction, or event chain.

Next, the HTTP message header is created in accordance with the HTTPS Specification [10] (section 3.3.2), which includes the content type, date, and the unique identifier. At this point, the HTTP message is ready to be sent to the Consumer via the POST method to the "/.well-known/openc2" endpoint, as per the specification [10].

Vice versa, the Consumer must be able to receive an HTTP message, decode its content, generate a Response, and sent it back to the Producer. To achieve this, the *receive* method, in the HTTPTransfer module, implements the Transfer interface to listen for and receive OpenC2 messages. Internally, it uses Flask as the HTTP server, and the process of creating the appropriate headers is similar to what is done on the Producer side.

3.5 Using openc2lib

Up to this point, the technical characteristics of the Python library have been discussed. Now, the focus shifts to the implementation process. Two entities are introduced to implement the openc2lib library: Server and Controller. The former simulates the behaviour of the OpenC2 Consumer which could be a firewall or an IoT device. The latter simulates the OpenC2 Producer, which could be, for example, a SOAR that dispatches commands to various consumers within the network.

In the following sections, the implementations of both the Server and the Controller will be defined to utilize the openc2lib and initiate OpenC2 communication. Furthermore, an example of sending OpenC2 Command is reported in the last section.

3.5.1 Server Implementation

A Server is intended to instantiate and run the OpenC2 Consumer. Instantiation requires the definition of the protocol stack and the configuration of the Actuators that will be exposed. The Consumer object is initialized with a constructor that takes as parameters: a string that identifies the Consumer, a list of available Actuators, an instance of the Encoder and of the Transfer protocol that will be used to send and receive messages. For both the Encoder and Transfer, the constructor requires not the generic class, but their more specific child classes, such as JSONEncoder and HTTPTransfer. The *run* function is the entry point for the Consumer and must be called to enable the Server to listen for messages on the network. In the example provided in the code below, a Server is instantiated with an Actuator that implements the SLPF Profile and listens on the HTTP channel at a specified IP address and port (in this case, it will be listening on the loopback interface).

```
import openc2lib as oc2

from openc2lib.encoders.json_encoder import JSONEncoder
from openc2lib.transfers.http_transfer import HTTPTransfer

import openc2lib.profiles.slpf as slpf
from openc2lib.actuators.iptables_actuator import IptablesActuator

actuators = {}
actuators[(slpf.Profile.nsid,'iptables')]=IptablesActuator()
c = oc2.Consumer("consumer.example.net",
                    actuators,
                    JSONEncoder(),
                    HTTPTransfer("127.0.0.1", 8080))
c.run()
```

3.5.2 Controller implementation

A Controller is intended to instantiate an OpenC2 Producer and to use it to control a remote security function. It is initialized with a constructor that takes as parameters: a string that identifies the Producer, an instance of an Encoding class derived from base “Encoder” and an instance of a Transfer protocol derived from base “Transfer”. For both the Encoder and Transfer, like for the Server, the constructor requires not the generic class, but their more specific child classes, such as JSONEncoder and HTTPTransfer. Below there is an example of a correct initialization of an openc2lib Producer that encodes messages in JSON and sends them to an openc2lib Consumer at a specified IP address and port:

```
import openc2lib as oc2
from openc2lib.encoders.json import JSONEncoder
from openc2lib.transfers.http import HTTPTransfer

p = oc2.Producer("producer.example.net",
                    JSONEncoder(),
                    HTTPTransfer("127.0.0.1", 8080))
```

3.5.3 Sending OpenC2 Commands

Once the Controller has been defined, the OpenC2 Command can be defined with all its fields. Command Arguments that will be used in the Command can be defined in this way:

```
arg = oc2.Args({'response_requested': oc2.ResponseType.complete})
```

The Command below indicates that the Producer wants to know with a query Action the OpenC2 versions and profiles supported by the Consumer. Note that no actuators are specified.

```
cmd = oc2.Command(oc2.Actions.query,
                    oc2.Features([oc2.Feature.versions, oc2.Feature.profiles),
                    arg,
                    actuator= None)
```

In this further example, the SLPF with hostname "firewall" is the chosen Actuator Profile and will allow incoming traffic for the specified range of IPv4 address, and this Command must be executed, on the Consumer side, by the SLPF Profile. After, the Controller sends the Command with the function *sendcmd*, which returns the Response.

```
import openc2lib.profiles.slpf as slpf
pf = slpf.Specifiers({'hostname':'firewall'})
arg = slpf.Args({'response_requested': oc2.ResponseType.complete,
                'direction': slpf.Direction.ingress})

cmd = oc2.Command(oc2.Actions.allow,
                    oc2.IPv4Net("172.19.0.0/24"),
                    arg, actuator=pf)
resp = p.sendcmd(cmd)
```

Chapter 4

Thesis objectives

OpenC2 language is defined through a series of specifications and can be extended by new Actuator Profiles, which represent additionally security capabilities. Defining a new Actuator requires adherence to the standards set by OpenC2. By following these specifications, the Profile ensures compatibility with other endpoints that implement the OpenC2 language, enabling seamless interoperability across different systems.

The main objective of this thesis is the development of a new Actuator Profile for the OpenC2 language, called Context Discovery (CTXD). It has been introduced because the Producer does not have detailed knowledge of the services that are running into the network, the interactions between them and the security features that they implement. Identifying a service involves finding key details such as the operating system, hostname, or application type. By using only the standard OpenC2 language, these details cannot be obtained because there are no Targets that provide this information.

One of the key pieces of information collected by the CTXD from a service is the details on how to connect to its associated peers. This data is crucial for enabling a recursive discovery process. Starting with an OpenC2 Consumer, the CTXD allows the OpenC2 Producer to identify its connected peers. Once identified, the Producer then connects to each peer and queries them for their own attributes and connection details. In this way, the Producer only needs to know how to connect to the first Consumer and then, with the recursive discovery, can have full visibility into the network. The discovery process results in a directed graph where nodes represent services and edges indicate the connections between them.

Once the main objectives of the CTXD Actuator Profile were defined, its architecture was designed in accordance with OpenC2 guidelines. This resulted in the creation of new data types, which are not part of the original standard and are detailed in this thesis. The entire architecture and behaviour of the CTXD was then implemented in the openc2lib library.

In the end, the CTXD was validated in a use case scenario which involves a web server deployed as a Kubernetes application within an OpenStack-based cloud infrastructure. The goal was to explore the relationships between OpenStack, Kubernetes, cloud-based virtual machines (VMs), and containers. The result of this validation process was a directed graph which represents these relationships. Additionally, failures were deliberately introduced to test if the discovery process could adapt to changes in the environment

Chapter 5

OpenC2 Profile for Context Discovery

5.1 Adding a new Actuator Profile

New cyber defence functions can be added to the OpenC2 language by defining new Actuator Profiles. The starting point, for implementing a new Actuator Profile, is the identification of the security functions, and it is important to verify that they are not already implemented by the standard OpenC2 to prevent unnecessary duplication.

Once the goals have been defined, the next step is to determine what type of data the new Profile will manage. If these data are not already included in the OpenC2 specification, they can be defined, but they will be valid only within the context of the Actuator Profile. Defining a new data type involves selecting the appropriate base type (e.g. *ArrayOf(vtype)*, *Map*, *Choice*. . .), providing a description of what the data represents and, if necessary, specifying any conformance clauses associated with the data type.

After defining the new data types, the Actuator Profile can introduce new Targets and Command Arguments, which can only be used in the Command if the specified Profile matched the Actuator Profile that was just defined. The Response may also include these new data types. However, according to OpenC2 specifications, the Actuator Profile cannot add new Actions to the ones already defined.

Finally, all conformance clauses must be defined to ensure the correct behaviour of the Actuator Profile. This includes specifying the allowed set of Targets for each supported Action and defining how the Consumer should respond based on Command Arguments.

This chapter defines the architecture of a new Actuator Profile, starting with the causes that drive the need for its implementation. Then, all the new data types introduced will be defined along with their conformance clauses. In the final sections, practical examples are provided to demonstrate how the Profile works.

5.2 “Query features” Command

When implementing the OpenC2 standard language, the Producer does not have detailed knowledge of the security features or configurations of every device connected to the network. As a result, the Producer must function without a comprehensive view of the entire network and rely on discovery mechanisms to gather the necessary information about devices, security services, and their interconnections.

The OpenC2 language, using the *query* Action along with the Target *features*, allows the Producer to gather information about the Actuator’s capabilities. This enables the Producer to obtain detailed information about the OpenC2 Language versions, profiles, rate limit, supported Actions and applicable Targets implemented by the Actuator. According to the specification [8], all OpenC2 devices must implement the “query features” Command. Below a practical example is shown.

Command:

```
{
    "action": "query",
    "target": {
        "features": ["rate_limit", "profiles", "versions"]
    }
}
```

Response:

```
{
    "status": 200,
    "results": {
        "versions": ["1.0"],
        "profiles": ["slpf"],
        "rate_limit": 30
    }
}
```

However, the “query features” does not allow the Producer to know the devices connected to the Consumer and no other Actions or Targets, defined in the OpenC2 specifications, permit to do so.

5.3 Goals of Context Discovery

To fill the gap left by the OpenC2 specifications, a new Actuator Profile has been introduced with the goal to abstract the services that are running into the network, the interactions between them and the security features that they implement. Identifying a service involves determining its type and the specific characteristics of that type. The service also provides essential information, such as hostname, encoding format, and transfer protocol, for connecting to it and to any linked services. In this way, the context in which the service is operating is identified. This new Actuator Profile has been named “Context Discovery”, herein referred as CTXD, with the nsid “ctxd”.

The Context Discovery employs a recursive function to achieve this task, querying each digital resource to determine its features. Thus, once the Producer has obtained from the Consumer the information on how to connect to the digital resources linked to the Consumer, it will query each new digital resource to determine its features, thereby producing a map.

The Context Discovery profile is implemented on the Consumer side and is one of the possible Actuator Profiles that the Consumer can support. Communication follows the OpenC2 standard, where a Producer sends a Command specifying that the Actuator to execute it is CTXD. If the Consumer implements CTXD, it will return a Response. The figure below summarizes this communication.

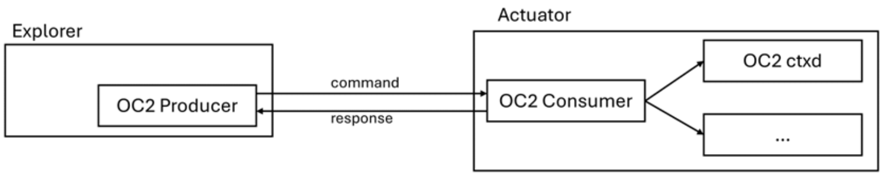


Fig. 5.1 Example of OpenC2 communication with CTXD

5.4 Data model

A data model is implemented to define which data the CTXD stores and the relationship between them. The most important data stored are:

- Service: it is the main class of the data model, and it describes the environment where the service is located, its links to other services, its subservices, the owner, the release, the security functions and the actuator.
- Service-Type: this class identifies the specific type of service. Each instance has its own parameters, and the Service has only one type. Examples: VM, Container, Cloud, etc.
- Link: this class describes the connection between the services. The field “peers” specifies the services that are on the other side of the link so, this class is useful for the recursive discovery. Also, security functions applied on the link are specified and they are described as OpenC2 Actuator Profiles.
- Consumer: It manages information about various services, including the security functions that protect them.
- OpenC2-Endpoint: they are described in the OpenC2-Endpoint class and correspond to both the OpenC2 Actuator Profile and the endpoint that implements it. A service can implement multiple security functions.
- Peer: this class describes the service that is connected to the service under analysis.

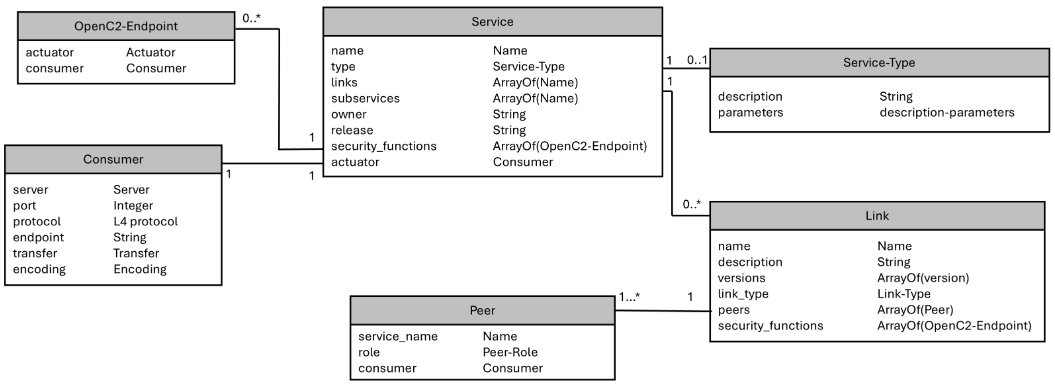


Fig. 5.2 CTXD data model

5.5 Command Components

This section identifies the applicable components of an OpenC2 Command. The components of an OpenC2 Command include:

- Action: list of Actions that are relevant for the CTXD. This Profile cannot define Actions that are not included in the OpenC2 Language Specification, but it may extend their definitions.
- Target: list of Targets included in the Language Specification [8] and one Target (and its associated Specifiers) that is defined only for the CTXD.
- Arguments: list of Command Arguments that are relevant for the CTXD.
- Actuator: list of Actuator Specifiers that are relevant for the CTXD.

5.5.1 Actions

Action is a mandatory field in Command message and no Actuator Profile can add a new Action that is not present in the specifications.

Type: Action (Enumerated)

ID	Name	Description
3	query	Initiate a request for information.

5.5.2 Target

Target is a mandatory field in Command message, and it is possible to define new Targets that are not present in the specifications. Only one Target is allowed in a Command, and that's why the cardinality of each one equals to 1.

Type: Target (Choice)

ID	Name	Type	#	Description
9	features	Features	1	A set of items used with the query Action to determine an Actuator's capabilities.
2048	context	Context	1	It describes the service environment, its connections and security capabilities.

A new target, called "context" is inserted because the Target "features" refers only to the Actuator capabilities and not to the characteristics of the execution environment.

5.5.3 Context

Type: Context (Record)

ID	Name	Type	#	Description
1	services	ArrayOf(Name)	0..1	List the service names that the command refers to.
2	links	ArrayOf(Name)	0..1	List the link names that the command refers to.

The Target Context is used when the Producer wants to know the information of all active services and links of the Consumer. The Producer can specify the names of the services and links it is interested in.

Usage requirements:

- A Producer may send a “query” Command with no fields to the Consumer, which could return a heartbeat to this command.
- A Producer may send a “query” Command containing an empty list of services. The Consumer should return all the services.
- A Producer may send a “query” Command containing an empty list of links. The Consumer should return all the links.
- A Producer may send a “query” Command containing an empty list of services and links. The Consumer should return all the services and links.

5.5.4 Command Arguments

Type: Args (Map)

ID	Name	Type	#	Description
4	response_requested	Response-Type	0..1	The type of Response required for the Command: none, ack, status, complete.
2048	name_only	Boolean	0..1	The response includes either only the name or all the details about the services and the links.

Command Arguments are optional, and a new one called “name_only” has been defined, which is not present in the Language Specification [8].

Usage requirements:

- The "response_requested": "complete" argument can be present in the "query features" Command. (Language specification 4.1 [8])
- The “query context” Command may include the "response_requested": "complete" Argument.

- The “query context” command may include the “name_only” argument:
  - If TRUE, the Consumer must send a Response containing only the names of the services and/or links.
  - If FALSE, the Consumer must send a Response containing all the details of the services and/or links.

5.5.5 Actuator Specifiers

List of Actuators Specifiers that are applicable to the Actuator. This is an optional field. These specifiers are not present in the Language Specification [8].

Type: Specifiers (Map)

ID	**Name**	**Type**	**#**	**Description**
1	domain	String	0..1	Domain under the responsability of the actuator
2	asset_id	String	0..1	Identifier of the actuator

5.6 Response Components

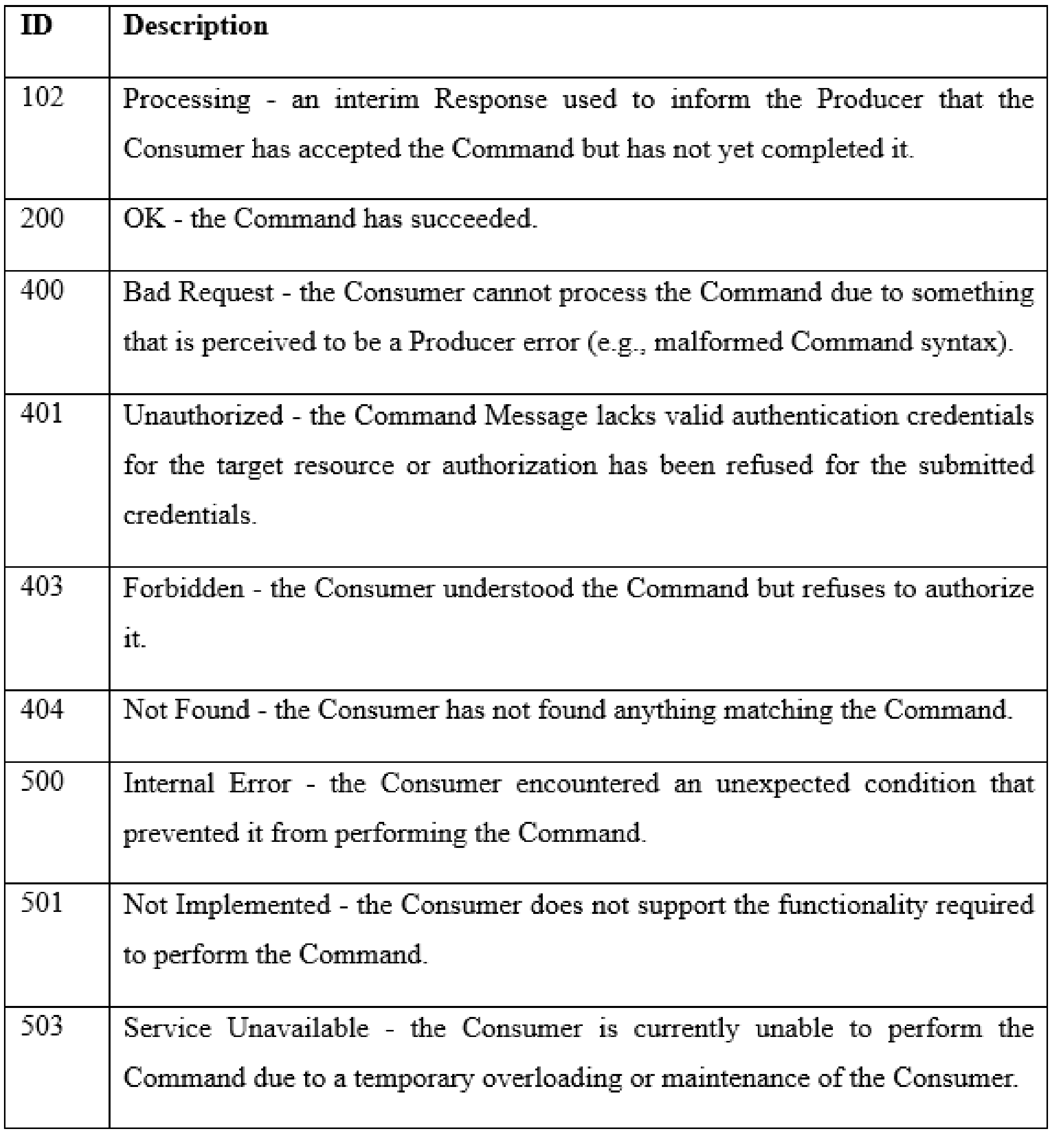
This section defines the Response Components relevant to the CTXD Actuator Profile. The table below outlines the fields that constitute an OpenC2 Response.

Type: OpenC2-Response (Map)

ID	**Name**	**Type**	**#**	**Description**
1	status	Status-Code	1	status code
2	status_text	String	1	description of the Response status
3	results	Results	1	results derived from the executed Command

5.6.1 Response status code

Type: Status-Code (Enumerated.ID)



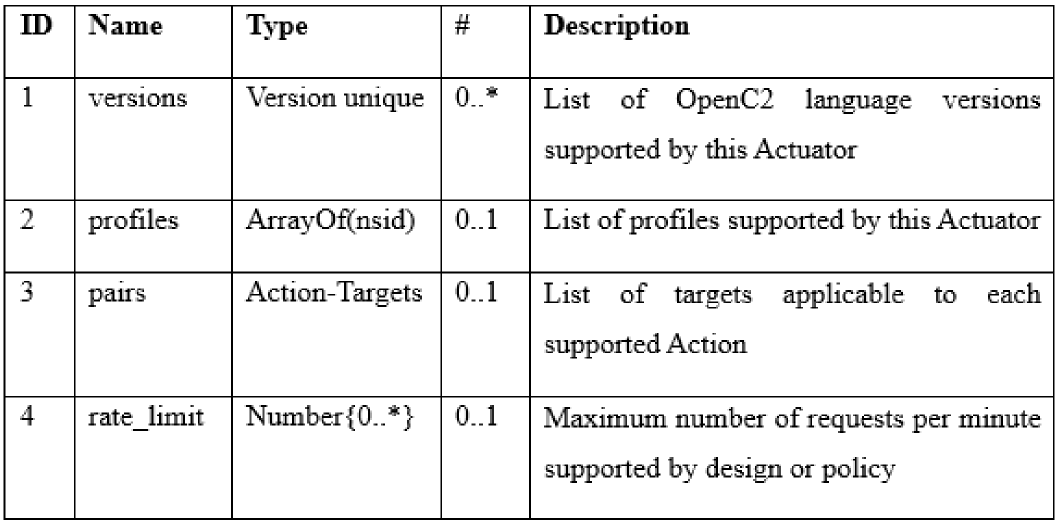
ID	Description
102	Processing - an interim Response used to inform the Producer that the Consumer has accepted the Command but has not yet completed it.
200	OK - the Command has succeeded.
400	Bad Request - the Consumer cannot process the Command due to something that is perceived to be a Producer error (e.g., malformed Command syntax).
401	Unauthorized - the Command Message lacks valid authentication credentials for the target resource or authorization has been refused for the submitted credentials.
403	Forbidden - the Consumer understood the Command but refuses to authorize it.
404	Not Found - the Consumer has not found anything matching the Command.
500	Internal Error - the Consumer encountered an unexpected condition that prevented it from performing the Command.
501	Not Implemented - the Consumer does not support the functionality required to perform the Command.
503	Service Unavailable - the Consumer is currently unable to perform the Command due to a temporary overloading or maintenance of the Consumer.

Fig. 5.3 Response status code [8]

5.6.2 Common Results

This section refers to the Results that are meaningful in the context of a CTXD and that are listed in the Language Specification [8].

Type: Results (Record0..*)



ID	Name	Type	#	Description
1	versions	Version unique	0..*	List of OpenC2 language versions supported by this Actuator
2	profiles	ArrayOf(nsid)	0..1	List of profiles supported by this Actuator
3	pairs	Action-Targets	0..1	List of targets applicable to each supported Action
4	rate_limit	Number{0..*}	0..1	Maximum number of requests per minute supported by design or policy

Fig. 5.4 Common results [8]

5.6.3 CTXD Results

These results are not included in the Language Specification [8] and are introduced specifically for the CTXD Actuator Profile.

Type: Results (Map0..*)

ID	Name	Type	#	Description
2048	services	ArrayOf(Service)	0..1	List all the services
2049	links	ArrayOf(Link)	0..1	List all the links of the services
2050	services_names	ArrayOf(Name)	0..1	List the names of all services
2051	link_names	ArrayOf(Name)	0..1	List the names of all services

Usage requirements:

- The response “services” can only be used when the target is “context”.
- The response “links” can only be used when the target is “context”.
- The response “services_names” can only be used when the target is “context”.

- The response “services_names” can only be used when the target is “context”.
- service_names/link_names are mutually exclusive with services/links, respectively. The choice is based on the value of the “name_only” argument in the query.

5.7 CTXD data types

With the introduction of new data types that are not specified in the original specifications, it is necessary to define these types along with their attributes, base type and eventually the conformance clauses. In this section, each new data type is defined, and for some, a use case example is provided.

5.7.1 Name

The Name type is used to indicate the name of any object. When the Command Argument is “name_only”, an array of Name is returned to the Producer.

Type: Name (Choice)

ID	Name	Type	#	Description
1	uri	URI	1	Uniform Resource Identifier of the service
2	reverse_dns	Hostname	1	Reverse domain name notation
3	uuid	UUID	1	Universally unique identifier of the service
4	local	String	1	Name without guarantee of uniqueness

5.7.2 Operating System (OS)

It describes an Operating System.

type: OS (Record)

ID	Name	Type	#	Description
1	name	String	1	Name of the OS
2	version	String	1	Version of the OS
3	family	String	1	Family of the OS
4	type	String	1	Type of the OS

5.7.3 Service

Digital resources can implement one or more services, with each service described by a Service type. This type is a key element of the data model, as it provides the information the Producer is seeking about the services.

type: Service (Record)

ID	Name	Type	#	Description
1	name	Name	1	Id of the service
2	type	Service-Type	1	It identifies the type of the service
3	links	ArrayOf(Name)	0..1	Links associated with the service
4	subservices	ArrayOf(Name)	0..1	Subservices of the main service
5	owner	String	0..1	Owner of the service
6	release	String	0..1	Release version of the service
7	security_functions	ArrayOf(OpenC2-Endpoint)	0..1	Actuator Profiles associated with the service
8	actuator	Consumer	1	It identifies who is carrying out the service

5.7.4 Service-Type

It represents the type of service, where each service type is further defined with additional information that provides a more detailed description of the service's characteristics.

Type: Service-Type (Choice)

ID	Name	Type	#	Description
1	application	Application	1	Software application
2	vm	VM	1	Virtual Machine
3	container	Container	1	Container
4	web_service	Web-Service	1	Web service
5	cloud	Cloud	1	Cloud
6	network	Network	1	Connectivity service
7	iot	IOT	1	IOT device

5.7.5 Application

It describes a generic application.

Type: Application (Record)

ID	Name	Type	#	Description
1	description	string	1	Generic description of the application
2	name	String	1	Name of the application
3	version	string	1	Version of the application
4	owner	string	1	Owner of the application
5	type	String	1	Type of the application

Sample Application object represented in JSON Format:

```
{
    ‘‘description’’: ‘‘application’’,
    ‘‘name’’: ‘‘iptables’’,
    ‘‘version’’: ‘‘1.8.10’’,
    ‘‘owner’’: ‘‘Netfilter’’,
    ‘‘type’’: ‘‘Packet Filtering’’
}
```

5.7.6 VM

It describes a Virtual Machine.

Type: VM (Record)

ID	Name	Type	#	Description
1	description	String	1	Generic description of the VM
2	id	String	1	ID of the VM
3	hostname	Hostname	1	Hostname of the VM
4	os	OS	1	Operating System of the VM

Sample VM object represented in JSON Format:

```
{
    ‘‘description’’: ‘‘vm’’,
    ‘‘id’’: ‘‘123456’’,
    ‘‘hostname’’: ‘‘My-virtualbox’’,
    ‘‘os’’: {
        ‘‘name’’: ‘‘ubuntu’’,
        ‘‘version’’: ‘‘22.04.3’’,
        ‘‘family’’: ‘‘debian’’,
        ‘‘type’’: ‘‘linux’’
    }
}
```

5.7.7 Container

It describes a generic Container.

Type: Container (Record)

ID	Name	Type	#	Description
1	description	String	1	Generic description of the container
2	id	String	1	ID of the Container
3	hostname	Hostname	1	Hostname of the Container
4	runtime	String	1	Runtime managing the Container
5	os	OS	1	Operating System of the Container

Sample Container object represented in JSON Format:

```
{
    “description”: “container”,
    “id”: “123456”,
    “hostname”: “container_name”,
    “runtime”: “docker”,
    “os”: {
        “name”: “ubuntu”,
        “version”: “22.04.3”,
        “family”: “debian”,
        “type”: “linux”
    }
}
```

5.7.8 Web Service

It describes a generic web service.

Type: Web Service (Record)

ID	Name	Type	#	Description
1	description	String	1	Generic description of the web service
2	server	Server	1	Hostname or IP address of the server
3	port	Integer	1	The port used to connect to the web service
4	endpoint	String	1	The endpoint used to connect to the web service
5	owner	String	1	Owner of the web service

Sample Web Service object represented in JSON Format:

```
{
    “description”: “web_service”,
    ‘‘server’’: ‘‘192.168.0.1’’,
    ‘‘port’’: 443,
    ‘‘endpoint’’: ‘‘maps/api/geocode/json’’,
    ‘‘owner’’: ‘‘Google’’
}
```

5.7.9 Cloud

It describes a generic Cloud service.

Type: Cloud (Record)

ID	Name	Type	#	Description
1	description	String	1	Generic description of the cloud service
2	id	String	1	Id of the cloud provider
3	name	String	1	Name of the cloud provider
4	type	String	1	Type of the cloud service

Sample Cloud object represented in JSON Format:

```
{
    ‘‘description’’: ‘‘cloud’’,
    ‘‘cloud_id’’: ‘‘123456’’,
    ‘‘name’’: ‘‘aws’’,
    ‘‘type’’: ‘‘lambda’’
}
```

5.7.10 Network

It describes a generic network service. The Network-Type is described in the following sections.

Type: Network (Record)

ID	Name	Type	#	Description
1	description	String	1	Generic description of the network
2	name	String	1	Name of the network provider
3	type	Network-Type	1	Type of the network service

Sample Network object represented in JSON Format:

```
{
    ‘‘description’’: ‘‘network’’,
    ‘‘name’’: ‘‘The Things Network’’,
    ‘‘type’’: ‘‘LoRaWAN’’
}
```

5.7.11 IOT

It describes an IoT device.

Type: IOT (Record)

ID	Name	Type	#	Description
1	description	String	1	Identifier of the IoT function
2	name	String	1	Name of the IoT service provider
3	type	String	1	Type of the IoT device

Sample IOT object represented in JSON Format:

```
{
    ‘‘description’’: ‘‘IoT’’,
    ‘‘name’’: ‘‘Azure IoT’’,
    ‘‘type’’: ‘‘sensor’’
}
```

5.7.12 Network-Type

This class describes the type of the network service. The details of these types are not further elaborated upon in this document.

Type: Network-Type (Choice)

ID	**Name**	**Type**	**#**	**Description**
1	ethernet	Ethernet	1	The network type is Ethernet
2	802.11	802.11	1	The network type is 802.11
3	802.15	802.15	1	The network type is 802.15
4	zigbee	Zigbee	1	The network type is Zigbee
5	vlan	Vlan	1	The network type is VLAN
6	vpn	Vpn	1	The network type is VPN
7	lorawan	Lorawan	1	The network type is LoRaWAN
8	wan	Wan	1	The network type is WAN

5.7.13 Link

A Service can be connected to one or more Services, the module Link describes the type of the connection, and the security features applied on the link.

Type: Link (Record)

ID	Name	Type	#	Description
1	name	Name	1	Id of the link
2	description	String	0..1	Generic description of the relationship
3	versions	ArrayOf(version)	0..1	Subset of service features used in this relationship (e.g., version of an API or network protocol)
4	link_type	Link-Type	1	Type of the link
5	peers	ArrayOf(Peer)	1	Services connected on the link
6	security_functions	ArrayOf(OpenC2-Endpoint)	0..1	Security functions applied on the link

5.7.14 Peers

The Peer object is useful for iteratively discovering the services connected on the other side of the link, enabling the Producer to build a map of the entire network.

Type: Peer (Record)

ID	Name	Type	#	Description
1	service_name	Name	1	Id of the service
2	role	Peer-Role	1	Role of this peer in the link
3	consumer	Consumer	1	Consumer connected on the other side of the link

5.7.15 Link-Type

This data type describes the type of the link between the peer and the service under analysis.

Type: Link-Type (Enumerated)

ID	Name	Type	#	Description
1	api	API	1	The connection is an API
2	hosting	Hosting	1	The service is hosted in an infrastructure
3	packet_flow	Packet-Flow	1	Network flow
4	control	Control	1	The service controls another resource

The types of API, Hosting, Packet-Flow, and Control are not defined in this document.

5.7.16 Peer-Role

It defines the role of the Peer in the link with the service under analysis.

Type: Peer-Role (Enumerated)

ID	Name	Description
1	client	The consumer operates as a client in the client-server model in this link
2	server	The consumer operates as a server in the client-server model in this link
3	guest	The service is hosted within another service.
4	host	The service hosts another service
5	ingress	Ingress communication
6	egress	Egress communication
7	bidirectional	Both ingress and egress communication
8	control	The service controls another service
9	controlled	The service is controlled by another service

5.7.17 Consumer

The Consumer provides all the networking parameters to connect to an OpenC2 Consumer.

Type: Consumer (Record)

ID	Name	Type	#	Description
1	server	Server	1	Hostname or IP address of the server
2	port	Integer	1	Port used to connect to the actuator
3	protocol	L4-Protocol	1	Protocol used to connect to the actuator
4	endpoint	String	1	Path to the endpoint (e.g., /.well-known/openc2)
5	transfer	Transfer	1	Transfer protocol used to connect to the actuator
6	encoding	Encoding	1	Encoding format used to connect to the actuator

5.7.18 Server

It specifies the hostname or the IPv4 address of a server.

Type: Server (Choice)

ID	Name	Type	#	Description
1	hostname	hostname	1	Hostname of the server
2	ipv4-addr	IPv4-Addr	1	32-bit IPv4 address as defined in [RFC0791]

5.7.19 Transfer

This data type defines the transfer protocol. This list can be extended with other transfer protocols.

Type: Transfer (Enumerated)

ID	Name	Description
1	http	HTTP protocol
2	https	HTTPS protocol
3	mqtt	MQTT protocol

5.7.20 Encoding

This data type defines the encoding format to be used. Other encodings are permitted, the type Encoding can be extended with other encoders (e.g., XML).

Type: Encoding (Enumerated)

ID	Name	Description
1	json	JSON encoding

5.7.21 OpenC2-Endpoint

This data type corresponds to both the OpenC2 Actuator Profile and the endpoint that implements it.

Type: OpenC2-Endpoint (Record)

ID	Name	Type	#	Description
1	actuator	Actuator	1	It specifies the Actuator Profile
2	consumer	Consumer	1	It specifies the Consumer that implements the security functions

"Actuator" type is described in Language Specification (section 3.3.1.3) [8].

5.8 Communication example

One helpful way to make clear the capabilities required for the CTXD profile is to create sample OpenC2 Command and Response messages. In this section, an example of an OpenC2 Communication is provided:

- First, the Producer requests the Actuator Profiles supported by the Consumer:

  Command:

```
{
    ‘‘action’’: ’’query’’,
    ‘‘target’’: {
        “features” : [“profiles”]
    }
}
```

Response:

```
{
    “status”:200,
    “results”:{
        “profiles: [“ctxd", “slpf”]
    }
}
```

- The Consumer implements the CTXD Actuator Profile so, it is possible to query about the services of the Actuator using an empty list. The result contains only the names of the services because the “name_only” argument is set to TRUE in the command.

  Command:

```
{
    "action": "query",
    "target": {
        "context": {
            "services": []
        }
    },
    "args": {
        "name_only": "TRUE"
    },
    "actuator": {
        "ctxd": {}
    }
}
```

  Response:

```
{
    "status": 200,
    "results": {
        "services": [
            {
                "name": {
                    "local": "example_service"
                }
            }
        ]
    }
}
```

- Now, a query for the service “example_service” is performed, requesting all details. In this case a VM is shown with SLPF security functionality. The service is connected to other services, the fields “links” is useful for OpenC2 discovery.

  Command:

```
{
    "action": "query",
    "target": {
        "context": {
            "services": [
                {
                    "name": {
                        "local": "example_service"
                    }
                }
            ]
        }
    },
    "args": {
        "name_only": "FALSE"
    },
    "actuator": {
        "ctxd": {}
    }
}
```

Response:

```
{
    "status": 200,
    "results": {
        "services": [
            {
                "name": {
                    "local": "example_service"
                },
                "type": {
                    "description": "vm",
                    "id": "123456",
                    "hostname": "My-virtualbox",
                    "os": {
                        "name": "ubuntu",
                        "version": "22.04.3",
                        "family": "debian",
                        "type": "linux"
                    }
                },
                "links": [
                    {
                        "name": {
                            "local": "link_1"
                        }
                    }
                ],
                "subservices": [
                    {
                        "name": {
                            "local": "example_subservice"
                        }
                    }
                ],
                "owner": "service_owner",
                "release": "1.0",
                "security_functions": [
                    {
                        "actuator": "slpf",
                        "consumer": {
                            "server": {
                                "ipv4-addr": "192.168.0.2"
                            },
                            "port": 80,
                            "protocol": "tcp",
                            "endpoint": "/.well-known/openc2",
                            "transfer": "http",
                            "encoding": "json"
                        }
                    }
                ],
                "actuator": {
                    "server": {
                        "ipv4-addr": "192.168.0.2"
                    },
                    "port": 80,
                    "protocol": "tcp",
                    "endpoint": "/.well-known/openc2",
                    "transfer": "http",
                    "encoding": "json"
                }
            }
        ]
    }
}
```

Note that in this example, the Consumer is the same for the stateless packet filter and for the CTXD.

- To discover the services that are connected to the previous one, another query is performed using the name obtained from the links field. In this example, the connection between these two services is monitored by an IDS hosted on another server (a mock OpenC2 profile for IDS has been created).

  Command:

```
{
    "action": "query",
    "target": {
        "context": {
            "links": [
                {
                    "name": {
                        "local": "link_1"
                    }
                }
            ]
        }
    },
    "args": {
        "name_only": "FALSE"
    },
    "actuator": {
        "ctxd": {}
    }
}
```

  Response:

```
{
    "status": 200,
    "results": {
        "link": [
            {
                "name": {
                    "local": "link_1"
                },
                "description": "example-generic-description",
                "versions": [
                    "1.0"
                ],
                "link_type": {},
                "peers": [
                    {
                        "service_name": {
                            "reverse-dns": "com.example-
                                connected-consumer"
                        },
                        "role": "host",
                        "consumer": {
                            "server": {
                                "ipv4_addr": "192.168.0.3"
                            },
                            "port": 80,
                            "protocol": "tcp",
                            "endpoint": "/.well-known/openc2",
                            "transfer": "http",
                            "encoding": "json"
                        }
                    },
                    {
                        "service_name": {
                            "local": "example_service"
                        },
                        "role": "guest",
                        "consumer": {
                            "server": {
                                "ipv4_addr": "192.168.0.2"
```

```
                            },
                            "port": 80,
                            "protocol": "tcp",
                            "endpoint": "/.well-known/openc2",
                            "transfer": "http",
                            "encoding": "json"
                        }
                    }
                ],
                "security_functions": [
                    {
                        "actuator": "ids",
                        "consumer": {
                            "server": {
                                "ipv4_addr": "192.168.0.4"
                            },
                            "port": 80,
                            "protocol": "tcp",
                            "endpoint": "/.well-known/openc2",
                            "transfer": "http",
                            "encoding": "json"
                        }
                    }
                ]
            }
        ]
    }
}
```

Chapter 6

CTXD implementation and validation in openc2lib

6.1 Structure of the CTXD Profile in openc2lib

The Context Discovery Actuator Profile can be easily added to the openc2lib library, which is designed to be extended with new profiles. A new subfolder named “ctxd” was added to the “profile” folder in the library, containing all the data types from Chapter 5 and the conformance clauses. This subfolder consists of the following elements:

- Profile: specifies the NameSpace Identifier (NSID) used to refer to the CTXD.
- Actuator: contains the specifiers that are meaningful in the context of the CTXD.
- Args: specifies the Command Argument “name_only” that can be used for the CTXD, extending the Command Arguments defined in the Language Specification [8].
- Results: extends the base Results specified in the Language Specification [8] by adding the new results specific to the CTXD, including "services", "links", "services_names" and "link_names".
- Validation: checks if the actions, targets, and arguments in the Command are valid.

- Target: this subfolder contains all the Targets introduced by the CTXD. The only new Target added, not present in the specifications, is "Context," which consists of two arrays of names corresponding to services and links.
- Data: This subfolder contains all the data types introduced in Chapter 5 that are not present in the specifications. Each new data type is implemented as a separate class and can inherit the properties and methods of the base types defined in the Language Specification [8].

6.2 Behaviour of the CTXD Profile in openc2lib

While the structure of the CTXD Profile is implemented in the “profile” folder, its behaviour is implemented in another folder named “actuator”. This separation enables the openc2lib library to support multiple implementations of the same Profile. The class CTXDActuator implements all the functions needed to create the correct OpenC2 Response and does not concern itself with the encoding formats. Starting from this class, child classes have been created which inherits all the functions to manage different Configuration Management System (CMS).

When a Command is received, this class checks if the pair of Action and Target is valid. Then, this class acts as a dispatcher, selecting the appropriate functions based on the Actuator field to execute the operation and create the correct Response.

The only Action allowed is *query*, which is implemented by a function of the same name. Based on the Target specified in the Command, this function dispatches the computation within the same class. If the specified Target is *features*, the corresponding function is selected. If the Target is *context*, a different function is chosen.

The *query_context* function within the CTXDActuator class defines the behaviour of the CTXD Actuator Profile and is called only when the Target is *context*. This function returns a list of Services and Links if the Command Argument *name_only* is false otherwise, returns a list of names corresponding to the Services and Links implemented by the Consumer. If no fields are specified, an empty response is returned, serving as a heartbeat to communicate to the Producer that the Consumer is active.

6.3 Implementation scope

Once the CTXD profile was completely added to the library, it was tested in a real case scenario. Given the extreme heterogeneity of execution environments, the first approach was to focus on the most dynamic environments, specifically cloud systems, which include both VMs and containers. For this, two of the most popular open-source CMS were considered: OpenStack and Kubernetes.

OpenStack is an open-source cloud computing platform that enables users to build and manage their own cloud infrastructure. It provides Infrastructure-as-a-Service (IaaS), which offers a framework for creating and controlling virtualized resources such as computing power, storage, and networking on demand.

Kubernetes is an open-source container orchestration system, and it can manage containerized applications across multiple hosts for deploying, monitoring, and scaling containers. Containers are portable units that include the code and everything the application needs to run.

The goal of CTXD is to discover the relationships between OpenStack, Kubernetes, cloud-based virtual machines (VMs), and containers. Additionally, the types of connections between these resources are documented. Further extensions can be implemented in the future to support additional CMS platforms, such as Docker Compose, Azure, Google Cloud, VMware, and others.

6.4 OpenStack discovery process

OpenStack is modelled as a Cloud Service in the CTXD, and, in this way, it is able to describe itself and its links to other Services so, to provide this additional information, a new Actuator called CTXDActuator_openstack is created. This new Actuator inherits all the functions from the CTXDActuator. The discovering of the services implemented by OpenStack is performed with the following command:

```
openstack service list -f json
```

The result is a JSON object containing all the services implemented by OpenStack (e.g., Nova), which is then parsed and reformatted according to CTXD data structures.

Then, to obtain the active servers (cloud-based VMs), it is necessary to run the following command:

```
openstack server list --status ACTIVE -f json
```

6.5 Kubernetes discovery process

A new Actuator, called CTXDActuator_kubernetes, starts another discovery process to retrieve information about the Kubernetes implementation. First, the Kubernetes Service is described as a Cloud and its nodes are obtained with the following command:

```
kubectl get nodes -o json
```

In this case, the nodes refer to the cloud-based VMs, which are the same as those in the OpenStack Cloud. Additionally, the namespaces implemented by Kubernetes are retrieved using a command, where the specific namespace is passed as a parameter:

```
kubectl get namespace <namespace_name> -o json
```

Next, the Actuator continues the discovery process to identify the relationships between nodes and pods, which can be, for example, VMs and containers, using the following command. The command takes the namespace and VM ID as parameters:

```
kubectl get pods -n <namespace_name>
--field-selector spec.nodeName=<vm_name>
-o json
```

Finally, each container is associated with its namespace via a link of the type "Packet flow".

When implementing the openc2lib Consumer, it is possible to specify the Kubernetes namespaces to which the VMs and containers belong. These namespaces can be listed in an array within a configuration file, after which the process can be initiated. If no namespace is specified in the configuration file, the discovery process will not filter any namespace and will be performed across all available namespaces.

6.6 Consumer implementation

The Consumer implementing the CTXD Actuator Profile must instantiate its Actuators. To do so, the CTXDActuator class is added to a dictionary, where the key is a tuple: the first element indicates the nsid of the CTXD Actuator Profile, and the second one specifies the hostname of the Service (it is possible to specify also the IPv4 address). The value is an Actuator that is initialized with its services, links, domain name and asset id. In the example below, an Actuator for the VM Service is shown:

```
actuators[(ctxd.Profile.nsid,str(vm.consumer.server.obj._hostname))] =
CTXDActuator(services= self.get_vm_service(
    vm.consumer.server.obj._hostname),
    links= self.get_vm_links(vm.consumer.server.obj._hostname),
    domain=None,
    asset_id=str(vm.consumer.server.obj._hostname))
```

Once all the Actuators are defined, the openc2lib Consumer class can be initialized. It is possible to define a Consumer that implements multiple actuators or one Consumer for each actuator.

6.7 Producer implementation

To correctly implement the discovery process, the Command Argument *name_only* is set to False to retrieve all information about the Service. The target is the *Context*, which consists of two empty *ArrayOf(Name)* to obtain all the data about all Services and Links of the Consumer. Additionally, the Actuator field is populated with the CTXD Actuator Profile.

The openc2lib Producer sends a Command to the Consumer that implements the OpenStack Cloud Service. Then, the VMs are identified and only the master node, that has the role of control-plane, hosts the Kubernetes Cloud Service. From the Kubernetes Service the same VMs are identified again, followed by the containers hosted on them. Finally, the relationships between the containers and their respective namespaces are determined.

To implement this research, a recursive function was created, with controls to prevent infinite loops by avoiding already known links. A link is considered known if both the nodes and the link type have been previously identified. Node and edges are implemented with the Graphviz library to make a directed graph. In the graph, along with the hostnames, other parameters such as the IPv4 address and the type of service are also displayed. In the next sections, several graphs resulting from the discovery processes will be presented.

6.8 Validation

Once the structure and behaviour of the Context Discovery Actuator Profile are implemented in the openc2lib library, a validation process is required to ensure its correct operation.

The use case implemented is a web server deployed as a Kubernetes application within an OpenStack-based cloud infrastructure [13]. In Figure 6.1 it is shown the architecture of the cloud-native application used to test the CTXD, where the web server (Server-0) is connected to a container (UPF-0) and all the traffic of the web server is routed through this User Plane Function which acts as an intermediary between the server and other network functions. The communication between the clients and gNB-0 is outside the scope of the validation process.

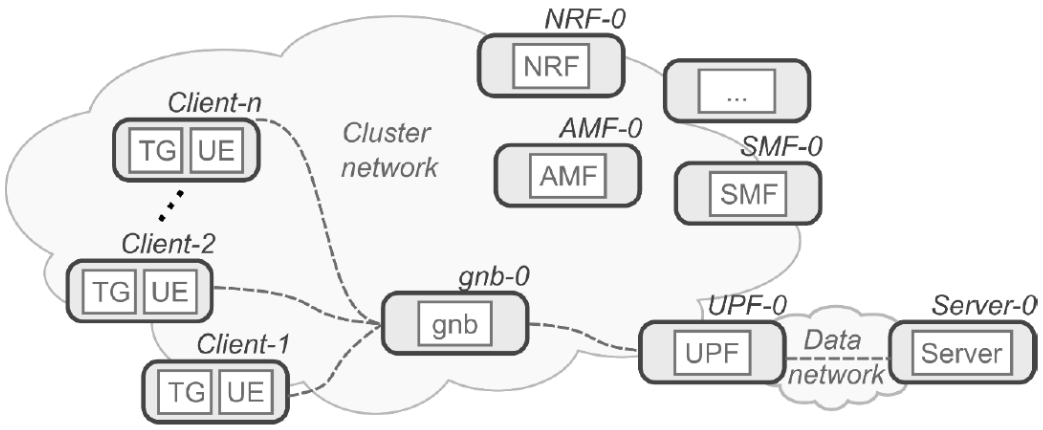


Fig. 6.1 5G network

The Kubernetes cluster is composed by three nodes (kube0, kube1, kube2), which are mapped to the virtual machines within OpenStack environment. Kube0 serves as the control-plane node and so it manages the Kubernetes cluster and schedules the workloads. Pods (UPF-0, AMF-0, NRF-0...) are dynamically scheduled to run on these Kubernetes nodes. The scope of the validation process is to identify the relationships between Kubernetes, its nodes and pods, and the OpenStack environment. Relationships between different pods are not considered.

Finally, the Consumer and Producer described in the previous sections will be used to perform the discovery process of the entire network. Note that all Consumers must be capable of implementing the CTXD Profile.

6.8.1 Functional test

Functional tests are conducted to verify the capabilities of the Consumer to implement the Context Discovery Actuator Profile and so, to parse and format data about its own services and connections in a way that is compatible with the CTXD data architecture and to send, in a correct way, to the Producer.

To verify the correct functioning of the software, the list of Kubernetes namespaces will be modified in the configuration file. This will determine the number of services discovered, depending on which namespaces are selected.

In the graph below, all Services and Links within the “my5gtestbed” namespace are visualized. For the VMs and Containers, their type and IPv4 address are also displayed. It’s important to highlight the relationship between Kubernetes and kube0, as without proper controls, it could lead to an endless loop: Kubernetes controls kube0, and kube0 hosts the Kubernetes Cloud Service.

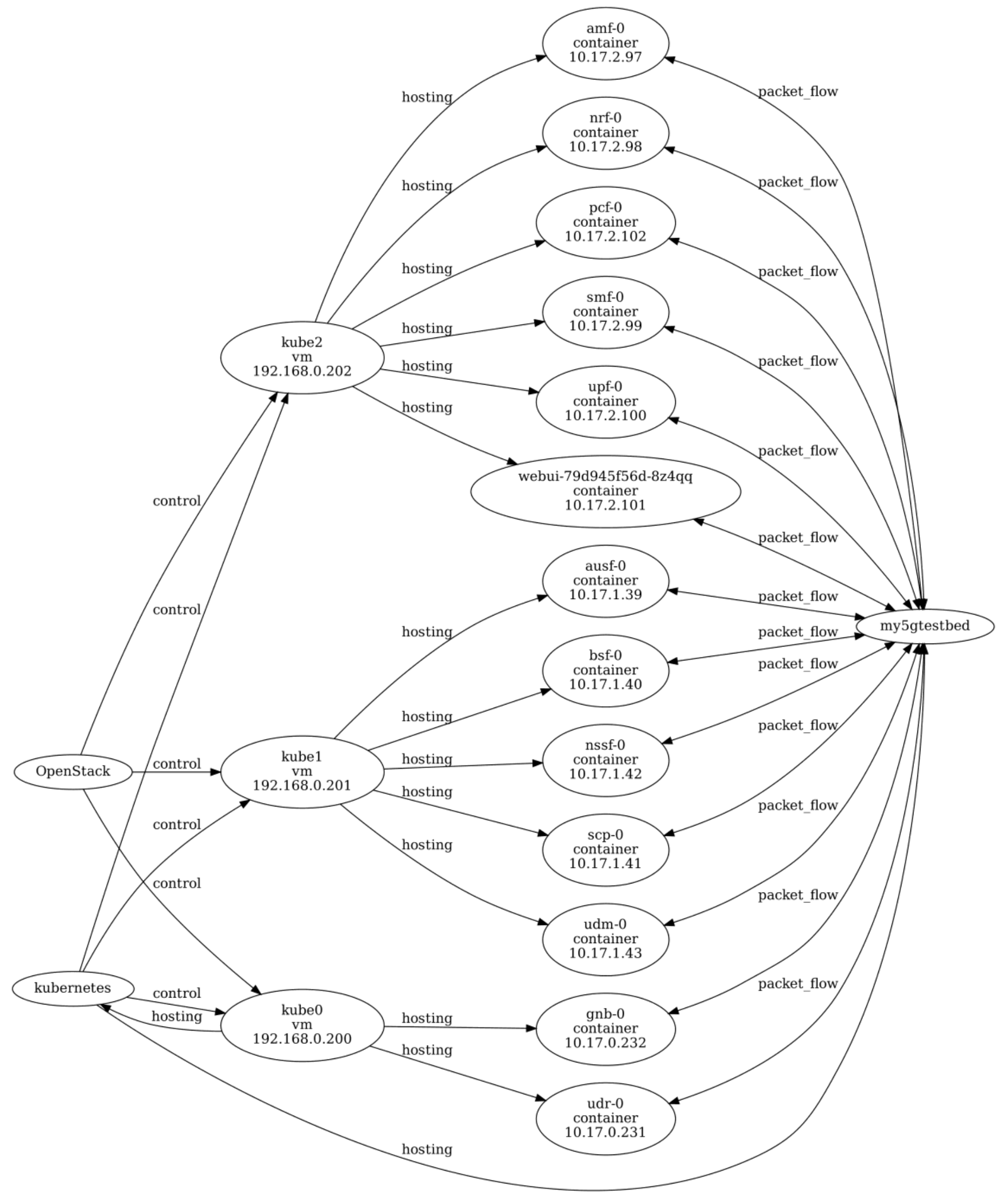

amf-0 container 10.17.2.97
nrf-0 container 10.17.2.98
pcf-0 container 10.17.2.102
smf-0 container 10.17.2.99
upf-0 container 10.17.2.100
webui-79d945f56d-8z4qq container 10.17.2.101
ausf-0 container 10.17.1.39
bsf-0 container 10.17.1.40
nssf-0 container 10.17.1.42
scp-0 container 10.17.1.41
udm-0 container 10.17.1.43
gnb-0 container 10.17.0.232
udr-0 container 10.17.0.231
kube2 vm 192.168.0.202
kube1 vm 192.168.0.201
kube0 vm 192.168.0.200
OpenStack
kubernetes
my5gtestbed
hosting
control
packet_flow

Fig. 6.2 directed graph of the my5gtestbed namespace

It is possible to select more than one namespace in the configuration file, as shown in the figure below, where the selected namespaces are “my5gtestbed” and “kube-flannel”:

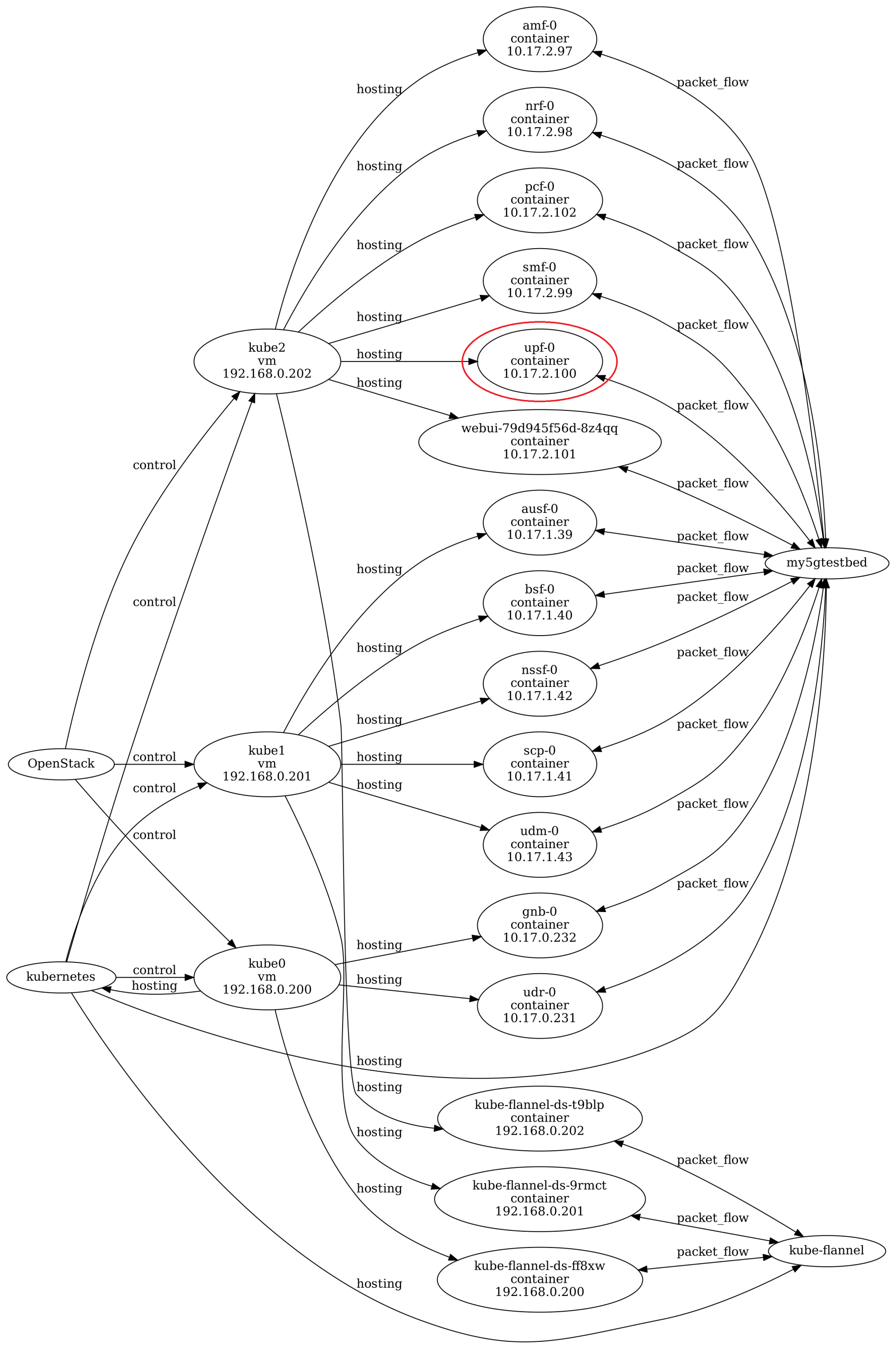

amf-0 container 10.17.2.97
nrf-0 container 10.17.2.98
pcf-0 container 10.17.2.102
smf-0 container 10.17.2.99
upf-0 container 10.17.2.100
webui-79d945f56d-8z4qq container 10.17.2.101
ausf-0 container 10.17.1.39
bsf-0 container 10.17.1.40
nssf-0 container 10.17.1.42
scp-0 container 10.17.1.41
udm-0 container 10.17.1.43
gnb-0 container 10.17.0.232
udr-0 container 10.17.0.231
kube-flannel-ds-t9blp container 192.168.0.202
kube-flannel-ds-9rmct container 192.168.0.201
kube-flannel-ds-ff8xw container 192.168.0.200
kube2 vm 192.168.0.202
kube1 vm 192.168.0.201
kube0 vm 192.168.0.200
OpenStack
kubernetes
my5gtestbed
kube-flannel
hosting
control
packet_flow

Fig. 6.3 directed graph of the my5gtestbed and kube-flannel namespaces

From Figure 6.3, it is possible to note that the relationships between the cloud services (OpenStack and Kubernetes) with the VMs (kube0, kube1 and kube2) are not modified because they don't depend on a specific namespace. New containers are added belonging to the new namespace "kube-flannel" that in the Figure 6.2 were not present.

6.8.2 Dynamic tests

The dynamic tests simulate a failure of the whole Kubernetes service or an individual pod, with the goal of verifying that the discovery process can detect changes in certain variables, such as the IPv4 address, which is re-assigned each time a shutdown and a restart occur.

The first test conducted was to shut down a single container (upf-0) with the command line:

```
kubectl delete pod upf-0
```

After shutting it down, the scheduler will reactivate the container, and the result is shown in the figure below:

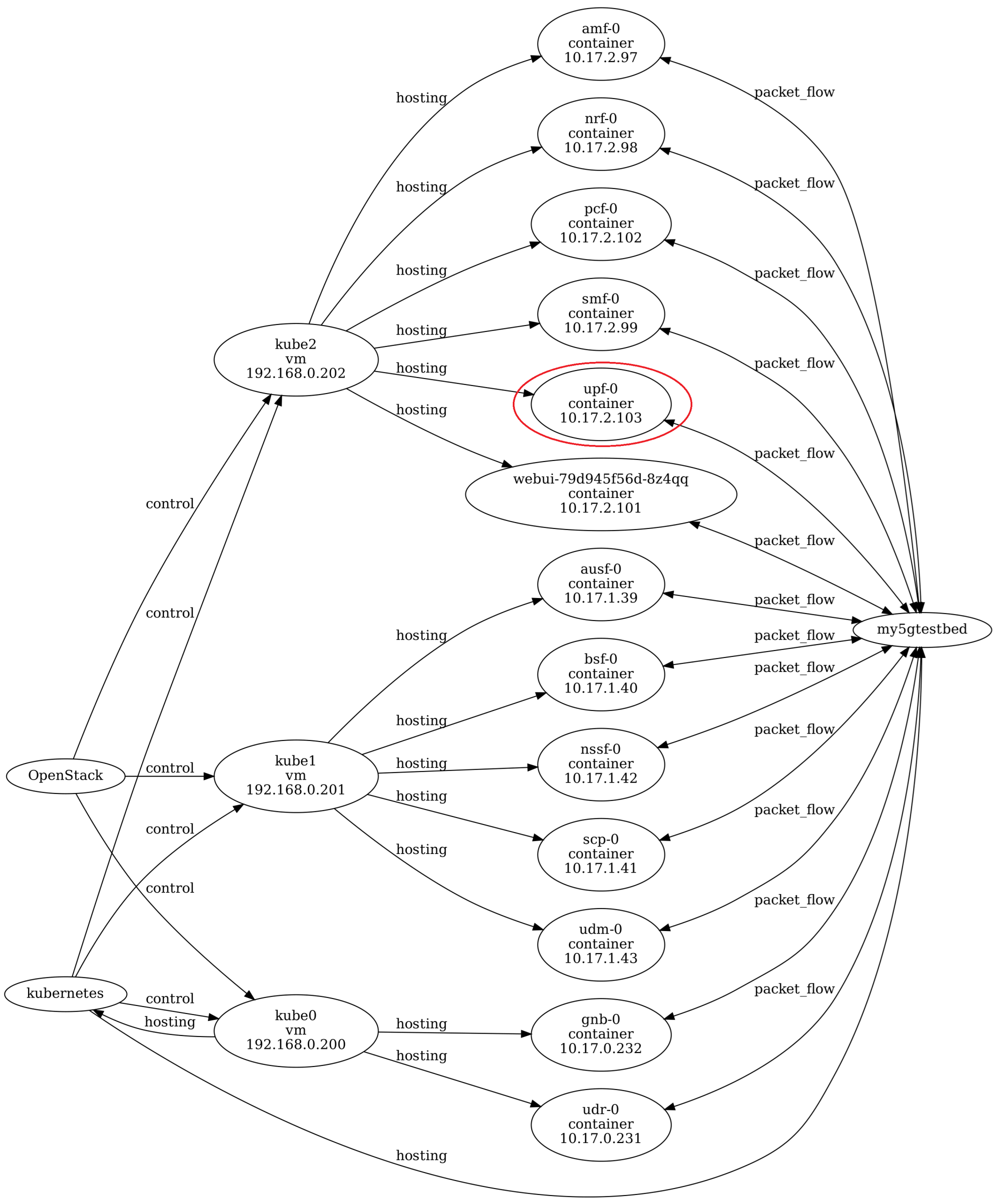

amf-0 container 10.17.2.97
nrf-0 container 10.17.2.98
pcf-0 container 10.17.2.102
smf-0 container 10.17.2.99
upf-0 container 10.17.2.103
webui-79d945f56d-8z4qq container 10.17.2.101
ausf-0 container 10.17.1.39
bsf-0 container 10.17.1.40
nssf-0 container 10.17.1.42
scp-0 container 10.17.1.41
udm-0 container 10.17.1.43
gnb-0 container 10.17.0.232
udr-0 container 10.17.0.231
kube2 vm 192.168.0.202
kube1 vm 192.168.0.201
kube0 vm 192.168.0.200
OpenStack
kubernetes
my5gtestbed
hosting
control
packet_flow

Fig. 6.4 directed graph of the my5gtestbed namespace after shutting down upf-0

The result is the same as the Figure 6.4 but with a substantial difference: the IPv4 address of the container upf-0 changed from 10.17.2.100 to 10.17.2.103.

Afterward, another test was performed, and the Kubernetes service was shut down and then reactivated:

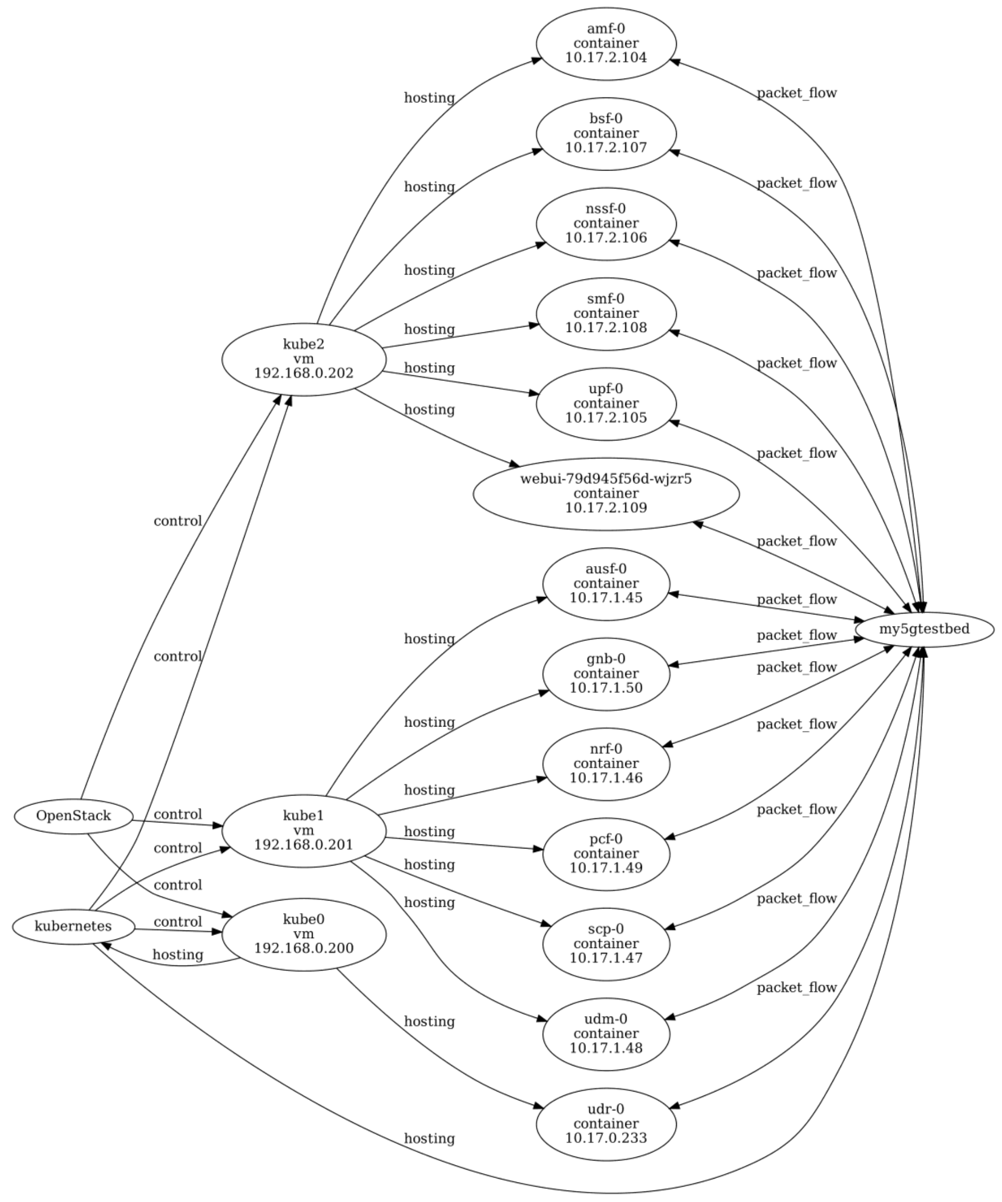

amf-0 container 10.17.2.104
bsf-0 container 10.17.2.107
nssf-0 container 10.17.2.106
smf-0 container 10.17.2.108
upf-0 container 10.17.2.105
webui-79d945f56d-wjzr5 container 10.17.2.109
ausf-0 container 10.17.1.45
gnb-0 container 10.17.1.50
nrf-0 container 10.17.1.46
pcf-0 container 10.17.1.49
scp-0 container 10.17.1.47
udm-0 container 10.17.1.48
udr-0 container 10.17.0.233
kube2 vm 192.168.0.202
kube1 vm 192.168.0.201
kube0 vm 192.168.0.200
OpenStack
kubernetes
my5gtestbed
hosting
packet_flow
control

Fig. 6.5 directed graph of the my5gtestbed namespace after shutting down Kubernetes service

It is possible to notice that the IPv4 addresses of each container and the VM where they are hosted have changed compared to what is shown in Figure 6.4. Finally, upf-0 was again shut down and then restarted:

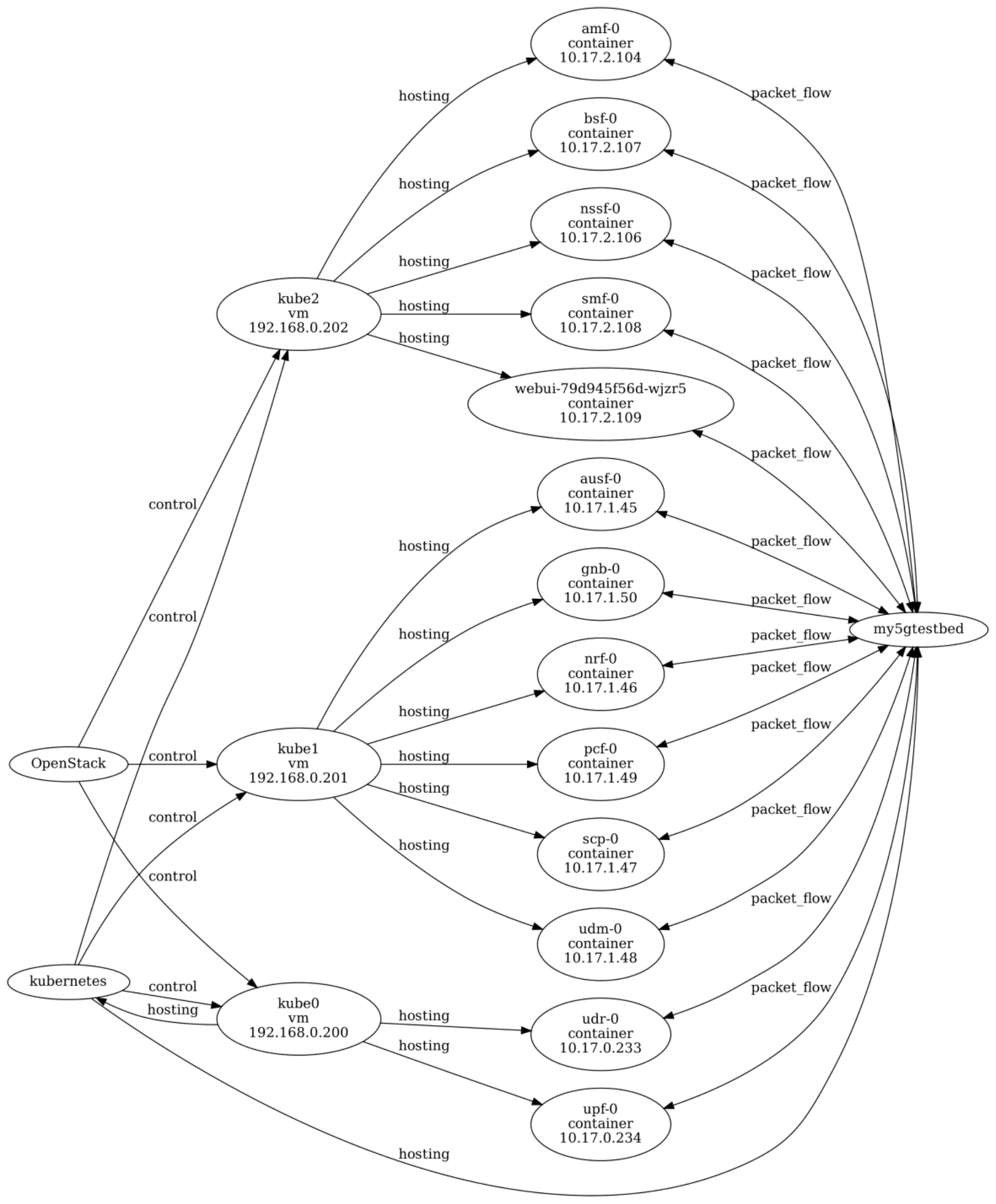


Fig. 6.6 directed graph of the my5gtestbed namespace after shuting down Kubernetes service and upf-0

The IPv4 address of the container upf-0 changed again, so the test was completed successfully.

To sum up all the dynamic tests done, all the IPv4 are memorized in the following table and the data are divided into 4 different phases.

	Before Shutting Down Kubernetes		**After Restarting Kubernetes**	
Name	before delete upf-0	after delete and restart upf-0	before delete upf-0	after delete and restart upf-0
amf-0	10.17.2.97	10.17.2.97	10.17.2.104	10.17.2.104
ausf-0	10.17.1.39	10.17.1.39	10.17.1.45	10.17.1.45
bsf-0	10.17.1.40	10.17.1.40	10.17.2.107	10.17.2.107
gnb-0	10.17.0.232	10.17.0.232	10.17.1.50	10.17.1.50
nrf-0	10.17.2.98	10.17.2.98	10.17.1.46	10.17.1.46
nssf-0	10.17.1.42	10.17.1.42	10.17.2.106	10.17.2.106
pcf-0	10.17.2.102	10.17.2.102	10.17.1.49	10.17.1.49
scp-0	10.17.1.41	10.17.1.41	10.17.1.47	10.17.1.47
smf-0	10.17.2.99	10.17.2.99	10.17.2.108	10.17.2.108
udm-0	10.17.1.43	10.17.1.43	10.17.1.48	10.17.1.48
udr-0	10.17.0.231	10.17.0.231	10.17.0.233	10.17.0.233
upf-0	10.17.2.100	10.17.2.103	10.17.2.105	10.17.0.234
webui-79d945f56d-8z4qq	10.17.2.101	10.17.2.101		
webui-79d945f56d-wjzr5			10.17.2.109	10.17.2.109

Table 6.1 Containers IPv4 addresses

6.8.3 Semantic tests

The final type of test conducted focused on the semantics of the new data introduced by the Context Discovery Actuator Profile, which are not included in the Language Specification [8]. The tests were performed using the Python library pytest, with the goal of providing both valid and invalid inputs to the class constructor.

Each data type introduced by the Context Discovery Actuator Profile was verified through up to three tests:

- Valid parameters: the class is tested with a valid set of parameters to ensure the object is created successfully.
- Invalid parameters: the class is tested with invalid inputs to verify that an exception is raised. If an exception is raised, the test is considered to have passed. Classes with only string attributes were not tested with invalid parameters.
- No parameters: the class is tested to ensure that it can be initialized without any parameters, verifying the default behaviour of the class constructor. Classes that are subclasses of the Enumerated and Choice classes were not tested.

The results of all the tests can be summarized in the following table:

Class	Valid parameters	Invalid parameters	No parameters
Application	Passed	Passed	Passed
Cloud	Passed	Passed	Passed
Consumer	Passed	Passed	Passed
Container	Passed		Passed
Encoding	Passed	Passed	
Iot	Passed	Passed	Passed
Link	Passed	Passed	Passed
LinkType	Passed	Passed	
Name	Passed	Passed	
Network	Passed		Passed
Openc2Endpoint	Passed	Passed	Passed
OS	Passed		Passed
Peer	Passed	Passed	Passed
PeerRole	Passed	Passed	
Server	Passed	Passed	
Service	Passed	Passed	Passed
ServiceType	Passed	Passed	
Transfer	Passed	Passed	
VM	Passed	Passed	Passed
WebService	Passed	Passed	Passed

Table 6.2 Semantic tests

Additionally, the JSON format of the Command and Response messages was tested. First, a JSON schema was created, and then the Command and Response messages were provided as input to verify that they conformed to this schema. The schema helps maintain interoperability and prevents errors caused by incorrect message formatting. Through unit tests, the schema's validation process is verified by decoding, encoding, and transferring valid and invalid OpenC2 messages. These tests ensure that the messages conform to the expected format and that any issues are caught early in the development cycle.

Chapter 7

Conclusions

Based on the OpenC2 specifications, a new Actuator Profile, called Context Discovery (CTXD), was defined and implemented. Defining a new Actuator Profile means extending the OpenC2 language to support new security capabilities that were previously not representable. The first step was to establish why the CTXD Profile was necessary and to demonstrate that the OpenC2 language, in its original form, could not address this specific need. The CTXD Profile was introduced because the Producer does not have complete knowledge of the services running in the network, and mechanisms were needed to enable the discovery of these services. To address this, the CTXD Profile abstracts the services running in the network, the interactions between them, and the security features that they implement, all in a way that is compatible with the OpenC2 language. The Profile also obtains from the service under analysis, essential information, such as hostname, encoding format, and transfer protocol, for connecting to it and to any linked services, which is crucial for implementing recursive discovery. Therefore, the CTXD Profile not only provides information about the service but also about the context in which it operates.

Once the structure was defined, the CTXD Profile was implemented in the openc2lib library and tested in a use case scenario. The testing environment was a Kubernetes application running within an OpenStack-based cloud infrastructure. The CTXD Profile successfully discovered the relationships between nodes, pods and namespaces. The main achievement was that the Producer only needed to know how to connect to the first Consumer, and by obtaining the information about its links, recursive discovery could take place, allowing the Producer to gather information about other services within the network.

OpenC2 language is expected to evolve in the future with new releases of the actual specifications and the addition of new Actuator Profiles. The openc2lib library is updated with the most recent specifications and with the Stateless Packet Filter Actuator Profile plus the addition of the Context Discovery Actuator Profile.

Future works can focus on the implementation of the CTXD in different use case scenarios and about further parameters, over those just defined, that can modify the data types already defined by the CTXD and can improve the discovery process.

Other future works can relate the maintenance of the entire library to keep up to date it with the specifications that will be defined in the future. Also, the library supports only the HTTP transfer protocol and the JSON encoder and so, the library can be extended with other transfers or encoders.

Bibliography

[1] Michael J Herring and Keith D Willett. Active cyber defense: a vision for real-time cyber defense. *Journal of Information Warfare*, 13(2):46–55, 2014.

[2] Manfred Vielberth, Fabian Böhm, Ines Fichtinger, and Günther Pernul. Security operations center: A systematic study and open challenges. *Ieee Access*, 8:227756–227779, 2020.

[3] Stef Schinagl, Keith Schoon, and Ronald Paans. A framework for designing a security operations centre (soc). In *2015 48th Hawaii International Conference on System Sciences*, pages 2253–2262. IEEE, 2015.

[4] Matteo Repetto, Alessandro Carrega, and Riccardo Rapuzzi. An architecture to manage security operations for digital service chains. *Future Generation Computer Systems*, 115:251–266, 2021.

[5] Faris Bugra Kokulu, Ananta Soneji, Tiffany Bao, Yan Shoshitaishvili, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Matched and mismatched socs: A qualitative study on security operations center issues. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1955–1970, 2019.

[6] Alessandro Carrega, Giovanni Grieco, Domenico Striccoli, Manos Papoutsakis, Tomas Lima, José Ignacio Carretero, and Matteo Repetto. A reference architecture for management of security operations in digital service chains. In *Cybersecurity of Digital Service Chains: Challenges, Methodologies, and Tools*, pages 1–31. Springer International Publishing Cham, 2022.

[7] Open Command and Control (OpenC2) Architecture Specification Version 1.0. Edited by Duncan Sparrell. 30 September 2022. OASIS Committee Specification 01. https://docs.oasis-open.org/openc2/oc2arch/v1.0/cs01/oc2archv1.0- cs01.html. Latest stage: https://docs.oasis-open.org/openc2/oc2arch/v1.0/oc2arch-v1.0.html.

[8] Open Command and Control (OpenC2) Language Specification Version 1.0. Edited by Jason Romano and Duncan Sparrell. 24 November 2019. OASIS Committee Specification 02. https://docs.oasis-open.org/openc2/oc2ls/v1.0/cs02/oc2ls-v1.0-cs02.html. Latest version: https://docs.oasis-open.org/openc2/oc2ls/v1.0/oc2ls-v1.0.html.

[9] OpenC2 Actuator Profile Development Process Version 1.0. Edited by David Lemire and David Kemp. 17 January 2024. OASIS Committee Note 01. https://docs.oasis-open.org/openc2/cn-appdev/v1.0/cn01/cn-appdev-v1.0-cn01.html. Latest stage: https://docs.oasis-open.org/openc2/cn-appdev/v1.0/cn-appdev-v1.0.html.

[10] Specification for Transfer of OpenC2 Messages via HTTPS Version 1.1. Edited by David Lemire. 30 November 2021. OASIS Committee Specification 01. https://docs.oasis-open.org/openc2/open-impl-https/v1.1/cs01/open-impl-https-v1.1-cs01.html. Latest stage: https://docs.oasis-open.org/openc2/open-impl-https/v1.1/open-impl-https-v1.1.html.

[11] Specification for Transfer of OpenC2 Messages via MQTT Version 1.0. Edited by David Lemire. 19 November 2021. OASIS Committee Specification 01. https://docs.oasis-open.org/openc2/transf-mqtt/v1.0/cs01/transf-mqtt-v1.0 cs01.html . Latest stage: https://docs.oasis-open.org/openc2/transf-mqtt/v1.0/transf-mqtt-v1.0.htmL.

[12] Open Command and Control (OpenC2) Profile for Stateless Packet Filtering Version 1.0. Edited by Joe Brule, Duncan Sparrell and Alex Everett. 11 July 2019. Committee Specification 01. https://docs.oasis-open.org/openc2/oc2slpf/v1.0/cs01/oc2slpfv1.0- cs01.html. Latest version: https://docs.oasis-open.org/openc2/oc2slpf/v1.0/oc2slpfv1.0. html.

[13] Matteo Repetto. Service templates to emulate network attacks in cloud-native 5g infrastructures. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 498–503. IEEE, 2023.