Politecnico di Torino

Master of Science in Computer Engineering

Master's Degree Thesis

Homogeneous control of stateless firewalls with OpenC2

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

Candidate:
Stefano Catenaro

Academic Year 2025-2026

Abstract

The growing complexity of modern network infrastructures and the heterogeneity of firewall and security technologies have made standardization and automation essential for effective cyber defense management. The Open Command and Control (OpenC2) standard, developed by OASIS, defines a unified language for commanding and controlling security components, enabling interoperability between heterogeneous systems through well-defined producer-consumer interactions. The Stateful Packet Filtering (SLPF) profile specifies how OpenC2 commands can be applied to network filtering systems, defining standardized actions, targets, arguments and results for managing firewalls and similar packet filtering technologies.

This thesis defines a fully-featured Actuator Manager that implements the Open Command and Control (OpenC2) Stateless Packet Filtering (SLPF) Actuator Profile for multiple firewall technologies: iptables, OpenStack Security Groups, Kubernetes Network Policies and Microsoft Azure Network Security Groups.
Each of the four selected firewall technologies has been implemented as a dedicated SLPF Actuator, managed by the SLPF Actuator Manager. Each actuator extends the Actuator Manager's functionality to support the specific firewall mechanisms of the corresponding system.
The work introduces a modular architecture that clearly separates the logic that depends on platform-specific implementations from the general OpenC2 SLPF management, allowing easier extension and integration of additional platforms in the future.

The implementation is validated through comprehensive testing, including syntactic and semantic verification, functional evaluation and performance measurements.
Finally, the thesis provides a critical assessment of the SLPF profile, evaluating its expressiveness and applicability across heterogeneous environments. The results confirm the feasibility of OpenC2 as a unified, interoperable and automatable control mechanism for diverse network protection systems.

Contents

List of figures

List of tables

1 Introduction

In an increasingly digital and interconnected world, safeguarding information systems against cyberattacks is of central importance. As cyber threats grow in speed, sophistication and automation, traditional security approaches are no longer sufficient. Attackers now can count on advanced technologies to launch highly targeted and rapid assaults, making it increasingly difficult to detect and mitigate threats in real time.
In this rapidly shifting landscape, defensive strategies must evolve accordingly, relying on intelligent tools, real-time responses and continuous efforts to safeguard digital infrastructures and sensitive information.
To achieve this, organizations often rely on a combination of security solutions from multiple vendors. While this multi-vendor approach can enhance protection by covering different layers of risk, it also introduces significant challenges such as poor interoperability, as products are not always designed to communicate natively with one another, fragmented visibility, which limits the ability to form a unified view of the security status, and increased operational complexity, as each solution may require its own configuration and maintenance procedures.
As the number of integrated tools grows, so does the difficulty of maintaining a coherent and responsive defense architecture: security workflows become harder to automate, incident response slows down and the risk of misconfigurations increases. In this context, the lack of standardized communication mechanisms among components becomes a critical bottleneck, limiting the efficiency and effectiveness of cyber defense operations.
This absence of native interoperability often pushes organizations to adopt proprietary languages, custom APIs or vendor-specific integration layers to enable communication between disparate components. While such solutions can fix compatibility gaps, they also add technical complexity, increase maintenance efforts and may lead to long-term vendor lock-in.
One way to overcome these challenges is through the adoption of standardized communication protocols. In this context, **OpenC2 (Open Command and Control)** offers a valid solution.

1.1 Open Command and Control (OpenC2)

OpenC2 is a standardized language developed by the OASIS consortium, designed to allow command and control of cyber defense components in a platform-agnostic and vendor-neutral manner. Its core objective is to provide a common syntax and structure for issuing security actions, such as blocking IPs, isolating devices or querying status, regardless of the specific technologies or vendors involved.
Unlike traditional approaches that depend on custom integrations or proprietary interfaces, OpenC2 separates *what to do* (the command) from *how to do it* (the implementation). This abstraction allows different cybersecurity components, such as firewalls or intrusion detection systems, to receive and understand commands in a consistent format, improving coordination and reducing the need for custom development.
By defining a machine-readable, extensible language for cyber defense operations, OpenC2 significantly reduces integration complexity, improves interoperability and allows faster and more coordinated responses to security events. This is really important in environments composed of many multi-vendor systems, where creating a unified control layer is often difficult and expensive. OpenC2 helps organizations move toward automation and orchestration, key parts of modern cybersecurity strategies, without being locked into a specific vendor ecosystem.
To address the wide variety of use cases in cybersecurity, OpenC2 is organized around **profiles,** specifications that define how the core language should be used with specific types of systems. These profiles standardize the syntax and parameters for sending commands to particular types of components, such as firewalls or threat detectors. By doing so, they ensure consistent interpretation and execution of OpenC2 commands across implementations.
Among the existing OpenC2 profiles, the **Stateless Packet Filtering (SLPF)** profile is particularly significant in operational environments.

1.2 Stateless Packet Filtering Profile (SLPF Profile)

The SLPF profile defines a standardized set of commands and target specifications for controlling packet filtering devices, such as firewalls and routers, without requiring stateful inspection. By creating a common language for managing access control rules across heterogeneous platforms, the SLPF profile addresses one of the most frequent and important tasks in network defense: the ability to quickly and precisely control traffic flows.
In multi-vendor environments, where firewalls and filtering components come from different manufacturers with their own command syntaxes and configuration models, the SLPF profile provides a unified abstraction layer. This abstraction divides the intent of a security action, such as blocking traffic from a malicious IP address, from the specific implementation details required by each individual system. As a result, organizations can dynamically apply or revoke filtering rules in a consistent and automated manner, regardless of the underlying technology.

This standardization not only simplifies policy enforcement across complex infrastructures, but also improves the agility of security operations. Instead of developing and maintaining custom scripts or vendor-specific integrations, OpenC2 commands can be sent through orchestration tools or threat response platforms, ensuring a faster and more uniform deployment of filtering actions.
Consequently, the operational overhead associated with managing diverse packet filtering systems is significantly reduced, while response times to emerging threats can be greatly improved. Furthermore, integrating the SLPF profile into existing security architectures contributes to reducing reliance on proprietary integrations, laying the foundation for more scalable and adaptable defense strategies.
In a landscape where rapid and coordinated response is essential, the ability to express and enforce filtering actions through a standardized, vendor-neutral interface is a substantial step toward interoperability and automation in cyber defense.

1.3 Thesis Structure

This section provides an overview of the structure of this thesis, outlining the contents and focus of each chapter.
Chapter 2 provides a comprehensive introduction to Open Command and Control (OpenC2), presenting its architecture, implementation considerations and profiles. In particular, the Stateless Packet Filtering (SLPF) Actuator Profile is analyzed in detail, including its actions, targets, arguments and result reporting mechanisms.
Chapter 3 introduces a set of firewall technologies to which the SLPF Profile can be applied, namely iptables, OpenStack Security Groups, Kubernetes Network Policies and Microsoft Azure Network Security Groups. Each technology is described in terms of its key functionalities, rule management capabilities and relevance within the context of the SLPF Profile.
Chapter 4 defines the objectives of this thesis, which are to design, implement and validate a fully-featured Actuator Manager that realizes the OpenC2 SLPF Actuator Profile for the firewall technologies discussed in Chapter 3.
Chapter 5 presents the design and implementation of the SLPF Actuator Manager, detailing how it manages the different firewall systems. The chapter also describes the development of backend-specific actuators responsible for executing SLPF commands within the respective firewall environments.
Chapter 6 focuses on the software libraries and frameworks utilized in the implementation of the Actuator Manager and the specific actuators. It also reports the validation and testing procedures carried out to assess the syntactic and semantic correctness, functional behavior and performance of the system.
Finally, Chapter 7 concludes the thesis by providing a critical analysis of the SLPF Profile based on the implementation experience, summarizing the main contributions and outlining possible directions for future work.

2 OpenC2

2.1 Introduction to OpenC2

In response to the increasing need for interoperability and automation in cyber defense, the **Open Command and Control (OpenC2)** standard has emerged as a promising solution. Developed by the OASIS OpenC2 Technical Committee, this standard defines a common, extensible language for issuing commands and interpreting responses across diverse cybersecurity components and platforms.
OpenC2 is designed to enable machine-to-machine communication between defense systems, regardless of vendor, architecture or deployment environment. By standardizing not only how actions are requested but also how outcomes are reported, it facilitates faster, more consistent and automated security operations. Its extensible design allows it to evolve alongside emerging technologies and support a wide variety of use cases, from simple packet filtering to complex multi-step threat mitigation workflows.
This chapter explores the core principles of OpenC2, including its architecture, message structure and operational model, and the SLPF Profile, laying the groundwork for understanding how it can be applied in real-world cybersecurity environments.

2.2 OpenC2 Architecture

At its core, the architecture of OpenC2 [1] is designed to facilitate decoupled and standardized communication between cybersecurity components. It follows a **producer–consumer model**, where commands are generated by a *producer* (e.g., an orchestrator or security automation platform) and received by one or more *consumers* (e.g., firewalls, endpoint protection systems or network devices) capable of executing the requested actions. On the receiving side, the consumer interprets the command, carries out the action on the specified target and responds with a message indicating the result, whether successful or failed.

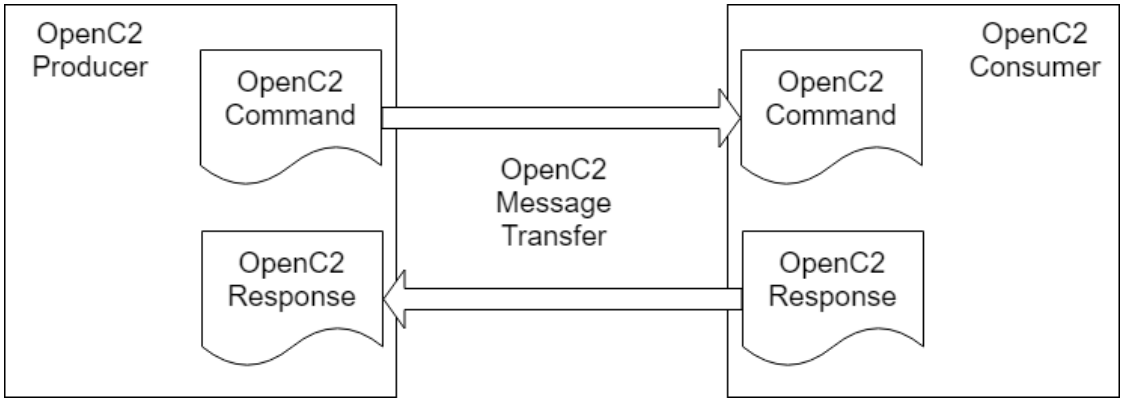

Fig. 2.1 OpenC2 communication model [1]

By separating the command generation from the execution logic, OpenC2 ensures that producers and consumers can evolve independently, as long as they adhere to the shared language and profiles. This abstraction also simplifies integration across heterogeneous environments, where tools may differ in purpose, implementation or vendor.
OpenC2 is transport-agnostic by design, meaning it does not hard-code a specific communication protocol into its architecture. However, to ensure reliable and interoperable exchanges between producers and consumers, the OpenC2 specification defines standardized transport bindings, such as for HTTPS and MQTT, that describe how messages should be formatted and transmitted over specific protocols.

In the OpenC2 architecture, a special type of consumer known as an **Actuator Manager** serves as an intermediary between the producer and a set of managed actuators. Unlike a regular consumer, which directly applies OpenC2 commands to the security functions it provides, an Actuator Manager acts as a translation interface: it receives an OpenC2 command from the producer and converts it into the appropriate device-specific control interface. These managed devices may be heterogeneous and may not expose an OpenC2-native interface, making the Actuator Manager responsible for abstracting their underlying protocols into a unified control plane. As a result, it enables producers to interact with a diverse collection of security tools through a consistent command model, improving interoperability, scalability and operational consistency. Depending on the deployment, an Actuator Manager may also act as a producer toward managed devices that support OpenC2 natively.

Each **command** in OpenC2 encapsulates an action that a producer wishes to perform on a specific target, such as "allow traffic", "deny connection" or "update file”. Its structure is designed to be concise and flexible, accommodating a wide range of use cases in cyber defense. A valid command consists of the following components:
- **Action** (required): what to do (e.g., allow, deny, update);
- **Target** (required): the object of the action (e.g., an IPv4 connection, a file);
- **Arguments** (optional): parameters that modify the behavior of the command (e.g., start/stop time, response requested);

- **Actuator** (optional): the type of system expected to carry out the action (e.g., a stateless packet filter, an endpoint).

This modular structure allows OpenC2 to support simple operations (e.g., blocking an IP address) as well as more complex tasks that may involve time constraints or multiple execution targets. The clear separation of subject (actuator), verb (action), object (target) and complements (arguments) also enhances readability and consistency across implementations.

The official OASIS OpenC2 Language Specification [2] defines the set of valid values for core command elements such as actions, targets and arguments. While the specification provides a standardized vocabulary to ensure interoperability, it also allows for extensibility, particularly in the definition of new actuators, allowing implementations to introduce custom types when required by specific operational contexts, as long as they remain consistent with the overall syntax and semantics of the language.

Once a consumer receives a command and processes it, it generates a **response** to indicate the result. This response follows a similarly structured grammar, enabling reliable feedback loops between producers and consumers. A typical response includes:

- **Status** (required): a numeric code indicating the outcome of the command execution (e.g., success, failure, pending);
- **Status Text** (optional): descriptive message providing additional context or explanation related to the status (e.g., error details, confirmation notes);
- **Results** (optional): field containing any data or information returned as a consequence of the command (e.g., query results).

This grammar ensures that consumers can provide both machine-readable and human-friendly information in their responses.

All OpenC2 messages are required to conform to the rules and constraints defined in the official specifications, which outline how the language must be used to ensure consistency, interoperability and correct behavior across compliant implementations.

The following example illustrates an OpenC2 command that issues an allow action on a connection target of type IPv6, specifying the destination address, protocol and source port, intended for an SLPF actuator.

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


Fig. 2.2 OpenC2 command example [1]

In response to the previously issued command, the consumer returns a message indicating successful execution with a status code 200. The response also includes a result field containing the rule number assigned to the newly created filtering rule. Within the SLPF profile, this rule number serves as a unique identifier, enabling the producer to manage the rule in future operations (such as revoking it).

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

Fig. 2.3 OpenC2 response example [1]

2.3 OpenC2 Implementation

To ensure interoperability across heterogeneous cybersecurity environments, OpenC2 adopts a layered protocol stack model, as specified in the OpenC2 Architecture Specification [1]. This model structures the specification suite into distinct levels, each addressing a different aspect of the communication process, from high-level intent expression to secure message transport.

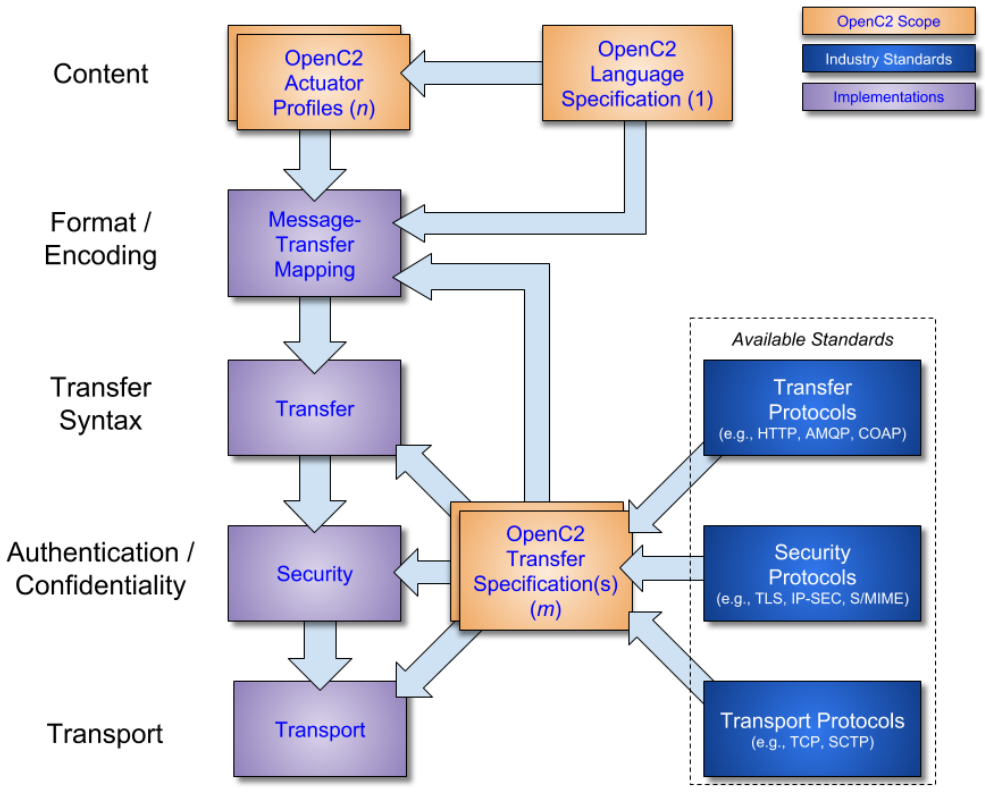

Fig. 2.4 OpenC2 Documentation and Layering Model [1]

At the top of the stack lie the Common Content layer and the Function-Specific Content layer, which together define the OpenC2 language and its specialized extensions.
The **Common Content** layer provides the core structure and semantics of the language, establishing a shared vocabulary for expressing commands and responses [2]. It defines the fundamental elements, such as actions, targets, arguments and actuators, that allow consistent communication across diverse systems. This layer serves as the foundation for interoperability, ensuring that all OpenC2 implementations can interpret and exchange messages in a uniform manner.
Building upon it, the **Function-Specific Content** layer introduces extensions designed for specific operational domains. This includes Actuator Profiles, which constrain and refine the common language to suit particular defensive capabilities, for example stateless packet filtering or endpoint control, by specifying the valid combinations of actions, targets and arguments relevant to each context [3].
Beneath the language layers lies the **Message/Transfer** layer, which bridges OpenC2 content with the underlying transport mechanisms. It defines how commands and responses are serialized (typically using JSON) and encapsulated into message constructs that include metadata such as message identifiers, timestamps and routing information.
Standardized transfer specifications exist for widely used protocols like HTTPS and MQTT, facilitating efficient and reliable message exchange between producers and consumers.
At the base of the stack is the **Secure Transport** layer, which ensures the actual delivery of messages between producers and consumers over secure and reliable communication protocols. Although OpenC2 itself remains transport-agnostic by design, this layer ensures confidentiality, integrity and interoperability across diverse implementations by defining specific transport bindings over chosen technologies. Examples of these technologies include TLS-based protocols such as HTTPS, which provides authenticated, ordered and lossless message delivery as specified in the Specification for Transfer of OpenC2 Messages via HTTPS [4]. Similarly, the Specification for Transfer of OpenC2 Messages via MQTT [5] defines message transfer over MQTT, a lightweight publish-subscribe protocol optimized for efficient and scalable communication in constrained environments. Beyond these, other established security mechanisms such as S/MIME (Secure/Multipurpose Internet Mail Extensions) and IPsec (Internet Protocol Security) can also be leveraged to secure OpenC2 message exchanges depending on the deployment context and security requirements.
In this modular and layered architecture, the language is designed to decouple the transport mechanism from its implementation, allowing flexibility in deployment and easier integration across diverse platforms. By adopting this structure, OpenC2 supports scalable, secure and standardized cyber defense automation across a wide range of operational environments.

2.4 OpenC2 Message Transport

OpenC2 supports message transfer over HTTPS, leveraging the well-established HTTP protocol secured by Transport Layer Security (TLS) to ensure confidentiality and integrity of commands and responses. This approach is defined in the Specification for Transfer of OpenC2 Messages via HTTPS [4] and represents one of the primary means of deploying OpenC2 in modern network environments.

The connection setup begins with establishing a TCP connection between the OpenC2 message producer (client) and consumer (server), followed by the standard TLS handshake, which creates a secure and encrypted channel. This secure transport layer protects the exchanged messages from interception or tampering, providing authentication of the communicating parties through certificates.

Once the TLS connection is established, OpenC2 messages are exchanged using HTTP/1.1 POST method. Commands and responses are encapsulated within the HTTP request and response bodies, formatted as JSON objects adhering to the OpenC2 Language Specification [2]. The POST method is preferred because it supports message payloads of arbitrary size, suitable for the structured command and response content. The URI used in the HTTP request is typically standardized as "/.well-known/openc2", serving as the designated endpoint on the consumer side for receiving and processing OpenC2 messages.

Several HTTP headers are essential in the exchange of OpenC2 messages over HTTPS ensuring proper identification, routing and processing of commands and responses, as shown in Figures 2.5 and 2.6:

- The HTTP POST request is directed to the URI "/.well-known/openc2", which is the standardized endpoint for OpenC2 messages over HTTPS.
- The **Content-Type** header is set to "application/openc2+json;version=1.0", indicating the payload format and OpenC2 version used. This allows the receiver to correctly parse and interpret the JSON-encoded OpenC2 content.
- The **Date** header provides a timestamp of when the message was created, which supports message correlation and replay protection.
- The **X-Request-ID** header carries a unique identifier that links requests and their corresponding responses, facilitating reliable tracking.

The OpenC2 message itself is embedded within the HTTP body as a JSON object containing two main sections: headers and body.

The **headers** field carries metadata such as:

- **request_id**: matching the X-Request-ID in the HTTP header to link the OpenC2 message to its HTTP request;
- **created**: a Unix epoch timestamp indicating when the message was created;
- **from** and **to**: identifiers of the message sender and receiver, supporting explicit addressing within the OpenC2 ecosystem.

The **body** contains the actual command or response.

```
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 POST with OpenC2 Command [4]

```
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 Response containing an OpenC2 Response [4]

2.5 Actuator Profiles

Within the suite of Open Command and Control (OpenC2) specifications, Actuator Profiles play a central role in adapting the general-purpose language to specific cyber defense functions. An Actuator Profile defines the subset of the OpenC2 language [2], including actions, targets, arguments and actuator specifiers, that are meaningful in the context of a particular operational domain (for example, stateless packet filtering or endpoint control).

Each Actuator Profile provides normative guidance on how the OpenC2 language elements are to be implemented for a given class of actuators. Specifically, a profile explicitly defines which **actions** are permitted, the **targets** to which they can be applied and the **arguments** that are supported or required, while also specifying the valid combinations among these elements and how they interrelate within the operational context of the actuator. Furthermore, an Actuator Profile can extend the base language by introducing custom targets or arguments that are specific to its operational domain, while still remaining compliant with the OpenC2 framework.

Beyond the core language elements, an Actuator Profile also defines the structure and semantics of **actuator specifiers**, which identify the specific actuator instance that should execute a command. These specifiers may include attributes such as device identifiers or logical groupings, allowing precise targeting in environments where multiple actuators are present.

Moreover, a profile may also introduce new **result** types that are specific to its operational domain, extending the language with domain-relevant output data. This flexibility allows each profile to accurately represent the feedback and telemetry that are most meaningful for its corresponding actuator class, while maintaining full compliance with the OpenC2 framework.

This balance between constraint and extensibility allows profiles to adapt to different technologies, such as firewalls, endpoint protection systems or intrusion detection tools, without fragmenting the standard, while ensuring that commands remain semantically appropriate and syntactically valid for the devices or services under management.

As stated in the OpenC2 Actuator Profile Development Process [3], profiles are developed following a structured methodology that promotes consistency across specifications and ensures compatibility with the core language [2] and architecture layers [1] of OpenC2.

The **Stateless Packet Filtering (SLPF)** profile stands as the primary Actuator Profile, illustrating how OpenC2 can be applied to implement and manage essential cyber defense functions, such as blocking or allowing network traffic and filtering packets based on protocol or port across diverse network environments.

2.6 Stateless Packet Filtering Profile (SLPF Profile)

The Stateless Packet Filtering (SLPF) Actuator Profile is designed to allow consistent and interoperable control over **stateless network filtering devices**. It defines the specific subset of the OpenC2 language that applies to packet filtering operations, constraining actions, targets, arguments and results to those meaningful in the context of network traffic management.
Packet filtering operations typically include inspecting individual network packets and making allow/deny decisions based on attributes such as source/destination IP addresses, transport protocol or source/destination port number, without maintaining session or connection state. These operations form the foundation for implementing firewall rules, access control policies and traffic segmentation.
The SLPF profile builds upon the core OpenC2 Language Specification [2] by utilizing a subset of the standard actions, targets and arguments defined in it, while also defining additional targets, arguments and result structures specific to stateless packet filtering operations, as detailed in the SLPF Actuator Profile specification [6].

To maintain consistency across the OpenC2 ecosystem and prevent ambiguity between elements originating from different specifications, the SLPF profile assigns a dedicated **namespace identifier**: **slpf**. This identifier is prefixed to all profile-specific constructs, such as custom targets (e.g., “slpf: rule_number”), arguments (e.g., "slpf: direction”), and result objects in responses (e.g., “slpf: rule_number”), to clearly distinguish them from those defined in the core language. The use of namespaces thus ensures unambiguous interpretation of commands and responses, promotes extensibility and facilitates interoperability when multiple actuator profiles coexist within the same OpenC2 implementation.

2.6.1 SLPF Actions

Within the SLPF Actuator Profile, the set of actions is defined as a subset of those established in the OpenC2 Language Specification [2]. These actions represent the fundamental operations that a producer can request from a stateless packet filtering actuator, each corresponding to a distinct network control behavior. The SLPF profile constrains and contextualizes their use to ensure predictable and interoperable execution. Specifically, the valid actions for the SLPF profile are:

- **allow:** permits network traffic whose characteristics match the filtering criteria defined within the target;
- **deny:** blocks network traffic whose characteristics match the filtering criteria defined within the target;
- **delete**: removes a previously defined rule or entry from the filtering configuration;
- **query**: initiates an information request toward the actuator, enabling the producer to learn the supported options and determine the state or settings of the packet filtering device;

- **update**: instructs the actuator to modify or refresh its configuration by obtaining and applying a new or updated configuration file or rule sets.

Each of these actions is bound to a specific set of valid targets and arguments, as defined within the profile [6], ensuring consistent interpretation and execution across heterogeneous implementations of stateless filtering systems.

2.6.2 SLPF Targets

The Stateless Packet Filtering (SLPF) Actuator Profile defines a precise subset of targets from the OpenC2 Language Specification [2], while also introducing a profile-specific target (rule_number) [6]. Each target identifies the entity or resource upon which an action operates, ensuring commands are meaningful and enforceable within stateless packet filtering devices. The defined targets for packet filtering operations are:

- **features**: a collection of capabilities or configurations of the actuator, such as supported action-target combinations, profile versions and available operational options.
- **file**: configuration or rule files that may be applied to the filtering engine. Each file target can include specific attributes such as *name*, *path* and *hashes* to precisely identify the file involved.
- **ipv4_net**: one or more IPv4 addresses specified in CIDR format, allowing the targeting of network segments for filtering purposes.
- **ipv6_net**: one or more IPv6 addresses specified in CIDR format, allowing the targeting of network segments for filtering purposes.
- **ipv4_connection**: a specific IPv4 network connection identified by a five-element tuple. For TCP, UDP or SCTP protocols, this tuple includes the source address, source port, destination address, destination port and protocol.
- **ipv6_connection**: a specific IPv6 network connection identified by a five-element tuple. For TCP, UDP or SCTP protocols, this tuple includes the source address, source port, destination address, destination port and protocol.
- **rule_number**: a profile-specific identifier for a particular filtering rule, used to precisely identify and delete the rule associated with that number.

	Allow	Deny	Query	Delete	Update
ipv4_connection	valid	valid			
ipv6_connection	valid	valid			
ipv4_net	valid	valid			
ipv6_net	valid	valid			
features			valid		
slpf:rule_number				valid	
file					valid

Fig. 2.7 Valid SLPF Action/Target pairs [6]

Each target can be combined with supported actions as shown in Figure 2.7 extracted from the OpenC2 SLPF Specifications [6], enabling precise and interoperable control over the packet filtering device
The namespace identifier ensures that the targets defined by the SLPF profile are uniquely scoped and correctly interpreted by actuators.

2.6.3 SLPF Arguments

The SLPF Actuator Profile defines a set of arguments that can be used with actions and targets, allowing precise control over stateless packet filtering operations. These arguments are divided into those inherited from the OpenC2 Language Specification [2] and those introduced specifically by the SLPF profile [6].
The language specification provides general-purpose arguments that apply across multiple actuator types. In the context of SLPF, the arguments defined by the language specifications are:

- **start_time:** defines when the action should begin;
- **stop_time:** defines when the action should end;
- **duration**: specifies the length of time for which the action remains active;
- **response_requested:** specifies the type of response expected from the actuator when executing the command, which can be *none*, *ack*, *status* or *complete*, depending on the required feedback level.

The SLPF profile introduces arguments designed for packet filtering, which provide fine-grained operational control:

- **direction**: specifies the traffic flow to which the action applies, with possible values *ingress*, *egress* or *both*, specifying respectively whether the filtering action concerns ingress traffic, egress traffic or traffic in both directions;
- **persistent**: determines whether a rule should persist across device reboots or configuration reloads;
- **insert_rule**: specifies the identifier or position of a rule within an ordered rule list, typically used in top-down rule sets to control where a new filtering rule should be inserted;
- **drop_process**: specifies how the actuator should handle packets that are denied by the filtering policy. It can take one of three values: *none*, meaning that the packet is silently dropped without notifying the source; *reject*, indicating that the packet is dropped and an ICMP host unreachable (or equivalent) message is sent to the source; *false_ack*, which instructs the actuator to drop the traffic while sending a false acknowledgment to the source.

The namespace identifier ensures that the targets defined by the SLPF profile, are uniquely scoped and correctly interpreted by actuators.
These profile-specific arguments are associated with relevant actions and targets, as shown in Figure 2.8 extracted from the OpenC2 SLPF Specifications [6], ensuring that commands can be applied precisely and consistently.

	Allow *target*	Deny *target*	Query features	Delete slpf:rule_number	Update file
response_requested	2.3.1	2.3.2	2.3.3.1	2.3.4.1	2.3.5.1
start_time	2.3.1	2.3.2		2.3.4.1	2.3.5.1
stop_time	2.3.1	2.3.2			
duration	2.3.1	2.3.2			
persistent	2.3.1	2.3.2			
direction	2.3.1	2.3.2			
insert_rule	2.3.1	2.3.2			
drop_process		2.3.2			

Fig. 2.8 Valid SLPF Action/Target/Arguments triplets [6]

2.6.4 SLPF Results

Within the OpenC2 framework, results represent the information returned by an actuator in response to a command, providing feedback about the execution status or the data requested. The SLPF profile extends this concept by defining a set of results that are meaningful for stateless filtering operations, ensuring that producers can consistently interpret the information returned by packet filtering devices [6].
In accordance with the OpenC2 Language Specification [2], every OpenC2 response includes a status code and optional result content.
The SLPF profile specifies the following result types:

- **versions:** lists the versions of the OpenC2 language supported by the actuator. This allows producers to determine compatibility and ensure that commands conform to the capabilities of the receiving device.
- **profiles:** identifies the actuator profiles implemented by the device, allowing producers to understand which OpenC2 profiles are available for interaction.
- **pairs:** represents the valid combinations of actions and targets supported by the actuator.
- **rate_limit:** communicates the actuator's command processing rate, providing insight into how frequently commands can be issued without overloading the device.
- **rule_number:** represents the number of the rule associated with the corresponding allow or deny command, providing a precise reference to the rule that was applied.

The first four results (*versions*, *profiles*, *pairs* and *rate_limit*) are defined in the OpenC2 Language Specification [2] and are returned in response to a query action.
The *rule_number* argument, instead, is introduced by the SLPF profile [6] itself and is returned in response to an allow or deny action.
As with targets and arguments, all results defined by the SLPF profile are associated with the SLPF namespace identifier, ensuring unique scoping and unambiguous interpretation within the OpenC2 ecosystem.

3 Firewall Technologies

Building upon the concepts introduced in the previous chapter, this section explores how the OpenC2 Stateless Packet Filtering (SLPF) profile can be applied to real-world firewall technologies. The SLPF profile provides a standardized and interoperable framework for expressing packet filtering commands, making it particularly suitable for integration with existing network defense components that rely on rule-based traffic control.
To illustrate its practical relevance, this chapter examines four representative firewall implementations: **iptables**, **OpenStack Security Groups, Kubernetes Network Policies** and **MS Azure Network Security Groups**. Although these systems differ in architecture, deployment context and configuration syntax, they all perform the essential function of controlling packet flows based on predefined criteria. By analyzing their respective rule models and management mechanisms, it becomes possible to understand how OpenC2, through the SLPF profile, can unify command and control across heterogeneous environments, laying the foundation for automation, consistency and coordinated policy enforcement in modern network infrastructures.

3.1 iptables

iptables [7] is a widely used **Linux utility** that provides packet filtering, network address translation (NAT) and other packet-mangling capabilities within the kernel. It operates by organizing **rules** into **chains**, which are grouped within **tables** depending on their function, commonly the **filter table** for general packet filtering, the **nat table** for address translation and the **mangle table** for specialized packet alterations.
Each rule specifies match criteria, such as source and destination IP addresses, protocols (TCP, UDP, ICMP) or ports and associates them with actions that determine how packets are handled: accept, drop or reject.
iptables rules can be combined to form complex policies that govern the flow of network traffic, providing both fine-grained control and high flexibility for administrators.
Rules in iptables are evaluated sequentially and the first match determines the fate of a packet. This allows for layered policies where general rules can be overridden by more specific ones and policies can be applied per interface or per network zone.
Administrators can persist and manage rules through saved configurations, scripts or higher-level tools.

It is important to note that rules configured directly in iptables are **not persistent**: they exist only in memory and are lost upon system restart unless explicitly saved and reloaded.
From the perspective of OpenC2 and the SLPF profile, iptables serves as a practical example of a packet filtering actuator that can be controlled programmatically. The SLPF profile's commands can be mapped to iptables operations: for example an *allow* command corresponds to adding a rule that accepts matching packets, while a *deny* command corresponds to a rule that drops or rejects traffic. Targets defined in SLPF, such as *ipv4_net*, *ipv6_net*, *ipv4_connection*, *ipv6_connection* and *file*, can be directly applied to iptables criteria, allowing precise specification of which packets are affected.
Arguments like *direction* and *drop_process* further refine how rules are applied and scheduled within iptables chains.
This approach demonstrates how the abstract definitions of SLPF translate into concrete, operational firewall management on Linux systems.

3.1.1 iptables Chains and Rules

The core functionality of iptables [7] focuses on defining **rules** that inspect and control network packets as they traverse a Linux system. These rules are organized into **tables**, each serving a specific purpose, and within each table rules are grouped into **chains**, which represent sequences of packet-processing steps. The most commonly used table is the **filter table**, which handles standard packet filtering tasks.
Within this table, three default chains are defined: INPUT, OUTPUT and FORWARD.
The **INPUT** chain processes packets destined for the local host, allowing administrators to accept, reject or drop incoming traffic based on a variety of criteria.
The **OUTPUT** chain deals with packets generated locally by the host, controlling what traffic leaves the system.
The **FORWARD** chain is responsible for packets that are routed through the system to other destinations, such as in the case of a Linux machine acting as a router or gateway.
Administrators can also define **custom chains** to group related rules, simplifying rule management and allowing more modular configurations. When a packet reaches a rule with a jump to a custom chain, it is processed according to the rules in that chain and then returned to the original chain unless explicitly directed otherwise.

Each chain contains a sequence of rules that match specific packet characteristics.
A rule typically defines one or more match criteria, such as **source** or **destination IP address**, **transport protocol** (TCP, UDP, ICMP), **source** or **destination ports**. These match criteria determine whether the rule applies to a given packet. Once a packet matches a rule, the target of the rule specifies the action to be taken. Standard targets include **ACCEPT**, which allows the packet to proceed to its destination; **DROP**, which silently discards the packet without notifying the sender; **REJECT**, which discards the packet while sending an error message (such as ICMP unreachable) back to the sender. Additional actions can also be implemented via extensions.
Furthermore, iptables allows administrators to **delete** filtering rules explicitly specifying its exact match with the `-D` option.

It is important to note that iptables evaluates rules in a **top-down order**, meaning the first rule that matches a packet dictates its fate. If no rules match, the chain's policy determines the default behavior, which can be set to ACCEPT, DROP or REJECT.
Figure 3.1 illustrates an ACCEPT rule inserted into the INPUT chain, specifying an IPv4 source address of 10.0.2.6, an IPv4 destination address of 10.0.2.15, using the TCP protocol with both source and destination port 80, while the chain's default policy is set to ACCEPT.

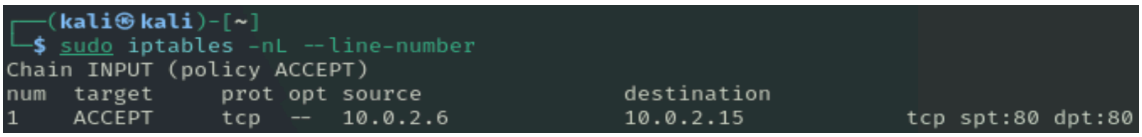

Fig. 3.1 Example of ACCEPT rule in INPUT chain

iptables rules are not persistent across system reboots, for this reason the current configuration can be saved to disk using the **iptables-save** command.
The resulting file, typically stored in `/etc/iptables/rules.v4` (`/etc/iptables/rules.v6` for IPv6 rules) or a custom location, can then be restored with **iptables-restore**, either manually or automatically during system startup.
This mechanism allows administrators to create reusable rule sets or deploy predefined firewall configurations directly from stored files.

3.1.2 IPv4 and IPv6 address handling

iptables primarily operates on IPv4 traffic, but its companion tool **ip6tables** extends the same rule-based filtering model to **IPv6**.
Both utilities share an almost identical syntax and semantics: rules for IPv6 are written and managed in the same way as for IPv4, except that they apply to IPv6 packet headers and address structures.
This separation ensures compatibility with dual-stack environments, where IPv4 and IPv6 traffic are processed independently according to their respective tables.

3.2 OpenStack

OpenStack is an **open-source cloud computing platform** designed to provide scalable and flexible infrastructure-as-a-service (IaaS) solutions. It allows organizations to deploy and manage large pools of compute, storage and networking resources in a multi-tenant environment. Within this framework, **virtual machines** represent the fundamental compute units provided to users, enabling the deployment of isolated workloads that can be dynamically created, configured and connected to virtual networks.
OpenStack is composed of multiple components, with **Neutron** [8] responsible for networking functionalities including **Security Rules**, which act as virtual firewalls for instances. These rules enable administrators to control the flow of ingress and egress traffic at the virtual machine level, specifying IP address, protocol type (TCP, UDP, ICMP), port range and traffic direction (ingress or egress). Each instance belongs to a **project** (tenant) and can be associated with one or more **Security Groups**, which are collections of rules that provide fine-grained, tenant-specific access control. This design ensures network isolation between projects and allows administrators to manage access policies consistently across multiple tenants.
Unlike traditional firewalls, OpenStack Security Rules function as a **white-list system**: only traffic that matches explicitly defined rules is permitted while all other traffic is implicitly denied. There is no separate `deny` action; the absence of an allow rule automatically blocks the traffic.
The rules are stored persistently within OpenStack's centralized database, ensuring that they remain effective across system restarts and can be deleted or dynamically modified to reflect changing security requirements. This design provides a consistent and scalable approach to enforcing network isolation and protecting workloads within cloud environments.
In this context, the SLPF profile can be used to automate and enforce security rules within OpenStack environments, mapping SLPF profile commands into OpenStack operations: for instance an *allow* command corresponds to creating a security rule that permits matching traffic, while a *delete* command removes an existing rule.
Targets defined in SLPF, such as *ipv4_net*, *ipv6_net*, *ipv4_connection* and *ipv6_connection*, can be applied to the IP addresses and connections specified in security rules, allowing precise targeting of network flows. Arguments like *direction* define whether the rule applies to ingress or egress, refining how traffic is controlled across projects and tenants.

3.2.1 OpenStack Security Groups and Rules

In OpenStack, **Security Groups** act as virtual firewalls that control inbound and outbound traffic to and from instances. Each security group is composed of one or more **Security Rules**, which define the specific network traffic that is permitted for associated virtual machines. When an instance is launched, it is attached to one or more security groups and all network traffic is filtered according to the cumulative set of rules defined within those groups.

Security groups operate at the **virtual network interface (vNIC)** level, meaning they apply directly to the network interfaces of instances rather than to shared network segments.

OpenStack security rules are **unidirectional**: each rule applies either to ingress or egress traffic. Consequently, separate rules must be created to allow bidirectional communication. Figure 3.2 presents an example of an ingress security rule.

```
┌──(kali㉿kali)-[~]
└─$ openstack security group rule list c8f8e56e-a479-4e02-8f2f-17f72a512794
+--------------------------------------+-------------+-----------+-----------------+------------+-----------+-----------------------+----------------------+
| ID                                   | IP Protocol | Ethertype | IP Range        | Port Range | Direction | Remote Security Group | Remote Address Group |
+--------------------------------------+-------------+-----------+-----------------+------------+-----------+-----------------------+----------------------+
| 5f8cbae2-62c5-4ac9-b1b6-d9f394fa5218 | tcp         | IPv4      | 192.168.0.201/32 | 80:80      | ingress   | None                  | None                 |
+--------------------------------------+-------------+-----------+-----------------+------------+-----------+-----------------------+----------------------+
```

Fig. 3.2 Example of ingress Security Rule

Each rule within a security group is associated with a **unique identifier (id)** and defines a set of parameters that determine how traffic is filtered [9]. Among the key attributes, the **direction** field specifies whether the rule applies to ingress (incoming) or egress (outgoing) traffic, while the **ethertype** identifies the IP version (IPv4 or IPv6). The **remote_ip_prefix** field specifies the source or destination IP address range in CIDR notation, depending on the traffic direction, while **remote_group_id** allows referencing another security group instead of a static IP range, enabling more dynamic and scalable relationships between instances.

The **protocol** parameter defines the transport protocol used, such as TCP, UDP, ICMP, or SCTP and the **port_range_min** and **port_range_max** values indicate the range of allowed destination ports (when both values are equal, the rule applies to a single port).

For example, Figure 3.2 shows an ingress security rule specifying IPv4 as the ethertype, a source IP address of 192.168.0.201/32, the TCP protocol and a port range of 80:80 to indicate a single port. The rule's unique identifier is also indicated.

Together, these parameters define the exact conditions under which traffic is permitted to or from an instance's network interface, providing granular control comparable to traditional firewall mechanisms.

Administrators can dynamically create, modify or delete security groups and their rules through the OpenStack Dashboard (Horizon), the OpenStack Command-Line Interface (CLI) or programmatically via the Neutron API.

3.3 Kubernetes

Kubernetes [10] is an open-source platform designed to **orchestrate and manage containerized applications** at scale, providing automated deployment, scaling and maintenance of workloads across clusters of machines.
A **cluster** is the fundamental unit of Kubernetes, consisting of a set of nodes (physical or virtual machines) that run containerized applications. Each node hosts one or more pods, which are the smallest deployable units in Kubernetes. A **pod** can contain one or more tightly coupled containers that share storage, network and runtime resources, and it provides the basic building block for running applications in the cluster.
A key organizational concept in Kubernetes is the **namespace**, which allows cluster administrators to logically isolate resources such as pods, services and network configurations. Namespaces make it possible to manage multi-tenant environments and separate development, testing and production environments within the same cluster.
Network Policies are a core feature in Kubernetes that define how pods can communicate with each other and with external endpoints. They function as programmable and declarative firewalls within the cluster, allowing administrators to control ingress (incoming) and egress (outgoing) traffic at the pod level. Each policy specifies rules that select which pods are affected using **labels**, the traffic direction (ingress or egress), allowed protocols (TCP, UDP or SCTP), IP blocks and port ranges.
Kubernetes Network Policies enforce a **default deny behavior** only for the pods they select: unless explicitly allowed by a policy, traffic to or from these pods is blocked.
In this way, Network Policies effectively create a whitelist-like model for selected pods, restricting communication strictly to the rules defined.
Pods without any applied Network Policy remain fully accessible, meaning that the default cluster behavior is **allow-all**.
Network Policies are persistently stored in the cluster's etcd database, ensuring that the rules remain effective across pod rescheduling, node failures or cluster restarts.
Policies are dynamic: they are automatically enforced as pods are created, deleted or moved across nodes, reflecting the current state of the cluster without requiring manual updates to firewall configurations. This makes them particularly suitable for highly dynamic environments where workloads are frequently changing.
From the perspective of OpenC2 and the SLPF profile, Kubernetes Network Policies represent a practical example of a programmable packet filtering actuator. SLPF commands such as *allow* and *delete* can be mapped to creating or removing policy rules. Targets like *ipv4_net*, *ipv6_net*, *ipv4_connection* and *ipv6_connection* can be applied to pod selectors and IP blocks to precisely define which traffic is affected. Arguments such as *direction* distinguish between ingress rules, egress rules or both, providing fine-grained control over traffic flows in the cluster.

3.3.1 Kubernetes Network Policies

Kubernetes **Network Policies** [10] are objects that define rules for controlling network traffic to and from pods. They provide a mechanism for administrators to enforce fine-

grained network segmentation within a cluster, allowing isolation of workloads and protection of sensitive services.
A Network Policy applies only to the pods within the namespace it is created in and it is implemented dynamically by the cluster's network plugin using the **Container Network Interface (CNI)**.
The primary mechanism for selecting which pods a Network Policy affects is **labels**. Labels are key-value pairs attached to pods that serve as identifiers or attributes, such as `app=frontend` or `role=db`. Within a Network Policy, **pod selectors** use these labels to determine the target set of pods for the rules. By using labels, policies can flexibly apply to dynamically changing workloads, automatically including new pods that match the selector without needing manual updates. Enforcement is handled in real time by the network plugin, which monitors pod labels, namespace assignments and rule definitions to permit or deny traffic according to the policy.
A Network Policy consists of one or more ingress and/or egress rules that define how traffic is allowed to flow to or from the selected pods. Each rule specifies a unique **name** within the namespace, **Policy Types** (the direction of the network policy: ingress, egress or both), **pod selectors** to identify the source or destination pods affected by the rule, **IPBlock** objects to permit traffic from or to specific IP ranges, **protocol** (TCP, UDP or SCTP) and **port ranges** or **individual ports** that are subject to the policy.
Network Policies can also be deleted when no longer needed, allowing administrators to revoke previously defined traffic permissions, or managed in bulk through **YAML files**, which define their configuration declaratively. By storing policies as YAML manifests, administrators can version, share and reapply them easily using Kubernetes command-line tools such as **kubectl apply**. This approach permits efficient policy management across environments, allowing consistent deployment, modification and removal of network security rules.
Figure 3.1 shows an example of an ingress Network Policy. In the Spec section (short for specification), the detailed configuration of the policy is defined, including the pod selector (in this case `oc2-net-10-17-2-44-32=true`) which identifies the pods the policy applies to, the traffic direction (ingress), the protocol (TCP), the port being controlled (80) and the source IP address from which traffic is allowed (10.17.1.62/32).

```
┌──(kali㉿kali)-[~]
└─$ kubectl describe networkpolicy slpf-np-mt25n -n my5gtestbed
Name:         slpf-np-mt25n
Namespace:    my5gtestbed
Created on:   2025-10-27 16:22:54 +0100 CET
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     oc2-net-10-17-2-44-32=true
  Allowing ingress traffic:
    To Port: 80/TCP
    From:
      IPBlock:
        CIDR: 10.17.1.62/32
        Except:
  Not affecting egress traffic
  Policy Types: Ingress
```

Fig. 3.3 Example of ingress Network Policy

3.4 MS Azure

Microsoft Azure (MS Azure) is a cloud computing platform developed by Microsoft that provides a wide range of infrastructure, platform and software services. It enables organizations to deploy and manage applications across Microsoft-managed data centers worldwide, offering high scalability, resilience and integration with enterprise ecosystems.
In the context of network security, Azure provides multiple mechanisms to control and filter traffic within virtual networks, primarily through Network Security Groups and Azure Firewall.
An Azure **Network Security Group (NSGs)** acts as a distributed, stateful firewall that allows or denies inbound and outbound traffic to network interfaces (NICs), subnets or virtual machines [11]. Each NSG contains a list of **security rules** that define how traffic is filtered based on parameters such as source and destination IP addresses, protocols (TCP, UDP, ICMP) and ports. Each rule also specifies the direction (inbound or outbound) and the action (allow or deny), as well as optional tags to simplify configuration across Azure services (e.g., “Internet,” “VirtualNetwork,” “LoadBalancer”).
The rules are evaluated in **priority order**, from the lowest number to the highest, until a match is found. Default rules always exist to allow essential Azure infrastructure communication while blocking unwanted traffic.
Azure NSGs operate according to an explicit **allow/deny model**, effectively functioning as a whitelist/blacklist hybrid. If no rule matches a given packet, the default action is to deny the traffic, providing a secure baseline configuration.
Rules in NSGs are **persistent**, stored and managed as part of the Azure resource configuration. They can be deleted or dynamically modified through the Azure Portal, the Azure CLI, PowerShell or REST APIs and changes take effect immediately across the infrastructure.
From the perspective of OpenC2 and the SLPF profile, Azure NSGs can act as packet-filtering actuator. Actions such as *allow* and *deny* directly correspond to Azure security rule configurations, while targets like *ipv4_net*, *ipv6_net*, *ipv4_connection* and *ipv6_connection* can map to Azure’s address and port definitions. Arguments such as *direction* are also directly applicable, as each rule explicitly defines whether it applies to inbound or outbound traffic. This makes Azure’s network security features a practical real-world environment where the SLPF profile could be applied to automate firewall policy management across cloud networks.

3.4.1 MS Azure NSGs and Security Rules

Azure **Network Security Groups (NSGs)** [11] are logical containers for a set of security rules that control inbound and outbound traffic at the level of network interfaces, subnets or virtual machines. Each NSG is created within an Azure **Resource Group**, a logical container that organizes and manages related resources such as virtual networks, virtual machines and storage accounts. While the resource group does not directly affect the behavior of the NSG, it provides the necessary context for its identification and lifecycle management within the Azure environment.

Each NSG can be associated with one or more resources, allowing fine-grained control over network communication within a virtual network. NSGs provide stateful filtering, meaning that once a connection is allowed in one direction, the return traffic is automatically permitted without requiring an explicit rule.

Each **security rule** within an NSG specifies conditions that determine whether a packet is allowed or denied. The main parameters of a rule include the **name**, which must be unique within the network security group, the **priority**, which determines the order in which rules are evaluated, the **direction** of traffic (inbound or outbound), the **source** and **destination**, which can be individual IP addresses, ranges or service tags, the **protocol** (TCP, UDP, ICMP or Any), and the **port** or **port range**. Rules also include an **action** field, which explicitly sets whether matching traffic is allowed or denied, underlining how Azure NSGs supports explicit deny rules.
Figure 3.4 shows an example of an inbound security rule configured to block all incoming traffic, with a priority value of 65500.

Priority	Source	Source ports	Destination	Destination ports	Protocol	Access
65500	0.0.0.0/0	0-65535	0.0.0.0/0	0-65535	Any	Deny

Fig. 3.4 Example of an inbound Azure security rule [11]

Rules within an NSG are evaluated in **priority order**, from the lowest number to the highest, stopping at the first match. This deterministic processing ensures that the most specific or critical rules are applied first, while default rules enforce security by denying traffic not explicitly allowed.
Two rules cannot share the same name and they cannot have the same combination of priority and direction, ensuring that each packet matches exactly one rule at a given evaluation step. However, rules with the same priority but opposite directions (one inbound, one outbound) are valid and are evaluated independently within their respective traffic flows.
NSGs are persistent, meaning that rules remain effective across virtual machine restarts or network changes and can be modified dynamically through the Azure portal, CLI or REST API. Administrators can also create or delete rules, supporting automation and integration with deployment pipelines.

4 Thesis objectives

The objective of this thesis is to **design, implement and evaluate** a fully-featured Actuator Manager that implements the **Open Command and Control (OpenC2) Stateless Packet Filtering (SLPF) Actuator Profile** for multiple firewall technologies [6].
The project aims to demonstrate the interoperability and flexibility of OpenC2 by applying its standardized command structure to heterogeneous environments, namely **iptables**, **OpenStack Security Groups**, **Kubernetes Network Policies** and **Microsoft Azure Network Security Groups (NSGs)**.
The implementation will cover the entire set of SLPF-defined language elements, ensuring full support for:

- **Actions**: *query*, *allow*, *deny*, *delete* and *update*, which respectively request status or configuration information, permit traffic, block traffic, remove existing filtering rules and update the actuator's configuration.
- **Targets**: *features*, *ipv4_net*, *ipv6_net*, *ipv4_connection*, *ipv6_connection*, *file* and the profile-specific *rule_number*.
- **Arguments**: from the core Language Specification (such as *start_time*, *stop_time*, *duration and response_requested*) and the SLPF profile (such as *direction*, *persistent*, *insert_rule and drop_process*).

Each of the four selected firewall technologies has been implemented as a dedicated SLPF Actuator, managed by the SLPF Actuator Manager. Each actuator extends the Actuator Manager's functionality to support the specific firewall mechanisms of the corresponding system, including a translation layer that maps OpenC2 commands to the native rule or policy definitions of the underlying technology.
To ensure the correctness and reliability of the implementation, the thesis will include a **three-phase testing and validation process**:

1. **Syntax and Semantic Validation**: to verify that the implemented actuators correctly interpret and process OpenC2 commands as defined in the SLPF and Language Specifications. This includes checking message formatting, field validation and adherence to namespace identifiers.
2. **Functional Testing**: to verify that OpenC2 commands produce the intended network behavior on each firewall. These tests will confirm that traffic is correctly allowed or blocked according to the creation or deletion of filtering rules.
3. **Performance Evaluation**: to assess the responsiveness and efficiency of each actuator implementation. On the producer side, latency is measured as the elapsed

time between the transmission of an OpenC2 command and the receipt of the corresponding response message. On the consumer side, two metrics are collected: the total execution time from command reception to completion, and the specific execution time of the underlying firewall operation (such as rule insertion or deletion). These measurements provide insight into communication overhead and the internal processing efficiency of each implementation.

Additionally, the thesis will conduct a **critical analysis of the SLPF profile** itself, examining its design choices, clarity and practical applicability in real-world environments. Particular attention will be given to areas such as the granularity of actions, the adequacy of targets and arguments in capturing the behavior and configuration models of different firewalls and the handling of platform-specific differences.

The outcome of this analysis aims to provide valuable feedback for future revisions of the SLPF specification [6] and to contribute to the adoption of OpenC2 as a unified standard for cyber defense automation.

5 SLPF Actuator Manager

This chapter presents the design and implementation of a fully functional Actuator Manager compliant with the Open Command and Control (OpenC2) Stateless Packet Filtering (SLPF) Actuator Profile. The implementation aims to provide a concrete realization of the SLPF Specification [6] across four heterogeneous firewall technologies: **iptables**, **OpenStack Security Groups**, **Kubernetes Network Policies** and **Microsoft Azure Network Security Groups (NSGs)**.
The main goal of this implementation is to validate the interoperability and flexibility of OpenC2 by creating a standardized command interface that can consistently manage packet-filtering capabilities across diverse platforms. To achieve this, a modular and extensible architecture has been developed, consisting of a generic **SLPF Actuator Manager** and a set of **backend-specific actuators** responsible for mapping OpenC2 commands to the native syntax and semantics of each firewall system.
The following sections describe the design and implementation of the proposed solution, including an explanation of the overall architecture and covering the initialization process of the SLPF Actuator Manager as well as that of the specific firewall integrations. It describes how the actuator handles the reception and validation of OpenC2 commands, including the verification of their actions, targets and arguments according to the SLPF Specification [6]. Moreover, the chapter details the implementation of the supported actions across the different backend modules, outlining how each platform-specific component executes the intended packet-filtering operations.

5.1 SLPF Actuator Manager Architecture

The **SLPF Actuator Manager**, namely **SLPFActuator**, has been designed following a modular and extensible architecture that separates core functionalities from platform-specific implementations. This separation is realized through an **object-oriented design based on inheritance**, where the SLPF Actuator Manager acts as a **superclass** managing the generic operations shared by any SLPF-compliant actuator. Platform-specific actuators, implemented as **subclasses**, inherit these functionalities and extend SLPFActuator to handle the particularities of each firewall technology.
By adopting this design, common functionalities, such as the handling of standard arguments like *start_time*, *stop_time*, *duration*, *response_requested*, *persistent* and *insert_rule,* are implemented once within the SLPFActuator and are automatically available

to all backend modules. This significantly reduces the development effort required to integrate additional platforms in the future, as only the backend-specific arguments (*direction* and *drop_process*) need to be implemented for each new firewall.
Each specific implementation must override selected methods of the SLPFActuator superclass to ensure proper operation within its environment, particularly those responsible for handling the SLPF-specific actions (*query*, *allow*, *deny*, *delete* and *update*). In addition, each implementation must define how the OpenC2 command targets are mapped to the corresponding entities within the specific platform.
On the other hand, the initialization and shutdown procedures of the SLPFActuator are already implemented, ensuring that these fundamental operations do not need to be redefined in each backend subclass.

Since the *allow* and *deny* actions create filtering rules that might later be referenced by a *delete* action or that may be persistent across system restarts, the SLPFActuator itself is responsible for managing a **database of active rules**. It maintains a consistent record by inserting new entries in response to allow and deny commands and removing rules when delete commands are received. Additionally, for backend implementations that do not natively support persistent rules, the SLPFActuator coordinates the saving of rules at system shutdown and their restoration upon system restart, although each specific implementation must provide the methods to perform these save and restore operations.

As the *start_time*, *stop_time* and *duration* arguments can be used to schedule the execution of OpenC2 commands at a future time, the SLPFActuator also includes a **scheduling mechanism**. This scheduler is responsible for delaying command execution until the defined start time and, when specified, automatically revoking their effect at the end time (as determined by the *stop_time* or *duration* arguments). Each platform-specific actuator is associated with its own dedicated scheduler, ensuring correct handling of timing parameters within the respective backend environment.
To ensure reliability across system restarts, the SLPFActuator saves any pending or scheduled commands in a dedicated database table, allowing them to be restored and resumed upon reboot. Figure 5.1 illustrates the overall architecture of the system.

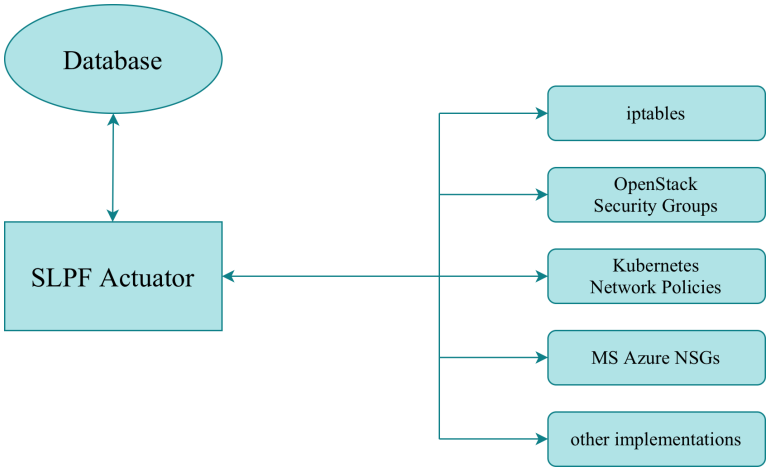

Fig. 5.1 SLPF Actuator Manager Architecture

As specified in the SLPF Actuator Profile Specification [6], the *update* action allows an actuator to refresh its active filtering rules based on the contents of a file provided as target. When an update command is received, all currently active rules in the specified firewall are removed and replaced with those defined in the file. To support this behavior, the SLPFActuator can operate in two modes: **DB mode** and **FILE mode**.
In **DB mode**, the actuator uses the rules stored in the internal database. For each allow or deny command received, a new rule is added to the database, while a delete command removes the corresponding one. If an update command is received while in DB mode, all active filtering rules are removed from both the database and the specific backend implementation, the actuator switches to FILE mode and the update command is executed.
In **FILE mode**, the actuator applies the rules specified in the file used as target of the update command. If an update command is received while in this mode, the filtering rules in the specific backend implementation are refreshed again according to the new file. Delete commands have no effect in FILE mode, as there are no entries in the database. If an allow or deny command is received while in FILE mode, all currently active rules in the specific firewall (those defined by the last target file received) are removed, the actuator switches to DB mode and the new rule is added to the database.

When a command is received, the SLPFActuator first performs a validation phase to ensure the command (and its associated action, target and arguments) conform to the OpenC2 Language Specification [2] and the SLPF Specification [6].
Once validated, the command is handled by the method corresponding to its specified action. Within this method, the SLPFActuator verifies the type correctness of the target and arguments, while the specific backend implementation ensures that the requested elements are actually supported. During this process, the backend implementation may also supply custom data to the SLPFActuator, containing information useful for executing allow, deny and delete commands, which the actuator will store in the internal database.
Subsequently, default values are assigned to any arguments that were not explicitly provided.
For commands that manage filtering rules, such as allow, deny or delete, the SLPFActuator interacts with the internal database accordingly, also storing any associated custom data provided by the backend implementation.
Finally, the command is scheduled for background execution at the specified start time, with its effect automatically revoked at the designated end time (when specified), and a response is sent back to the producer. Since the action, target and arguments have already been fully validated, a response can be sent to the producer even before the command is actually executed by the specific backend implementation. The execution itself is handled by the actuator corresponding to the targeted firewall, where platform-specific operations are performed according to the native syntax and semantics of the underlying system.
This architecture ensures both consistency and reliability across heterogeneous platforms, while maintaining extensibility, allowing new backend actuators to leverage the core functionalities without duplicating fundamental logic.

5.2 SLPFActuator Initialization

The initialization process of the SLPFActuator, as well as that of each specific backend implementation, is based on **configuration files**. This design choice avoids the use of static attributes, allowing configuration flexibility, easier maintenance and facilitating deployment across different environments without requiring code modifications.
Being the superclass, the SLPFActuator is not instantiated directly. Instead, its initialization is performed as the final step of the startup procedure of each specific backend implementation, which provides all the necessary parameters required to correctly initialize the core actuator. These parameters include:

- **Actuator specifiers**, used to identify the specific implementation: *hostname*, *named_group*, *asset_id* and *asset_tuple*. Among these, *asset_id* serves as the primary specifier, providing a unique identifier for the given actuator instance.
- **Database configuration parameters**, namely *db_path_directory*, which specifies the absolute path to the directory containing (or where to create) the database; *db_name*, which defines the database name; *db_commands_table_name* and *db_jobs_table_name*, which indicate the tables used respectively to store active filtering rules and scheduled but not yet executed commands in case of a system shutdown.
- The **update parameter** *update_directory_path* defines the absolute path to the default directory that contains the files used for update commands.

None of these parameters are strictly required for initialization. When one or more are not provided, default values are automatically assigned during the initialization process.
After parameter setup, the actuator proceeds with **database initialization**. If the database or the required tables are not present, they are automatically created.
The actuator then checks its current **operating mode,** DB mode or FILE mode, to determine which data source will be used for active filtering rules.
Next, persistent rules are restored in case the specific backend implementation does not natively support persistence across restarts. This operation is performed by calling a method that must be overridden by each backend implementation, as the restoration process depends on the underlying firewall system.
Subsequently, the **scheduler** responsible for command execution management is initialized. Each backend actuator is associated with its own scheduler instance, designed to ensure that only one command is executed at a time. If the execution intervals of two commands overlap, the second command is delayed until the first has completed. This serialized execution model prevents race conditions and ensures consistent modification of filtering rules, avoiding potential conflicts between concurrent operations.
Finally, any previously scheduled but unexecuted commands, which were saved in the database due to a system shutdown, are restored and reloaded into the scheduler. This mechanism guarantees operational continuity and command reliability, ensuring that no scheduled actions are lost between system restarts. Once all pending commands have been restored, the scheduler is started, marking the completion of the initialization process.

5.3 Query action

The **query action** is used to initiate an information request toward the actuator, allowing the producer to obtain details about the actuator's capabilities and configuration. Through this action, the producer can learn which options are supported and determine the current settings of the packet-filtering device. Specifically, a query response can include several types of information: *versions,* lists the versions of the OpenC2 language supported by the actuator; *profiles,* identifies the actuator profiles implemented by the device; *pairs,* represents the valid combinations of actions and targets supported by the actuator; *rate_limit,* communicates the actuator's command processing rate.
SLPFActuator provides a complete **base implementation** of the query action. In this default implementation, the actuator returns:

- as **version**, the OpenC2 language version supported by the SLPFActuator, namely 1.0;
- as **profiles**, the list of actuator profiles implemented by the SLPFActuator;
- as **pairs**, the complete set of action–target combinations defined in the SLPF Specification [6];
- while the **rate_limit** feature is currently not implemented.

If a specific backend implementation supports a different OpenC2 language version or does not implement some of the actions or targets defined in the SLPF Specification [6], it must explicitly **override** this method to accurately report the supported features to the producer. This ensures that the producer always receives an accurate and current description of the actuator's capabilities, preserving interoperability and preventing invalid command execution.

5.4 Allow and deny actions

The **allow and deny actions** are responsible for managing the packet-filtering rules that determine which network traffic is permitted or blocked by the actuator. Both actions support the same set of targets, namely *ipv4_net*, *ipv6_net*, *ipv4_connection* and *ipv6_connection*, and share a common set of arguments: *start_time*, *stop_time*, *duration*, *response_requested, direction*, *insert_rule* and *persistent*. The only exception is the *drop_process* argument, which is exclusively supported by the deny action. Given their nearly identical structure and behavior, these two actions are handled and implemented together. Because the *ipv4_net* and *ipv6_net* targets define only a single address as specified in the OpenC2 Language Specifications [2], the SLPFActuator is designed to interpret this address as the destination of the traffic to be permitted or blocked.
When processing an allow or deny command, the SLPFActuator first verifies the **type correctness** of the command's target, the parameters it contains and all provided arguments. It then performs an overall **validity check** to ensure the coherence of the command. In particular, the actuator verifies that if a source or destination port is specified, a corresponding protocol must also be defined, since ports cannot be specified independently of a protocol. Furthermore, it ensures that the protocol value is one of TCP, UDP or SCTP, as these are the only protocols that support source and destination ports according to the SLPF Specification [6].

Regarding the command arguments, the actuator enforces the constraints defined in both the OpenC2 Language Specification [2] and the SLPF Specification [6]. Specifically, it checks that the arguments *start_time*, *stop_time* and *duration* are not used simultaneously and that if the *insert_rule* argument is present, the *response_requested* argument must also be included. If any of these conditions are violated, the actuator returns a response to the producer with status code 400 (Bad Request). Additionally, if a command includes the *insert_rule* argument and a rule with the same rule number already exists in the database, the actuator rejects the command and responds with status code 501 (Not Implemented) and status text 'Rule number currently in use' as specified in the SLPF Specification [6].
After these initial validations, the command is passed to the specific backend implementation for further verification. This **backend-level validation** allows each platform-specific module to check whether the requested action, target and arguments are supported and to return any custom data that may be required for the later execution of the command. These data are then stored in the internal database by the SLPFActuator. Backend implementations that do not support certain features must explicitly override this method to provide their own behavior.
Default values are then assigned to arguments that were not explicitly provided: the current date and time for *start_time* (when both *stop_time* and *duration* are not specified.), 'both' for *direction*, 'true' for *persistent* and 'none' for *drop_process*. If the *duration* argument is present but either *start_time* or *stop_time* is missing, the actuator computes the missing value according to the relation '*stop_time* = *start_time* + *duration*' as defined in the OpenC2 Language Specification [2].
Before proceeding with database management, the actuator checks its current operating mode. If it is in **FILE mode**, all active filtering rules previously loaded from the file specified in the update command are deleted from the backend implementation. This operation is carried out through a method that must be overridden by each specific implementation, as the process of removing rules depends on the characteristics of the underlying firewall system.
The actuator then proceeds to the **database management phase**, where the validated command, together with its action, target, arguments and any associated custom data, is recorded in the database. A unique **rule_number** is assigned to the new entry and returned to the producer to allow future reference to that specific rule.
Next, the command is scheduled for execution at the defined start time and, if an end time has been specified, the corresponding removal of the rule is also scheduled. The actual execution of these operations is handled by the specific backend implementation, meaning that each implementation supporting the allow and deny actions must override the corresponding methods to define how network traffic is permitted or blocked within its native environment. To perform the requested action, the backend implementation receives from the SLPFActuator the *target* and the relevant *arguments* required for command execution, specifically the *direction* argument in the case of an *allow* action, and both *direction* and *drop_process* in the case of a *deny* action, as well as any custom data necessary to complete the operation. Finally, a success response is sent back to the producer, including the assigned *rule_number*.

If any unexpected error occurs during the execution of the *allow* or *deny* command, the SLPFActuator returns a response with status code 500 (Internal Error) and status text 'Rule not updated', as specified in the SLPF Specification [6].

5.5 Delete action

The **delete action** is responsible for removing an existing filtering rule from the system. It supports the target *rule_number* and the arguments *response_requested* and *start_time*.
When managing a delete command, the SLPFActuator first performs a **type check** on the target and arguments. Unlike allow and deny commands, a validity check and backend validation are not necessary, since the target and all arguments are already fully supported by the SLPFActuator. If the *start_time* argument is not explicitly provided, it is set to the current date and time.
The actuator then checks whether a rule with the specified rule number exists in the database. If no matching rule is found, a response with status code 500 (Internal Error) is sent to the producer. If the rule exists, it is retrieved from the database along with its action, target, arguments and any associated custom data and the execution of the delete command is scheduled for the specified start time.
The delete command also interacts with allow or deny commands that include a *stop_time* argument. When the scheduled execution time of the delete command arrives, the actuator checks for any scheduled process intended to revoke the effect of the corresponding allow or deny command. If such a process exists, it is cancelled; if the process has already completed, the rule has already been removed from the database and the delete command execution terminates.
Similarly, the *start_time* argument of an allow or deny command influences the behavior of a delete command. If the start time of a scheduled allow or deny command has not yet arrived, the rule exists in the database but has not yet been applied in the specific backend implementation. In this case, the scheduled command that would activate the rule is cancelled.
Finally, unless the rule has already been removed due to a *stop_time* argument from an allow or deny command, the delete command removes the rule from both the database and the specific backend implementation.
The execution of the delete action is delegated to the specific backend implementation, which is responsible for defining how filtering rules are removed within its native environment. To carry out the deletion, the backend implementation receives from the SLPFActuator the command to be removed, including its action, target and arguments, as well as any custom data associated with the original command.
If any unexpected error occurs during the execution of a delete command, the SLPFActuator returns a response with status code 500 (Internal Error) and status text 'Firewall rule not removed or updated', as specified in the SLPF Specification [6].

5.6 Update action

The **update** action instructs the actuator to modify or refresh its configuration by obtaining and applying a new configuration file or rule set. It supports the *file* target, which includes three key attributes: *name*, which is mandatory and specifies the name of the file; *path*, which, when present, defines the absolute location of the file; *hashes*, which may contain one or more cryptographic digest values used to verify the integrity of the file before it is applied. Within the SLPFActuator, the supported hashing algorithms are **MD5**, **SHA1** and **SHA256**, in accordance with the definitions provided in the OpenC2 Language Specification [2]. The action also supports the *response_requested* and *start_time* arguments.

When processing an update command, the SLPFActuator begins by performing a **backend validation** to verify that the action and the file extension are actually supported by the specific firewall implementation. This validation step is executed through the same mechanism used for the allow and deny actions, checking whether the action, target and arguments are implemented by the backend module.

After confirming backend support, the actuator performs **type checking** on the command target, including its name, path and hashes attributes, as well as on the command arguments. The presence of the name attribute is verified, as mandated by the SLPF Specification [6]. Next, the absolute file path is determined by the *path* parameter in the command target, if provided; otherwise, it is derived by combining the *name* specified in the target with the default directory defined by the *update_directory_path* parameter set during initialization. Once the full path has been resolved, the actuator verifies that the corresponding file actually exists.

If the file is successfully located, its MD5, SHA1 and SHA256 hashes are computed and compared with the corresponding values specified in the command target (if provided), ensuring the integrity and validity of the file before execution.

If the *start_time* argument is not explicitly specified, it is set to the current date and time and the command is scheduled for execution at that moment.

When the scheduled time is reached, the SLPFActuator checks its current operating mode. If it is in **DB mode**, it switches to FILE mode, removing all filtering rules stored in the database and any active rules in the specific backend implementation. This cleanup is performed using the same method employed by the allow and deny actions when switching from FILE mode back to DB mode, ensuring consistent state management across both directions of mode transition. Additionally, any pending jobs in the scheduler associated with the deleted database rules are also canceled.

Finally, the actuator invokes the method responsible for the actual execution of the update action. Each backend implementation that supports this action must explicitly override this method, defining how the configuration or rules file is applied within its native environment. For correct execution, the SLPFActuator provides the backend implementation with the file name and absolute path of the file to be processed.

If any unexpected error occurs during the execution of an update command, the SLPFActuator returns a response with status code 500 (Internal Error) and status text ‘File not updated’, as specified in the SLPF Specification [6].

5.7 Database management

The database represents a fundamental component of the developed SLPF Actuator implementation. It enables the persistent storage and structured management of active filtering rules created through allow and deny commands received from the producer, as well as their deletion upon delete commands and their restoration after system restarts.
Each backend implementation of the SLPFActuator interacts with two dedicated database tables: one for active filtering rules and another for scheduled commands.
The **active filtering rules table** stores all the essential information associated with each allow or deny command received from the producer. This includes the **rule number** (the unique identifier of each rule), the **action** (*allow* or *deny*) and, in the case of deny action, the **drop_process** argument, which can assume the values *none*, *reject* or *false_ack*. It also records the **direction** argument (*ingress*, *egress* or *both*), the **target type** (*ipv4_net*, *ipv6_net*, *ipv4_connection* or *ipv6_connection*) and all its parameters, such as **source** and **destination addresses**, **protocol**, and **source** and **destination ports**. Storing this information allows for easy reconstruction of the command when a delete action must be executed. Additional stored fields include the **start_time** and **stop_time** arguments, which define the activation and deactivation times of the rule, and the **persistent** argument, which specifies whether the rule must survive system restarts. The table also keeps any **custom data** returned by the backend implementation, as well as **scheduler data** used to track the processes responsible for activating or revoking rules in response to *start_time* or *stop_time* arguments when a delete command is received from the producer.
The **scheduled commands table**, on the other hand, stores information about commands that were scheduled for execution but had not yet been executed at the time of system shutdown. Specifically, it maintains the process identifier responsible for executing the command, the corresponding execution date and time, and the action, target and arguments defining the command itself. This ensures that any pending operations are not lost during restarts, preserving consistency and allowing for automatic resumption of scheduled tasks.
The class defining the database provides all the necessary methods for managing both active filtering rules and scheduled command entries, offering a unified and coherent interface for database interactions.
The database is entirely managed by the SLPFActuator itself. As a result, each new backend implementation can rely on the existing infrastructure without the need to reimplement its own database management logic. During initialization, the actuator automatically creates the required tables for each backend implementation within the shared database environment. This approach simplifies the integration process, ensures uniform behavior across implementations and enhances system scalability by allowing new platforms to be supported with minimal development effort.

5.8 SLPFActuator shutdown

The shutdown procedures of the SLPFActuator are essential to ensure the correct and consistent operation of the system and are entirely managed by the actuator itself.
During this phase, all non-persistent commands are removed from both the internal database and the corresponding backend implementation, ensuring that only rules explicitly marked for persistence remain active.
Then, all remaining persistent rules within the specific backend implementation are saved, allowing them to be properly restored when the system restarts. Each backend implementation that does not natively support rule persistence must override the corresponding method to ensure that the SLPFActuator can handle this process correctly across restarts.
Finally, all processes associated with scheduled but not yet executed commands are stored in the database, preserving their state for future recovery. Once this operation is completed, the scheduler linked to the specific backend implementation is shut down, ensuring a clean and reliable termination of all background activities.

5.9 SLPF Actuator for iptables

To manage the filtering rules related to the **iptables** ecosystem, a dedicated subclass of the SLPFActuator has been implemented, named **SLPFActuator_iptables**.
SLPF Commands are applied in the iptables environment through the creation of **custom chains**: three dedicated to IPv4 traffic and three to IPv6 traffic, each structured to handle input, output and forward packet flows.
The iptables framework enables the creation and deletion of filtering rules of type allow and deny. Since these rules are natively non-persistent, it can interact with rule files having the '.v4' and '.v6' extensions. This interaction allows the active filtering rules to be properly saved when the device is shut down and restored at startup. The saving and restoration of rules are performed by overriding the corresponding methods of the SLPF Actuator Manager and using the *iptables-save* and *iptables-restore* commands, or *ip6tables-save* and *ip6tables-restore* for IPv6 addresses.
Furthermore, iptables supports updating the current active rule set with the content of a rule file, which allows the correct implementation of the update action. Thus, the method used by the SLPFActuator to clear all active filtering rules during a transition between DB mode and FILE mode (or vice versa) is properly overridden and implemented to remove all active rules across all iptables and ip6tables chains.
This versatility makes iptables a natural fit for the SLPF profile, allowing for the full implementation of all its actions, targets and arguments, with the only exception of the drop_process argument set to *false_ack*, which is not natively supported by the firewall. For this reason, the SLPFActuator_iptables class does not need to redefine the query action: since it supports all the features required by the SLPF profile, it can rely directly on the implementation already provided by the SLPF Actuator Manager.

5.9.1 SLPFActuator_iptables Initialization

The initialization of SLPFActuator_iptables is carried out entirely through **configuration files**, which provide all the parameters required to properly set up the actuator.
This approach offers significant advantages, including greater flexibility and simplified maintenance, as changes to the actuator's behavior can be made without modifying the code.
The configuration files provide the parameters required to correctly initialize the SLPF Actuator Manager (as described in Subchapter 5.2). The most important parameter is asset_id, which is set to *iptables* if not specified in the configuration file. In addition, the configuration includes parameters specific to the iptables environment:

- **iptables_rules_directory_path**, which specifies the absolute path to the directory where the iptables rule files will be saved in the event of a system shutdown or an update action.
- **iptables_rules_v4_filename** and **iptables_rules_v6_filename**, which define the filenames for the IPv4 and IPv6 rule files, respectively.
- **iptables_input_chain_name**, **iptables_output_chain_name** and **iptables_forward_chain_name**, which allow customization of the names of the chains that will be created to manage iptables filtering rules.
- **iptables_cmd** and **ip6tables_cmd**, which specify the base command to interact with the iptables environment, which, being a command-line-only firewall, requires explicit command invocation for all operations.

None of these parameters are strictly mandatory for the actuator's correct initialization: default values are automatically assigned to all parameters if they are not explicitly specified, ensuring that SLPFActuator_iptables can function properly.
After validating the provided parameters, SLPFActuator_iptables creates three custom chains for IPv4 filtering rules, one each for input, output and forward, and three corresponding chains for IPv6 filtering rules, only if they do not already exist. These custom chains are then linked to the three main chains: INPUT, OUTPUT and FORWARD.
Next, the actuator verifies the existence of the directory specified by *iptables_rules_directory_path*. If the path is valid, it creates the files corresponding to the IPv4 and IPv6 filtering rules. Finally, SLPFActuator_iptables calls the initialization method of the SLPF Actuator Manager, marking the completion of its own initialization process.

5.9.2 Allow and deny actions

SLPFActuator_iptables supports both the allow and deny actions by overriding the corresponding methods of the SLPF Actuator Manager in order to implement these operations within the iptables environment. As explained in Subchapter 5.4, this actuator receives from the SLPFActuator the *target* of the allow or deny action, the *direction* argument and, in the case of a deny action, the *drop_process* argument. The iptables actuator does not require custom data to perform its actions, therefore they are not retrieved from the SLPF Actuator Manager or stored in the database.
When the actuator needs to execute an allow or deny action, the first step is to construct the **iptables command**. The command begins with the *iptables_cmd* or *ip6tables_cmd* parameter provided during initialization, depending on whether the target specified in

the OpenC2 command is IPv4 or IPv6. Next, the actuator selects the appropriate custom chain in which to insert the filtering rule based on the value of the *direction* argument. Since the FORWARD chain is not explicitly managed by the OpenC2 specifications, a rule is inserted into this chain for both ingress and egress commands.
The actuator then determines the correct position for the new rule among the existing active rules in the specific iptables chain. This step is essential because iptables processes rules using a **first-match** approach: packets are evaluated against the rules in the order they are listed and the first matching rule determines the action taken. Placing the rule in the correct position ensures that it has the intended effect, allowing more specific rules to be evaluated before more general ones, which guarantees the proper functioning of the firewall.
Next, if the command target is of type *ipv4_connection* or *ipv6_connection*, the actuator sets the source and destination addresses, the protocol and the source and destination ports accordingly. For targets of type *ipv4_net* or *ipv6_net*, the specified address, used as the destination address as explained in Subchapter 5.4, is applied.
Finally, the actuator specifies the iptables target to be applied: ACCEPT for an allow action, DROP for a deny action or REJECT for a deny action with the *drop_process* argument set to *reject*. Once the command is fully constructed, SLPFActuator_iptables executes it via the command line.
If the OpenC2 command specifies a direction of *both*, separate iptables commands are constructed and executed to insert the rule into the corresponding custom input and output chains, as well as into the forward chain.

5.9.3 Delete action

The SLPFActuator_iptables handles the delete action by overriding the corresponding method of the SLPF Actuator Manager, allowing the removal of specific rules within the iptables environment. As described in Subchapter 5.5, the actuator receives from the SLPFActuator the OpenC2 command specifying the rule to be deleted. As with the allow and deny actions, no custom data is required for this operation and thus it is not retrieved from the SLPFActuator.
In iptables, a rule within a chain can be removed by specifying its exact match using the ‘-D’ option. This approach is adopted because rule numbering in iptables is dynamic and cannot be reliably tracked by an external program. To implement deletion, the actuator constructs an iptables command to be executed from the command line in much the same way as for the allow and deny actions and described in Subchapter 5.9.2, but with the addition of the ‘-D’ option. The OpenC2 command received from the SLPFActuator is used to determine the rule details. Finally, the constructed iptables command is executed.
If the OpenC2 command specifies a direction of *both*, separate iptables commands are constructed and executed to delete the rule from the corresponding custom input and output chains, as well as from the forward chain.

5.9.4 Update action

The SLPFActuator_iptables handles the update action by overriding the corresponding method of the SLPF Actuator Manager, allowing the content of a rule file to be applied within the iptables environment. As specified in Subchapter 5.6, the actuator receives from the SLPFActuator the *name* and the *path* of the file to be used for the update action.
The filtering rules contained in this file are used to update the corresponding system rule file managed by iptables, located in the directory specified by the *iptables_rules_directory_path* parameter, obtained during iptables initialization, with the filename *iptables_rules_v4_filename* or *iptables_rules_v6_filename*, depending on the extension of the file received as the target of the OpenC2 command.
During the backend verification phase, which takes place before the action is actually scheduled by the SLPF Actuator Manager, SLPFActuator_iptables verifies that the file specified as the target of the update command has a supported extension, either '.v4' or '.v6'. If the file extension is invalid, the actuator returns a response to the producer with status code 400 (Bad Request).
To perform the update action, SLPFActuator_iptables first examines the extension of the file specified in the OpenC2 command. If the file has a '.v4' extension, the update is applied to the file designated by *iptables_rules_v4_filename,* if the file has a '.v6' extension, the update targets the file identified by *iptables_rules_v6_filename*. Once the appropriate file is selected, its content is updated and loaded into the iptables environment, activating the filtering rules contained within.

5.10 SLPF Actuator for OpenStack

To manage **OpenStack Security Groups** and their associated **Security Rules**, a dedicated subclass of the SLPFActuator has been implemented, named **SLPFActuator_openstack**.
In the OpenStack environment, SLPF Commands are enforced through the dynamic creation of **dedicated Security Groups**, each associated with the specific OpenStack ports referenced by the IPv4/IPv6 subnet or connection targets in the command.
The OpenStack networking service enables the creation and deletion of security rules of type allow, but it does not support the explicit definition of deny rules, as its security model is based on a default deny policy. For this reason, **the deny action is not implemented** and when the actuator receives a deny command it returns a response with status code 501 (Not Implemented) to the producer.
Security rules in OpenStack are **natively persistent**, which means that the OpenStack networking service, unlike iptables, does not require the management of external files to preserve the current rule set across system restarts. Consequently, this actuator does not override the SLPFActuator methods responsible for saving and restoring filtering rules upon reboot.
Moreover, since the OpenStack networking service does not support updating the active filtering rules with the content of an external file, **SLPFActuator_openstack does not implement the update action**.

When an update command is received, a response with status 501 (Not Implemented) is sent back to the producer. Similarly, the actuator does not override the method used by the SLPF Actuator Manager to delete all active filtering rules when switching between DB mode and FILE mode (or vice versa), since this actuator **does not support the FILE mode** due to the absence of the update action.
Because of these unimplemented features, SLPFActuator_openstack overrides the query action to accurately report the action-target pairs that are actually supported by this actuator.

5.10.1 SLPFActuator_openstack Initialization

Similarly to the previous actuator, the initialization of SLPFActuator_openstack is entirely managed through a configuration file that specifies all the parameters required for the actuator's setup. The configuration file includes all parameters needed to correctly initialize the SLPF Actuator Manager. The most important of these parameters is *asset_id*, which is set to *openstack* if not specified in the configuration file. In addition, the file defines several parameters specific to the OpenStack environment:

- **environment_variables_file** specifies the absolute path to the file containing the environment variables required to establish a connection with OpenStack.
- **project_name** defines the name of the OpenStack project to be associated with the SLPFActuator_openstack.
- **security_group_base_name** specifies a standard prefix for constructing new security group names.
- **security_group_base_description** specifies a standard description for constructing new security group.

The *environment_variables_file* and *project_name* parameters are mandatory, as they are essential for the actuator's correct operation. If either of these parameters is missing, the initialization process is immediately terminated with an error. Otherwise, if *security_group_base_name* or *security_group_base_description* are not provided, default values are automatically assigned to them.
A key part of the initialization process consists of establishing a connection with the OpenStack environment. During this phase, the environment variables defined in the file specified by *environment_variables_file* are loaded into the operating system. These variables are then used to authenticate and establish an authorized connection to OpenStack.
Subsequently, the actuator retrieves from OpenStack the project identified by the **project_name** parameter. If the specified project cannot be found, the initialization process is aborted and an error is returned.
Finally, SLPFActuator_openstack invokes the initialization method of the SLPF Actuator Manager, completing its own setup phase.

5.10.2 Allow action

The SLPFActuator_openstack supports the allow action by overriding the corresponding method of the SLPFActuator to enable the creation of an OpenStack security rule.
A fundamental aspect of the SLPFActuator_openstack's behavior when handling an allow command occurs during the backend verification phase, before the action is actually scheduled by the SLPF Actuator Manager. In this phase, the actuator provides the SLPFActuator with the necessary custom data for the future execution of the allow command: specifically, the ID of the security group where the rule controlling ingress traffic will be inserted (if the *direction* argument is *ingress* or *both*) and the ID of the security group where the rule controlling egress traffic will be inserted (if the *direction* argument is *egress* or *both*). To determine these IDs, the actuator first identifies all relevant OpenStack ports (network interfaces) that will be affected by the command. For targets of type *ipv4_connection* or *ipv6_connection*, if the direction is ingress or both, the actuator selects the ports whose fixed IP address matches the destination address of the target. Otherwise, if the direction is egress or both, the ports whose fixed IP address matches the source address of the target are selected. In the case of targets of type *ipv4_net* or *ipv6_net*, which specify only a destination network as described in Subchapter 5.4, all ports with IPs included in the specified network are selected for an ingress command, while for an egress command all ports attached to the OpenStack project defined in the configuration file are included.
If no ports within the OpenStack project match the command's selection criteria, a response with status code 400 (Bad Request) is returned to the producer.
Once the relevant ports have been identified, the actuator searches within the project for a security group named as the concatenation of the *security_group_base_name* parameter (obtained during initialization) and the IP address identifying the selected ports. If such a security group does not exist, it is created with this name and a description equal to the *security_group_base_description* parameter defined during initialization. If the security group already exists, the actuator verifies that it does not already contain a security rule identical to the one specified in the received command. If a matching rule is found, a response with status code 400 (Bad Request) is returned to the producer, as OpenStack does not allow the insertion of duplicate rules within the same security group.
Once the security groups are found or created, their IDs can be returned to the SLPF Actuator Manager to be stored as custom data.

When the command is scheduled for execution, the SLPFActuator_openstack receives from the SLPFActuator the OpenC2 command target, the direction and the custom data, as described in Section 5.4. The OpenC2 command is translated into an OpenStack security rule:

- The **security_group_id** is set to the value of the *ingress_id* from the custom data for directions ingress or both or to *egress_id* for directions egress or both.
- The **direction** attribute of the security rule is set accordingly.
- The **ethertype** is set based on the type of the target: 'IPv4' for *ipv4_net* or *ipv4_connection* and 'IPv6' for *ipv6_net* or *ipv6_connection*.

- The **remote_ip_prefix** defines the source address of the traffic for ingress security rules or the destination address for egress security rules. For targets of type *ipv4_connection* or *ipv6_connection*, it takes the value of the source address if the direction is ingress or the value of the destination address if the direction is egress. For targets of type *ipv4_net* or *ipv6_net*, it is set to the value specified in the target itself for egress commands, while for ingress commands it is set to '0.0.0.0/0' for IPv4 or '::/0' for IPv6.
- In case of *ipv4_connection* or *ipv6_connection* target, the **protocol** field is set to the value specified in the OpenC2 target.
- The **port range** is defined, for targets of type *ipv4_connection* or *ipv6_connection*, based on the source or destination port and represented as a single port in the security rule.

After mapping all the relevant parameters, the security rule is created.
Finally, all ports in the project that are affected by the OpenC2 command and not yet attached to the security group to which the new rule was added, are associated with it.
If an OpenC2 command specifies a direction of *both*, two distinct security rules are created and inserted into the respective security groups for ingress and egress traffic.

5.10.3 Delete action

The SLPFActuator_openstack implements the delete action by overriding the corresponding method of the SLPF Actuator Manager, allowing the removal of security rules within security groups. As described in Subchapter 5.5, the actuator receives from the SLPFActuator the OpenC2 command to be deleted, along with the custom data containing the IDs of the security groups where the security rules were inserted.
First, the ID of the security rule corresponding to the OpenC2 command is retrieved. If the security group contains only the security rule to be deleted, all ports attached to the security group are disconnected from it and the security group is removed; otherwise, only the specific security rule is deleted.
If the command to be deleted specifies a direction of both, both previously created security rules are removed from their respective security groups.

5.11 SLPF Actuator for Kubernetes

In order to manage **Kubernetes Network Policies**, a dedicated subclass of the SLPFActuator has been implemented, namely **SLPFActuator_kubernetes**.
In the Kubernetes environment, SLPF Commands are enforced through the dynamic creation of dedicated **Labels**, which are applied to the pods selected by the IPv4/IPv6 subnet or connection targets in the command and referenced by the corresponding Network Policies generated to implement the required filtering behavior.
The Kubernetes network policy framework allows the creation and deletion of network policies of type allow, but it does not provide support for explicit deny policies, as Kubernetes enforces a default deny behavior. Consequently, **the deny action is not implemented** and if the actuator receives a deny command, it responds to the producer with status code 501 (Not Implemented).

Network policies in Kubernetes are **persistent**, meaning that the Kubernetes network policy framework, like OpenStack, does not rely on external files to maintain the current set of policies across system restarts. As a result, this actuator does not override the SLPFActuator methods used for saving and restoring filtering rules after a reboot.
SLPFActuator_kubernetes correctly **implements the update action**, as it is able to refresh the active network policies within a given namespace using the contents of a **YAML file**. This capability allows the actuator to fully support the FILE mode of the SLPF Actuator Manager and therefore it overrides the method used by the SLPFActuator to remove all active filtering rules when switching from DB mode to FILE mode (or vice versa).
However, due to the lack of support for the deny action, SLPFActuator_kubernetes overrides the query action in order to provide the producer with the pairs of action-target combinations that are actually implemented.

5.11.1 SLPFActuator_kubernetes Initialization

As with the other implementations, the initialization of SLPFActuator_kubernetes is fully managed through configuration files. These files define the parameters required to properly initialize the SLPF Actuator Manager, the most important of which is *asset_id*, set to *kubernetes* if not specified in the configuration file. Additional parameters specific to the Kubernetes environment are also included:

- **config_file:** specifies the absolute path to the Kubernetes configuration file.
- **kube_context:** defines the name of the Kubernetes context to be used.
- **namespace:** indicates the namespace where the network policies will be managed.
- **subnet_base_label_key:** defines the prefix used to create labels associated with pods and network policies.
- **generate_name:** specifies the prefix used to generate unique names for newly created network policies.

None of these parameters are strictly required for the actuator's initialization, default values are automatically assigned to any parameter not provided.
A key aspect of the Kubernetes actuator initialization process is establishing a connection to the Kubernetes cluster. During this phase, the specified configuration file and context are loaded, the Kubernetes API clients are instantiated and the namespace defined in the configuration file is created if it does not already exist.
Finally, SLPFActuator_kubernetes calls the initialization method of the SLPF Actuator Manager, marking the completion of its own initialization process.

5.11.2 Allow action

The SLPFActuator_kubernetes supports the allow action by overriding the corresponding method of the SLPF Actuator Manager to enable the creation of new Kubernetes Network Policies. As previously explained for OpenStack in Subchapter 5.10.2, a key part of processing an allow command takes place during the validation phase handled by the specific implementation. In this phase, SLPFActuator_kubernetes provides the SLPF Actuator Manager with the custom data to be stored in the database, which are required for the future execution of the command.

In the case of Kubernetes, these custom data define the labels to be associated with pods and network policies. These labels are generated by concatenating the *subnet_base_label_key* parameter, received during the actuator's initialization, with the IP address identifying the Kubernetes subnet affected by the received allow command. The subnet itself is determined using the same logic described in Subchapter 5.10.2 for the OpenStack actuator. Once the labels have been created, they are associated with the pods that must enable the corresponding network policy and the custom data are returned to the SLPF Actuator Manager for storage in the database. If no containers within the namespace match the command's selection criteria, a response with status code 400 (Bad Request) is returned to the producer.
During this validation phase, the actuator also checks that, in the case of targets of type *ipv4_connection* or *ipv6_connection*, the protocol value is TCP, UDP or SCTP, as these are the only protocols supported by Kubernetes.

When the allow action is scheduled for execution, SLPFActuator_kubernetes receives from the SLPF Actuator Manager the *target* of the command, the *direction* argument and the custom data, as described in Subchapter 5.4.
The OpenC2 command is translated into a Kubernetes NetworkPolicy object, where the main fields are set as follows:

- **podSelector,** set to the label defined by the *ingress_label* parameter in the custom data when the direction is ingress or both, or by the *egress_label* parameter when the direction is egress or both.
- **policyTypes,** set according to the direction specified in the OpenC2 command (Ingress or Egress).
- **ipBlock,** defines the IP address or subnet representing the source of ingress traffic or the destination of egress traffic, following the same logic used for the OpenStack *remote_ip_prefix* parameter described in Subchapter 5.10.2.
- **protocol,** set according to the protocol specified in the target, in the case of *ipv4_connection* or *ipv6_connection* targets.
- **port,** set, in the case of *ipv4_connection* or *ipv6_connection* targets, to the value defined in the source or destination port parameter, depending on the direction of the OpenC2 command.

Finally, the network policy is created within the namespace specified by the *namespace* parameter obtained during initialization, with a name partially composed using the *generate_name* prefix, also defined in the configuration file.
If the OpenC2 command specifies a direction of both, two distinct network policies are created: one controlling ingress traffic and the other controlling egress traffic.

5.11.3 Delete action

The SLPFActuator_kubernetes implements the delete action by explicitly overriding the corresponding method of the SLPF Actuator Manager, enabling the proper removal of Kubernetes Network Policies. As described in Subchapter 5.5, the actuator receives from the SLPFActuator the OpenC2 command to be deleted along with its associated custom data.
The actuator first locates the name of the Network Policy corresponding to the OpenC2 command to be removed. Once identified, the NetworkPolicy is deleted. If the deleted NetworkPolicy was the last one associated with its corresponding label, that label is subsequently removed from all pods that were influenced by it.
If the OpenC2 command specifies a direction value of both, both Network Policies previously created to allow the corresponding ingress and egress traffic are deleted.

5.11.4 Update action

The SLPFActuator_kubernetes implements the update action by overriding the corresponding method of the SLPF Actuator Manager, allowing the replacement of the active network policies within the namespace with those defined in a YAML file provided as the target of the command. As specified in Subchapter 5.6, the actuator receives from the SLPFActuator the *name* and *path* of the target file.
During the backend verification phase, which occurs before the action is scheduled by the SLPF Actuator Manager, SLPFActuator_kubernetes checks whether the file specified as the target of the update command has a valid and supported extension (‘.yaml’).
If the file extension is not supported, the actuator responds to the producer with status code 400 (Bad Request).
To correctly perform the update action, the SLPFActuator_kubernetes first deletes all existing network policies in the namespace, since they will be replaced by the new ones defined in the YAML file. Subsequently, all network policies contained in the file are loaded into the Kubernetes environment.

5.12 SLPF Actuator for MS Azure

To manage **Azure Network Security Groups (NSGs)** and their related **security rules**, a dedicated subclass of the SLPFActuator has been implemented, named **SLPFActuator_azure**.
Azure supports both the insertion and deletion of allow and deny rules, however, it does not support customizing the response to denied traffic. Consequently, the *drop_process* argument of the deny command is not supported by this actuator.
Security rules in Azure are **natively persistent**, meaning that the Azure networking model automatically preserves rule configurations across system restarts or infrastructure changes. As a result, this actuator does not override the SLPFActuator methods responsible for saving and restoring filtering rules upon reboot.
Since Azure does not support updating NSG security rules or configurations from external files, SLPFActuator_azure **does not implement the update action.**

When an update command is received, a response with status 501 (Not Implemented) is returned to the producer. Likewise, the actuator does not redefine the method used by the SLPF Actuator Manager to clear all active filtering rules when switching between DB mode and FILE mode (or vice versa), since the absence of the update action implies that the FILE mode is not supported.
Because of these design constraints, SLPFActuator_azure redefines the query action to accurately report the action-target pairs that are effectively supported by the actuator.

5.12.1 SLPFActuator_azure Initialization

As with the previous actuators, the initialization of the Azure actuator is entirely managed through a configuration file. In addition to the parameters required to correctly initialize the SLPF Actuator Manager, the most important of which is *asset_id*, set to *azure* if not explicitly specified, the configuration file defines several Azure-specific settings:

- **authentication_file**, which specifies the absolute path to the authentication file containing the parameters required to establish a connection with the Azure environment, namely *tenant_id*, *client_id*, *client_secret* and *subscription_id*;
- **resource_group_name**, which indicates the name of the Azure Resource Group containing the Network Security Groups to be managed;
- **network_security_group_name**, which specifies the name of the Network Security Group where filtering rules will be created or removed.

Among these, only *authentication_file* is mandatory for the correct operation of the actuator, while default values are automatically assigned to the other parameters if not provided.
As in the OpenStack and Kubernetes actuators, an essential part of the initialization process consists of establishing a connection with the Azure environment.
During this phase, the authentication information specified in *authentication_file* is used to obtain the credentials required to access Azure services and to initialize both a Resource Management client, responsible for managing Azure resource groups, and a Network Management client, which handles network-related resources such as Network Security Groups. The actuator then verifies the existence of the Resource Group specified by *resource_group_name*, creating it if it does not already exist. Likewise, it searches for a Network Security Group matching the name defined by *network_security_group_name* and creates it when necessary.
Finally, the initialization of the SLPF Actuator Manager is performed, marking the completion of the setup process.

5.12.2 Allow and deny actions

The SLPFActuator_azure implements the *allow* and *deny* actions by overriding the corresponding methods of the SLPF Actuator Manager, enabling the management of Azure Network Security Group (NSG) rules.
As described in Subchapter 5.4, this actuator receives from the SLPFActuator the *target* of the allow or deny command, the *direction* argument and, in the case of a deny command, the *drop_process* argument, although the latter has no practical effect since it is not supported by the Azure platform.

As in the case of the iptables implementation, the Azure actuator does not require custom data to execute its actions and therefore no additional data is retrieved from or stored in the database.
When executing an allow or deny action, SLPFActuator_azure first converts the received OpenC2 command into a valid Azure Network Security Rule. The mapping between OpenC2 parameters and Azure rule fields is performed as follows:

- **Action**, set according to the OpenC2 action (*allow* or *deny*).
- **Direction,** configured as *Inbound* if the OpenC2 argument direction is *ingress* or *both*, or as *Outbound* if it is *egress* or *both*.
- **Source** and **destination addresses,** populated with the IP addresses specified in the OpenC2 target.
- **Protocol,** configured based on the protocol defined in the OpenC2 target, when the target type is *ipv4_connection* or *ipv6_connection*.
- **Source** and **destination port ranges,** configured according to the port values specified in the OpenC2 target when applicable (*ipv4_connection* or *ipv6_connection*).

As explained in Subchapter 3.4.1, within a single NSG each Azure Network Security Rule must have a unique name and the priority of rules must be unique within each traffic direction. For this reason, each rule is assigned a unique name and priority value.
To determine an appropriate priority, all existing rules in the NSG are analyzed and the new rule is inserted at the position that best reflects its level of specificity. When necessary, the priorities of subsequent rules are adjusted accordingly to preserve the proper evaluation order.
Finally, the newly created security rule is added to the Network Security Group.
If the OpenC2 command has its direction set to both, two separate rules are created, one handling inbound traffic and the other handling outbound traffic.

5.12.3 Delete action

The SLPFActuator_azure implements the delete action by overriding the corresponding method of the SLPF Actuator Manager, allowing the proper removal of Azure Network Security Rules.
As explained in Subchapter 5.5, the actuator receives from the SLPFActuator the OpenC2 command to be deleted. Similar to the allow and deny actions, custom data is not required for this operation and therefore is not retrieved from the SLPFActuator.
The actuator first identifies the Azure Network Security Rule corresponding to the OpenC2 command to be deleted. Once located, the rule is removed from the Network Security Group.
If the OpenC2 command has its direction set to both, the actuator deletes both rules that were previously created to control inbound and outbound traffic, ensuring that all traffic affected by the command is properly removed.

6 Implementation and validation

This chapter presents the practical aspects related to the implementation of the SLPF Actuator Manager and its validation.
The first part focuses on the software tools, frameworks and libraries employed to support both the management of the SLPF Actuator Manager and its interaction with different network filtering technologies. Specifically, it describes the Python libraries chosen for handling the OpenC2 environment (**Otupy**), database management (**SQLite3**), task scheduling (**APScheduler**) and integration with OpenStack (**OpenStackSDK**), Kubernetes (**Kubernetes Python Client**) and Microsoft Azure (**Azure SDK for Python**).
iptables, by contrast, is controlled via direct command-line calls through Python's subprocess module. Since its management does not involve dedicated libraries, it is not discussed further in this chapter.
The second part presents the **validation** phase, which includes a comprehensive testing and evaluation process designed to verify both the syntactic and semantic correctness of the implementation and its operational behavior. The conducted tests aim to assess the correctness of command execution, the consistency of results across heterogeneous platforms and the overall performance of the system in terms of responsiveness and reliability.
Together, these sections demonstrate the feasibility and robustness of the proposed implementation, confirming its compliance with the OpenC2 SLPF Specification [6] and its effectiveness as a unified control interface for diverse network protection systems.

6.1 Otupy

Otupy [12] is an open-source Python framework designed to provide a flexible, extensible and portable implementation of the OpenC2 standard. Its main purpose is to simplify the creation, transmission and parsing of OpenC2 messages, while offering a solid foundation for developing and integrating security tools.
The strength of Otupy lies in its modular architecture, which decouples the core components: profiles, data types, serialization formats and transport protocols can be extended or modified without affecting the core library. This is achieved through a modular communication stack that allows different transport protocols and serialization formats to be combined seamlessly, along with an inheritance-based model for data types and meta-serialization, which ensures both consistency and flexibility in message handling.

The overall architecture and workflow of Otupy are shown in Figure 6.1.

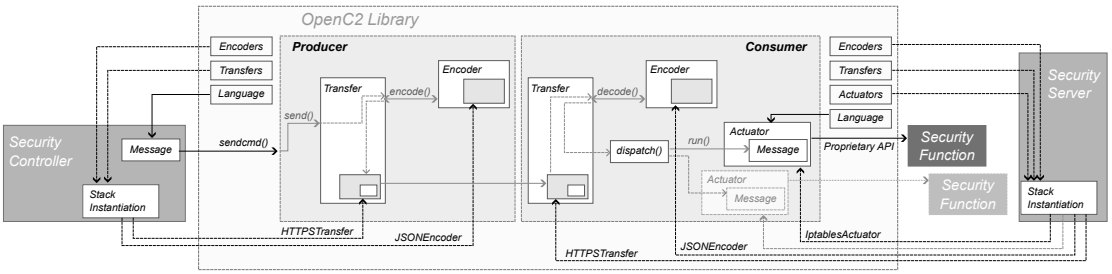

Fig. 6.1 Otupy's architecture and workflow [12]

The framework provides an API closely aligned with the OpenC2 grammar, supporting both the data models defined in the standard and common data types used in networking and security applications. The API is fully agnostic of serialization and transport details, making it intuitive to learn and practical to use.
Operationally, the main components are the Producer and Consumer, implementing the sending and receiving roles of OpenC2 messages. Both rely on Encoder and Transfer interfaces for message serialization, such as JSON, and delivery over protocols, such as HTTP.
Actuators translate OpenC2 messages into actual security commands, like iptables rules or Kubernetes network policies, and return execution results.
Otupy implements all objects described in the OpenC2 Language Specification [2] (Message, Command, Response, targets and data types), relying on standard Python types and widely-used libraries for numbers, strings, IP addresses and other commonly used types, while also providing a full definition of the SLPF profile, including all its actions, targets, arguments and results [6].
Finally, Python ensures portability across platforms, supports reflective programming and allows easy integration of new extensions such as profiles, encoders, transfers and actuators. The framework has been extensively validated, demonstrating full OpenC2 compliance and facilitating the development of interoperable tools in heterogeneous environments.

6.2 SQLite3

For managing the internal database of the SLPF Actuator Manager, the SQLite3 library was selected.
SQLite [13] is a lightweight, open-source relational database engine that stores data locally in a single file, without requiring the deployment or maintenance of a separate database server. Its simplicity and integration within the Python standard library make it an ideal choice for applications where ease of use, portability and low overhead are essential. Despite its lightweight design, SQLite provides full support for standard SQL syntax, transactions and data integrity constraints, ensuring reliable and consistent data management.

SQLite is fully self-contained, meaning that it does not depend on external libraries or services, which greatly reduces installation and configuration complexity. It supports atomic commit and rollback operations, allowing safe execution of multiple operations even in the event of a system failure. The database engine is thread-safe and can handle concurrent read operations efficiently, while write operations are serialized to maintain data integrity.
Additionally, SQLite3 allows for flexible schema design and easy adaptation to changing data requirements, making it suitable for managing diverse types of internal state information. Moreover, its Python interface exposes a simple API for executing SQL queries, fetching results and handling transactions.
Overall, SQLite3 combines robustness, simplicity and portability, aligning well with the goals of maintaining a compact and self-contained system architecture.

6.3 APScheduler

For scheduling OpenC2 commands at predetermined times, the Python APScheduler library was selected.
APScheduler [14] is a lightweight, open-source framework that provides advanced scheduling capabilities directly within Python applications, allowing jobs to be executed periodically, at specific dates or according to complex recurring rules.
One of its main advantages is flexibility: APScheduler supports multiple scheduling strategies, including date-based scheduling for single events, interval-based for recurring tasks and cron-style scheduling for complex periodic execution. This makes it suitable for a wide range of applications, from simple automation scripts to complex background services.
Another key benefit of APScheduler is its ability to run tasks in the background without blocking the main program execution. It offers multiple execution models, including thread-based and process-based job stores, which allows concurrent task execution while maintaining application responsiveness.
Additionally, APScheduler allows jobs to be configured to run exclusively, ensuring that only one instance of a task is executed at a time. This feature helps prevent race conditions and potential conflicts, providing greater control and reliability in task scheduling.
The library also provides a user-friendly API for adding, modifying or removing scheduled jobs dynamically, along with built-in error handling and logging features. These capabilities simplify monitoring, debugging and maintaining scheduled tasks, reducing the risk of missed executions or system failures.
Overall, APScheduler offers a robust and flexible solution for task scheduling in Python applications, combining ease of use with powerful features for managing periodic or time-specific execution.

6.4 OpenStackSDK

To efficiently manage OpenStack and its associated security groups and rules, the Python library OpenStackSDK was chosen.
OpenStackSDK [15] is the official Python library that provides a high-level and programmatic interface for interacting with OpenStack services.
By abstracting the complexity of direct API calls, it allows developers to manage OpenStack resources using Python objects and methods, avoiding the need to construct raw HTTP requests manually. This abstraction simplifies development, reduces the risk of errors and ensures that operations are executed consistently and predictably.
A key application of OpenStackSDK is the management of networking resources, particularly **security groups** and **security rules**. Through the library, it is possible to create, retrieve, modify and delete security groups, as well as define or delete rules that control the inbound and outbound traffic of instances. These capabilities provide a reliable and programmatic way to enforce network policies, ensuring that access control and isolation requirements are consistently applied across the OpenStack environment.
The library also includes built-in mechanisms for **error handling and exception management**, returning clear and structured exceptions when API operations fail. This simplifies debugging and allows developers to implement robust error recovery procedures. OpenStackSDK efficiently handles large datasets by supporting **automatic pagination** and **query filtering**, which is especially useful when retrieving extensive lists of resources.
Additionally, the library manages **authentication tokens and sessions** automatically, reducing the complexity associated with credential management.
Another important feature of OpenStackSDK is its **object-oriented design**. All OpenStack resources are represented as Python objects with dedicated methods for common operations, such as create, modify and delete. This approach enhances code readability and maintainability, making it easier to develop, extend and automate complex workflows.
Overall, OpenStackSDK is a robust, reliable and developer-friendly tool for managing OpenStack resources. Its combination of simplified API interaction, structured error handling, support for large datasets, automated session management and object-oriented resource representation makes it particularly valuable for automating security group configuration and enforcing security rules in dynamic cloud environments.

6.5 Kubernetes Python Client

To efficiently manage Kubernetes resources, such as namespaces and network policies, the official **Kubernetes Python client** [16] was selected.
This library provides a high-level, programmatic interface that abstracts the complexity of direct API interactions, enabling developers to manipulate Kubernetes objects through Python classes and methods rather than constructing raw HTTP requests.
The library is **organized into multiple API clients**, each responsible for a specific group of resources.

For example, *CoreV1Api* manages fundamental objects such as Pods, while *NetworkingV1Api* handles Network Policies.
Each Kubernetes resource type is represented by a **dedicated Python class**, with attributes that correspond directly to the fields of the API objects. These classes allow developers to create new resources in memory, modify existing ones and then pass them to the appropriate API client methods to perform create, retrieve or delete operations in the cluster. This object-oriented approach improves code readability, ensures proper structuring of resource definitions and eliminates the need to manually construct JSON or YAML payloads for API requests.
Authentication is managed by the library, allowing developers to connect to a cluster using either a configuration file (kubeconfig) or a context explicitly specified in code. This ensures secure access to the Kubernetes API and simplifies the management of credentials and connection settings, enabling seamless communication with clusters without requiring manual handling of tokens or certificates.
A key advantage of the official client is its active maintenance in alignment with Kubernetes releases, providing support for new features while properly handling deprecated APIs. Additionally, the library offers robust error handling, detailed exception reporting and logging capabilities, which simplify monitoring, debugging and the reliable execution of automated tasks.
By leveraging the official Kubernetes Python client, developers can manage network policies and other cluster resources in a programmatic, scalable and robust manner, while maintaining consistency with Kubernetes API standards and best practices.

6.6 Azure SDK for Python

To implement support for the Microsoft Azure environment, including management of Resource Groups and Network Security Groups, the official **Azure SDK for Python** [17] was adopted. This library provides a rich, high-level interface to interact programmatically with Azure services, allowing developers to manage cloud resources using Python constructs instead of manually handling REST API calls.
At its core, the SDK enables creation, retrieval, updating and deletion of fundamental Azure components. For instance, it supports the provisioning and modification of resource groups via the *ResourceManagementClient* and the management of network security groups (NSGs) through the *NetworkManagementClient.* These capabilities make it possible to enforce network protection policies and organizational resource structure within Azure subscriptions in a consistent and automated way.
One of the main advantages of the Azure SDK for Python is its integration with **Azure's authentication and identity framework**. The library supports multiple authentication mechanisms, including *DefaultAzureCredential*, *ManagedIdentityCredential* and *ClientSecretCredential*, allowing flexible use across both interactive and automated environments. Among these, *ClientSecretCredential* is commonly used for service applications and background processes, as it enables secure authentication to Azure Active Directory using a tenant ID, client ID and client secret associated with a registered application. Once authentication is established, the SDK automatically manages token acquisition

and renewal, and the serialization of resource definitions into the appropriate JSON payloads for interaction with Azure APIs.
Furthermore, the SDK provides dedicated **operation groups** for managing security rules within Network Security Groups. Through these interfaces, it is possible to list existing rules, create new ones, modify their properties or remove obsolete entries, all via structured API calls. By encapsulating these operations within well-defined client classes and asynchronous operation pollers, the SDK improves the reliability and maintainability of automation workflows.
Overall, the Azure SDK for Python provides a robust and maintainable foundation for automating the management of resources and network security configurations within Azure environments. Its high-level abstractions, integration with Azure's identity and resource management services, and comprehensive support for network security operations make it a reliable and versatile tool for developing scalable cloud automation solutions.

6.7 Validation

This section presents the **validation phase** of the developed system, aimed at verifying the correctness, functionality and performance of the implemented SLPF Actuator Manager.
The *SLPFActuator_azure*, responsible for managing Microsoft Azure environment, could not be tested due to the absence of an available Azure infrastructure suitable for validation.
The testing process follows a three-phase approach: **syntax and semantic validation**, **functional testing** and **performance evaluation**.
The first phase focuses on ensuring that OpenC2 commands are properly formatted, interpreted and processed in full compliance with the OpenC2 Language [2] and SLPF [6] Specifications.
The second phase evaluates the actual behavior of the actuators, verifying that each command produces the expected network effect on the corresponding firewall.
Finally, the third phase measures the responsiveness and efficiency of the system through latency and execution time analysis, providing insight into both communication overhead and internal processing performance.
All tests were conducted using the **pytest** framework, a widely adopted Python testing tool that supports automated test discovery, execution and reporting.

6.8 Syntax and semantic validation

The syntax and semantic validation of the SLPF Actuator Manager was performed following an approach closely aligned with the one adopted for the Otupy framework [12], while extending it to address all specific aspects of the SLPF profile.
As in the case of Otupy, the JSON representation of OpenC2 *Commands* and *Responses* serialized over HTTP was used as the reference format for validation.

The input set for testing was constructed by generating all possible OpenC2 Commands that could be issued to the Actuator Manager. For each supported action (*allow*, *deny*, *delete*, *update* and *query*) every valid combination of targets and arguments was considered. For targets composed of multiple parameters, all possible combinations of parameter values were tested, while for arguments that can assume multiple valid values, every possible variation was included.
The resulting commands were categorized into **valid** and **invalid** cases. Valid commands are expected to be correctly processed and executed, whereas invalid ones should trigger appropriate error responses.
For the *allow* and *deny* actions, commands were considered invalid when a target of type *ipv4_connection* or *ipv6_connection* specified a source or destination port without indicating the corresponding protocol, when *start_time*, *stop_time* and *duration* arguments were used simultaneously, or when the *insert_rule* argument was present without the *response_requested* argument, in accordance with the SLPF Specification [6].
Commands specifying a *start_time* later than *stop_time* were also treated as invalid.
For the *update* action, commands were marked as invalid if the target did not include the required *name* parameter, as specified in the SLPF Specification [6], or contained invalid values for *name*, *path* or *hashes* parameters.
Table 6.1 summarizes, for each action, the total number of generated commands along with the corresponding counts of valid and invalid cases.

Action	Total commands	Valid commands	Invalid commands
Query	15	15	0
Allow	217	142	75
Deny	347	228	119
Delete	3	3	0
Update	35	17	18

Table 6.1 Valid and invalid generated commands per action

Semantic validation was conducted by deserializing and serializing each command, both valid and invalid, and comparing the serialized representation with the original JSON message to ensure full structural and semantic consistency.
Syntactic validation, on the other hand, was carried out through an actual communication exchange between a producer and a consumer. All generated commands, and related responses, were transmitted over HTTP and the resulting message payloads were validated against JSON schemas obtained from a third-party repository.
Additionally, for *allow* and *deny* actions containing the *insert_rule* argument, a duplicate command with the same rule number was transmitted to verify that the system correctly rejects attempts to insert multiple filtering rules with identical identifiers, as required by the SLPF Specification [6].

For the transmission of invalid delete commands, the same set of valid delete commands was used, but with the corresponding filtering rule (identified by the *rule_number* specified in the delete command) absent from the database, ensuring that the system correctly handles attempts to delete non-existent rules.
At the end of the syntax validation phase, cleanup operations were performed to restore both the Actuator Manager database and the tested backend implementation to a state free of active filtering rules.
This entire validation process was executed for all tested backend implementations: **iptables**, **OpenStack** and **Kubernetes**. Table 6.2 summarizes the results of semantic and syntactic validation tests, which were consistent across all evaluated actuators as a response with status 501 (Not Implemented), returned for unsupported actions or protocols, was considered valid, ensuring that all actuators were evaluated in a comparable manner.

Test	Commands	Success rate
Decoding commands	617	100%
Encoding commands	617	100%
Sending valid query	15	100%
Sending invalid query	0	-
Sending valid allow	142	100%
Sending invalid allow	75	100%
Sending valid deny	228	100%
Sending invalid deny	119	100%
Sending valid delete	3	100%
Sending invalid delete	3	100%
Sending valid update	17	100%
Sending invalid update	18	100%

Table 6.2 Semantic and syntactic test results

6.9 Functional testing

To evaluate the effectiveness and correctness of the SLPF Actuator Manager, a series of functional tests were conducted to observe how ingress and egress network traffic is affected by the application of *allow*, *deny* and *delete* commands.
These tests were designed to verify that each implemented actuator correctly translates SLPF commands into concrete enforcement actions in its respective environment.
Given the heterogeneous nature of the supported implementations, dedicated test environments were set up for each developed actuator to accurately reflect their operational characteristics.

Traffic between the two nodes of each environment was generated using **hping3**, a versatile packet-crafting tool widely used for testing network behavior. hping3 allows the construction of custom TCP, UDP and ICMP packets, making it particularly suitable for validating the effect of filtering rules under controlled conditions.
To capture the traffic produced during each test, **tcpdump** was employed. tcpdump is a lightweight command-line packet sniffer capable of recording raw network frames directly from a network interface. Its minimal overhead and high reliability make it ideal for collecting detailed traces even in constrained or remote environments such as cloud-based virtual machines. The resulting packet captures provide a faithful record of all inbound and outbound traffic during each experiment.
Finally, the captured traces were analyzed using **Wireshark**, a graphical network analysis tool that offers powerful filtering, decoding and visualization capabilities. Wireshark enables a fine-grained inspection of packet flows, making it straightforward to determine whether traffic was allowed or blocked as a result of the applied rules. By combining tcpdump's capture capability with Wireshark's analysis tools, the evaluation achieves both precision and clarity in verifying the functional behavior of each actuator.
This functional-testing methodology ensures that the SLPF Actuator Manager is validated not only in terms of syntactic and semantic correctness, but also in its real-world ability to influence live network traffic across heterogeneous enforcement environments.

6.9.1 iptables

To evaluate the functionality of the *SLPFActuator_iptables,* two virtual machines, **VM1** (IP address *10.0.2.15*) and **VM2** (IP address *10.0.2.6*), were created using **VirtualBox** and attached to the same virtual network, allowing them to communicate with each other.
Figure 6.2 illustrates the test environment used to evaluate the SLPFActuator_iptables.

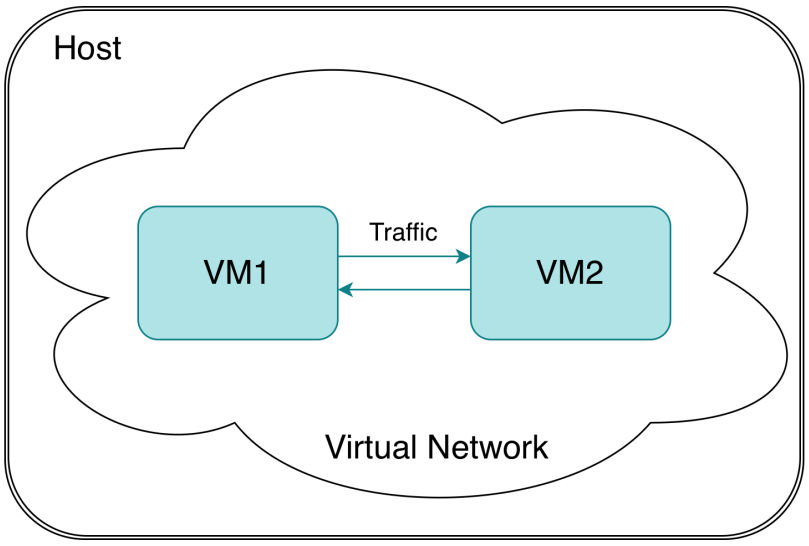

Fig 6.2 iptables test environment

Initially, ICMP, TCP and UDP traffic was generated from VM1 to VM2 and the variations in the **ingress traffic** on VM2 were observed as different SLPF commands were issued to allow or deny specific types of inbound traffic directed toward VM2.
At the beginning of the test and until t = 10 s, no filtering rules were active. This interval therefore illustrates the default behavior of iptables, in which all inbound traffic is accepted.
At t = 10 s, a *deny* command was issued to block all inbound traffic to VM2 originating from the network *10.0.2.0/24* (*ipv4_net* target), which includes VM1. As a result, all three traffic types were successfully blocked.
At t = 20 s, an *allow* command was executed to permit only ICMP traffic originating from the IP address *10.0.2.15* associated with VM1 (*ipv4_connection* target). This step not only demonstrates the effectiveness of the allow and deny commands for the iptables actuator, but also highlights the specificity of iptables rules: only ICMP traffic from VM1 is allowed, while non-ICMP traffic from VM1 and all traffic from any other IP address continues to be blocked due to the previously issued deny command.
At t = 30 s, a *delete* command was issued to remove the rule allowing ICMP traffic and a new *allow* command was applied to permit only TCP traffic from VM1.
At t = 40 s, the TCP rule was deleted and replaced with an allow rule permitting only UDP traffic from the same IP address.
At t = 50 s, the UDP allow rule was deleted, resulting once again in all inbound traffic to VM2 being blocked.
Finally, at t = 60 s, the deny rule targeting the entire *10.0.2.0/24* network was removed, restoring the ability of all three traffic types to reach VM2.
Figure 6.3 illustrates the results of this test, showing the evolution of inbound traffic received by VM2 as different SLPF commands were applied.

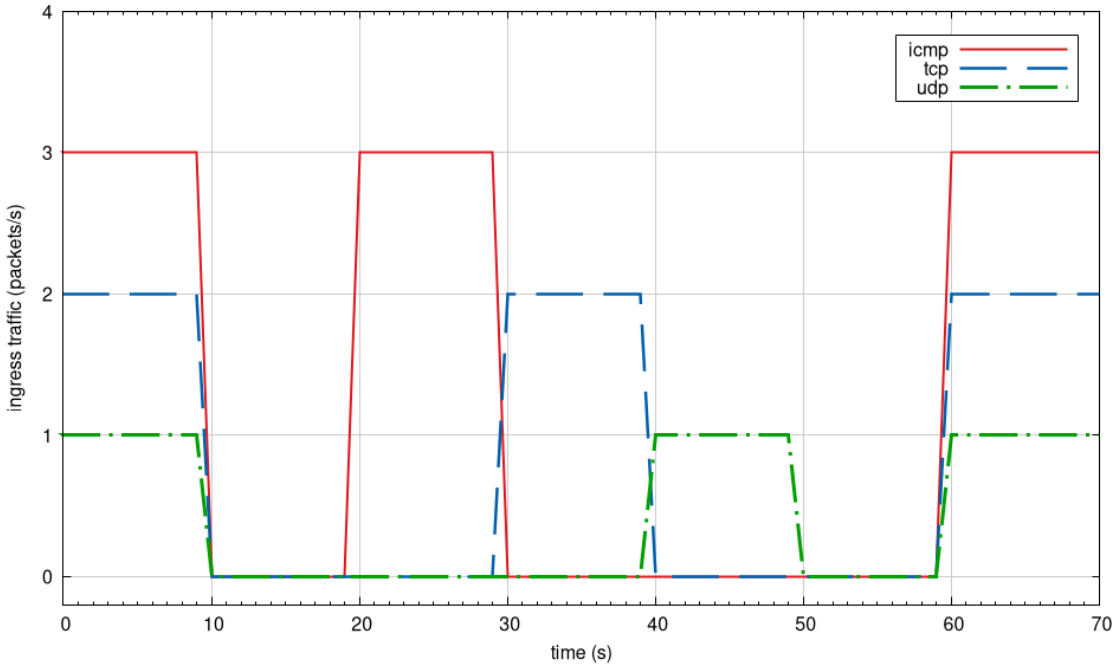

Fig. 6.3 Effect of SLPF commands on VM2 ingress traffic

To assess the effectiveness of SLPF commands for controlling **egress traffic**, a second test was performed in which ICMP, TCP and UDP traffic was generated from VM2 toward VM1.
The same procedure used for inbound traffic was repeated, but in this case SLPF commands were issued to allow or deny different types of outbound traffic from VM2, or to delete previously installed rules.
Figure 6.4 presents the results of the egress-traffic test, highlighting how the actuator consistently enforces the intended control logic.

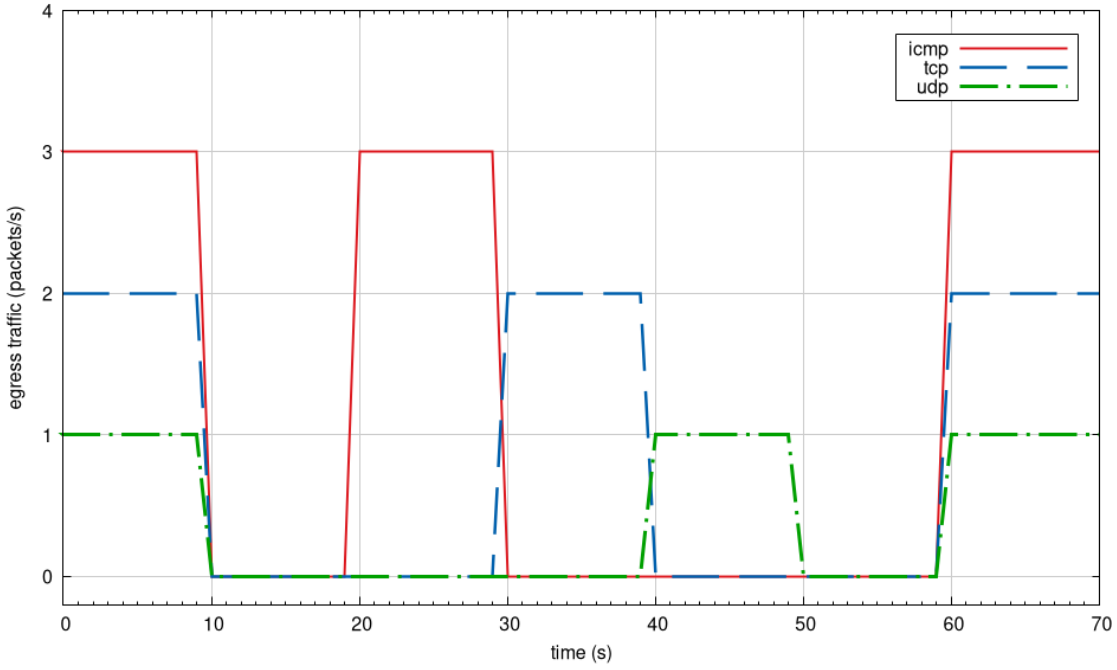

Fig. 6.4 Effect of SLPF commands on VM2 egress traffic

6.9.2 OpenStack

To evaluate the functionality of the *SLPFActuator_openstack*, three virtual machines deployed within the OpenStack cloud environment were considered: **kube0**, **kube1** and **kube2**.
Among them, **kube0** is the only virtual machine that is directly reachable from outside the OpenStack network. Access to kube0 is obtained through a VPN connection to the cloud infrastructure, followed by an SSH login to the instance.
In contrast, **kube1** and **kube2** are internal OpenStack virtual machines, that is, instances that are not exposed externally, and can be accessed only by first connecting to kube0 and then establishing an SSH session from there. These two instances serve as the endpoints used for generating and exchanging traffic during the functional tests.
Figure 6.5 illustrates the OpenStack test environment used to validate the *SLPFActuator_openstack*, showing kube0 as the externally accessible entry point and kube1/kube2 as internal instances used for traffic generation and analysis.

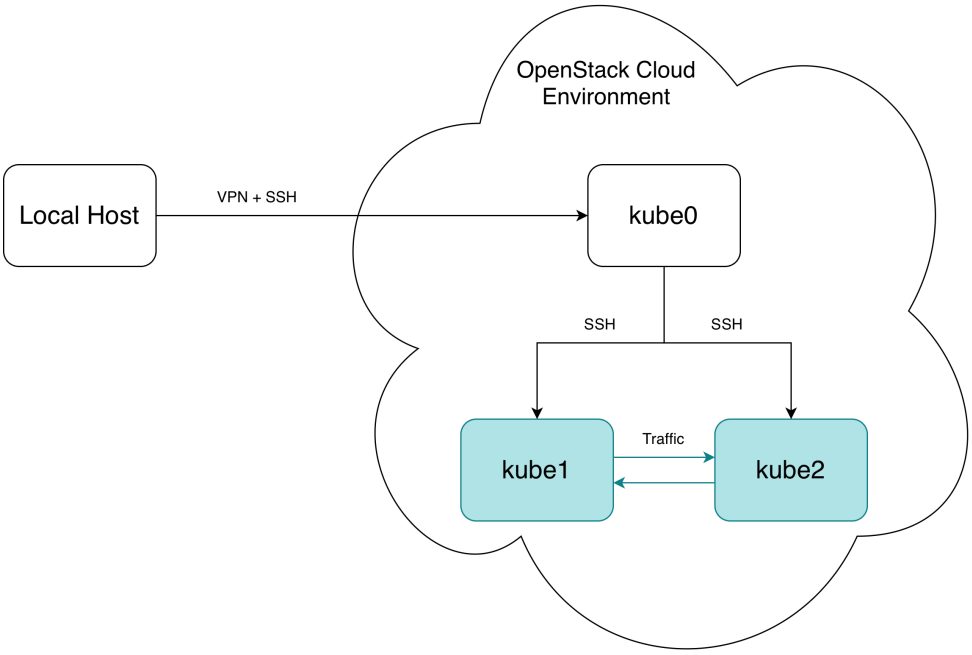

Fig 6.5 OpenStack test environment

At the beginning of the test, ICMP, TCP and UDP traffic was generated from kube1 toward kube2, and the resulting **ingress traffic** on kube2 was monitored while SLPF commands, responsible for regulating kube2's ingress traffic, were executed through the OpenStack actuator.
Unlike iptables, the default configuration of OpenStack Security Groups blocks all inbound traffic unless explicitly permitted. Consequently, during the initial phase of the test, up to t = 10 s, no packets of any type reach kube2, reflecting the baseline enforcement behavior of the OpenStack security model.
Since the *SLPFActuator_openstack*, as discussed in Section 5.10, does not implement the *deny* action, it is not possible, unlike in the iptables case, to issue an SLPF deny command to block traffic originating from the network to which kube1 belongs.
At t = 10 s, an *allow* command is applied to permit only ICMP traffic from kube1 (*ipv4_connection* target). This rule is removed at t = 20 s, which results in ICMP traffic being blocked again.
At t = 30 s, a new *allow* command enabling only TCP traffic from kube1 is executed. This rule is deleted at t = 40 s, causing TCP traffic to stop once more.
Similarly, at t = 50 s, an *allow* command for UDP traffic from kube1 is applied and its removal at t = 60 s restores the blocking of UDP packets.
In contrast to the behavior observed with iptables, where each *delete* command was immediately followed by an *allow* command enabling the next traffic type, here a deliberate interval of ten seconds is introduced between consecutive command executions. This gap accounts for the slower propagation and application time of security group updates in OpenStack, as further discussed in Section 6.10.

For the same reason, it can be observed from the graph that SLPF commands do not take effect immediately but present an approximate one-second delay. In the case of ICMP traffic, which is generated at a higher rate, the impact of SLPF commands is fully enforced after approximately two seconds, reflecting the time required for security group updates to propagate and be enforced.
Finally, in the interval from t = 60 s to t = 70 s, no filtering rules are active, thereby exposing once again the default OpenStack behavior, in which all inbound traffic is blocked.
Figure 6.6 shows the results of the test, illustrating how kube2's inbound traffic evolved in response to the application of various SLPF commands through the OpenStack actuator.

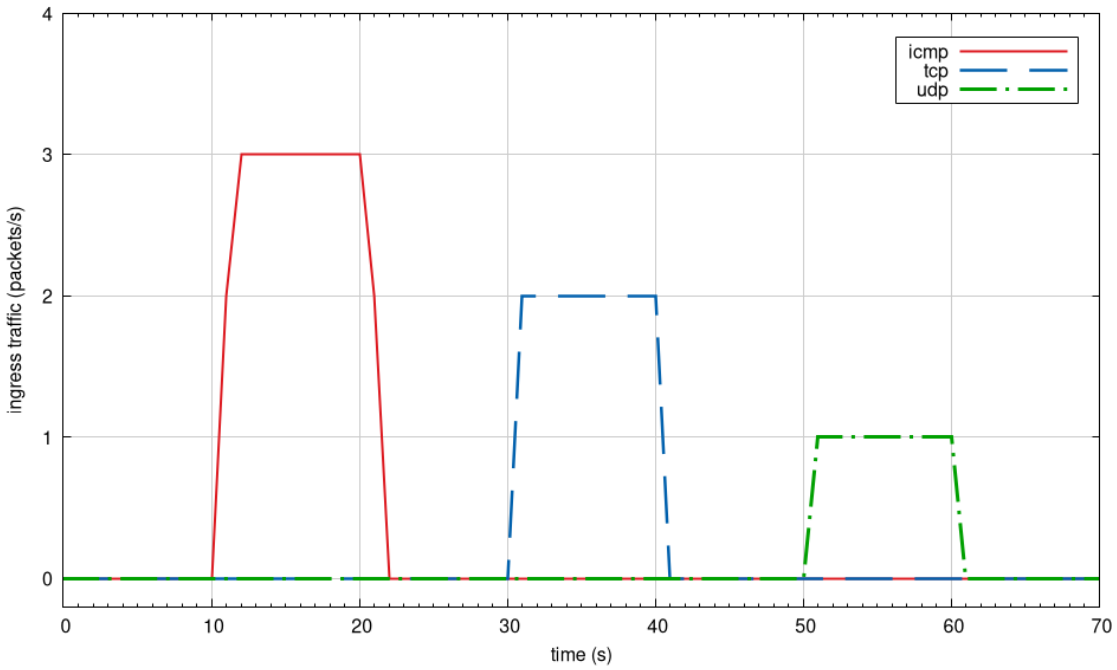

Fig. 6.6 Effect of SLPF commands on kube2 ingress traffic

To evaluate the effectiveness of SLPF commands in controlling **egress traffic**, a second test was conducted in which ICMP, TCP and UDP traffic was generated from kube2 toward kube1.
The same procedure used for inbound traffic was followed; however, in this case, SLPF commands were applied to allow different types of outbound traffic from kube2, or to remove previously applied rules.
Figure 6.7 presents the results of the egress-traffic test, showing how the OpenStack actuator enforces the intended traffic control behavior.

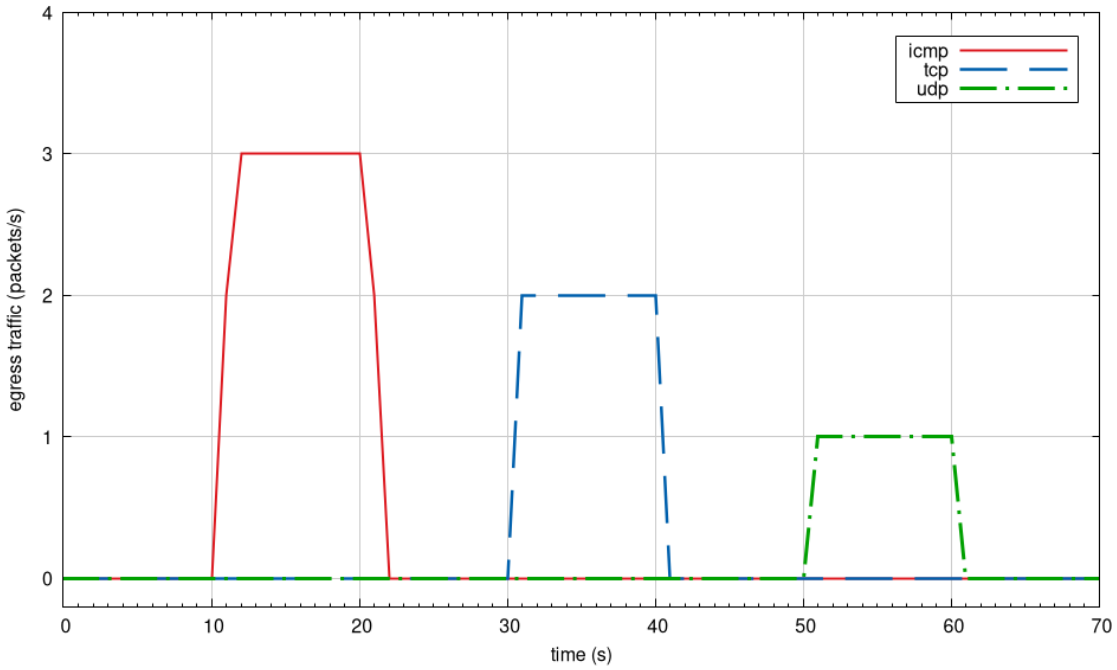

Fig. 6.7 Effect of SLPF commands on kube2 egress traffic

6.9.3 Kubernetes

To evaluate the functionality of the *SLPFActuator_kubernetes*, the same OpenStack-based cluster architecture used in the previous test was considered, consisting of three nodes: **kube0**, **kube1** and **kube2**.
Access to kube0 was obtained through a VPN connection to the cloud environment followed by an SSH login, while kube1 and kube2, being internal nodes, were reachable only by first connecting to kube0 and then establishing an SSH session from there.
On kube1 and kube2, two specific **Pods** were deployed: **bsf-0** on kube1 and **amf-0** on kube2.
The functional tests were carried out by accessing the **containers** running inside these Pods, namely, the **bsf** container on kube1 and the **amf** container on kube2, and generating traffic directly between them.
For the purposes of these tests, only TCP and UDP traffic were generated and monitored, since the *SLPFActuator_kubernetes* does not support the ICMP protocol, as explained in Subsection 5.12.2.
This setup allowed the evaluation of the actuator's capability to enforce SLPF commands within a Kubernetes environment.
Figure 6.8 illustrates the Kubernetes test environment used to validate the *SLPFActuator_kubernetes*, showing kube0 as the externally accessible entry point and kube1/kube2 as internal cluster nodes. On kube1 and kube2, the bsf-0 and amf-0 Pods contained the bsf and amf containers, respectively, which were used to generate and exchange traffic during the functional tests.

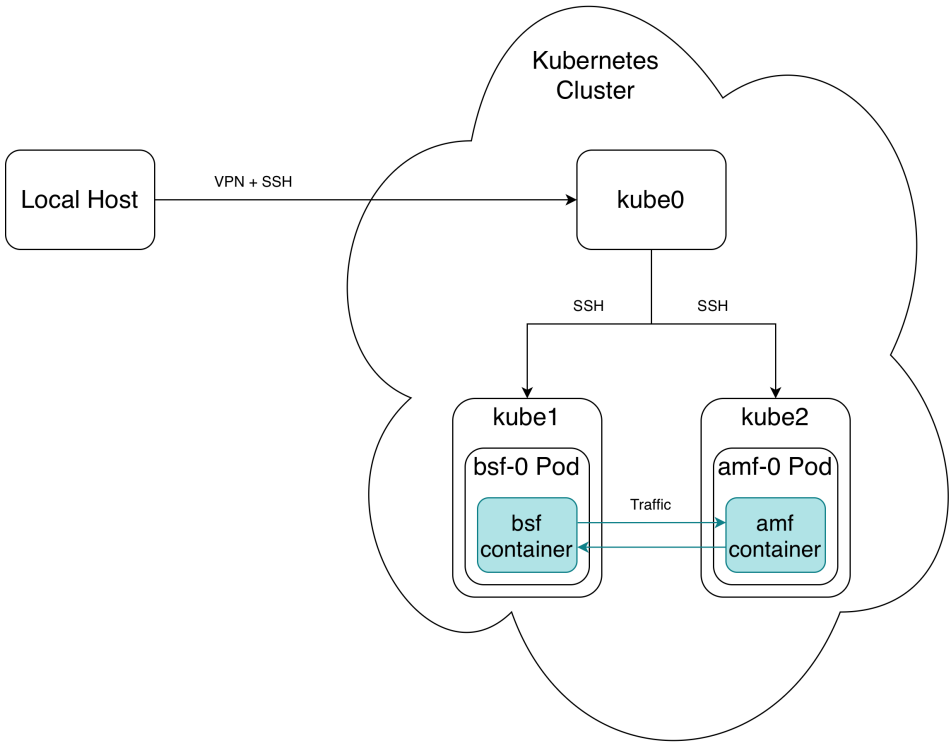

Fig 6.8 Kubernetes test environment

At the beginning of the test, TCP and UDP traffic was generated from the bsf container inside Pod bsf-0 (running on kube1) toward the amf container inside Pod amf-0 (running on kube2). The **ingress traffic** received by the amf container was then monitored while different SLPF commands were issued through the Kubernetes actuator to regulate the ingress traffic directed at this pod.

During the initial phase of the experiment, up to t = 10 s, no Network Policies were active. In Kubernetes, pods that are not selected by any Network Policy remain fully open to the cluster network: all inbound and outbound connections are allowed unless explicitly restricted. This interval therefore reflects the default allow-all behavior applied to pods without assigned policies.

Although Kubernetes does not provide a built-in *deny* action, it is possible to define a Network Policy that omits all selectors and protocol specifications (no IP blocks, no protocol and no ports). Such a policy is interpreted as a rule that effectively denies all traffic in the specified direction for the pods it selects [10]. Accordingly, at t = 10 s, an *allow* command was issued to install a policy of this form, thereby blocking all ingress traffic to the amf container.

At t = 20 s, an *allow* command was issued to enable only TCP traffic originating from the bsf container (*ipv4_connection* target).

At t = 30 s, this rule was removed and replaced with an *allow* command permitting only UDP traffic.

At t = 40 s, the UDP rule was deleted, which caused the amf container to fall back to the previously installed deny-all policy and block all inbound traffic once again.
Finally, at t = 50 s, the initial deny-all policy was removed, restoring the pod to its default allow-all state and enabling TCP and UDP traffic to flow freely again.
Figure 6.9 presents the results of this test, illustrating how the inbound traffic received by the amf container evolves over time as different SLPF commands are applied through the Kubernetes actuator.

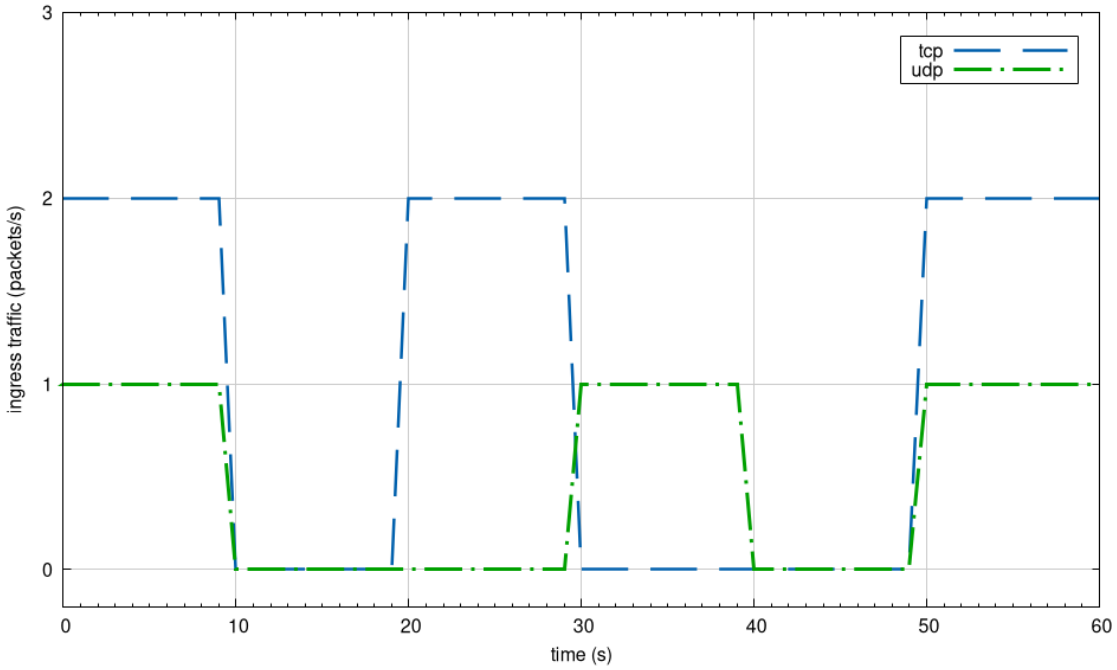

Fig. 6.9 Effect of SLPF commands on amf container ingress traffic

To assess the effectiveness of SLPF commands in controlling **egress traffic**, a second test was performed in which TCP and UDP traffic was generated from the amf container toward the bsf container.
The same procedure used for the ingress test was applied, but SLPF commands were instead issued to regulate the egress traffic originating from the amf container.
Figure 6.10 summarizes the results of this second test, showing the evolution of egress traffic from the amf container as the corresponding SLPF commands were applied.

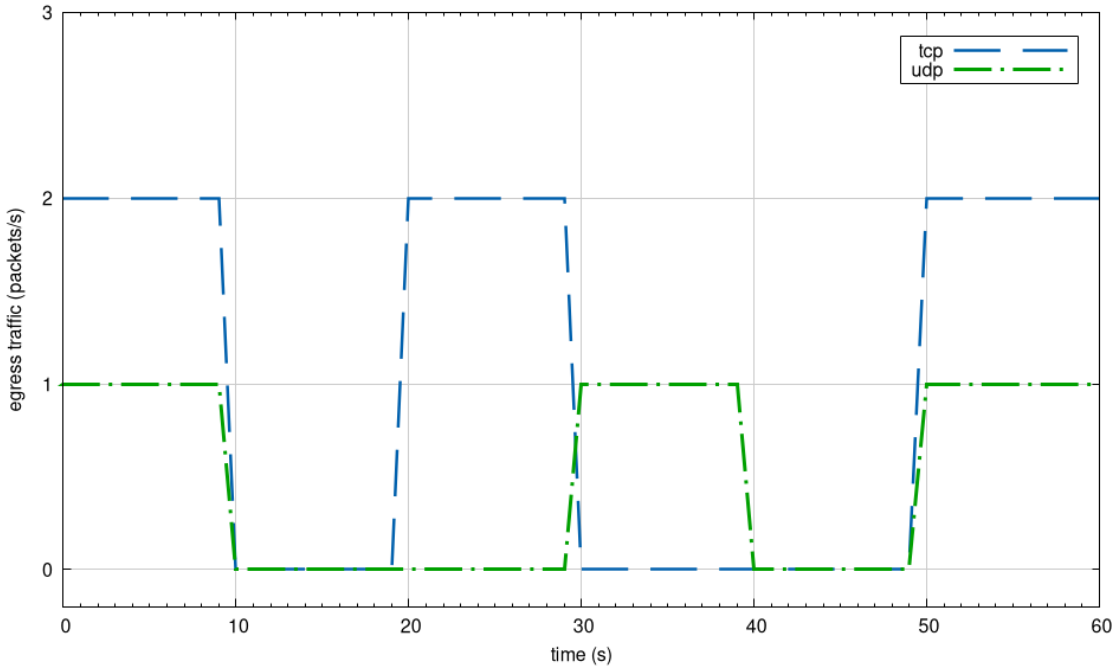

Fig. 6.10 Effect of SLPF commands on amf container egress traffic

6.10 Performance evaluation

The performance tests were designed to evaluate the responsiveness and efficiency of the SLPF Actuator Manager and its underlying actuator implementations in realistic operational scenarios. Performance evaluation considers multiple perspectives.

On the **producer side**, the component issuing OpenC2 Commands, latency is measured as the time elapsed between the dispatch of a command and the receipt of the corresponding response. This metric captures the communication overhead introduced by the system, including network delays, serialization and message processing.

On the **consumer side**, namely the SLPF Actuator Manager and the individual actuators it manages, two complementary metrics are analyzed. First, the total execution time is recorded, from the moment a command is received until its complete application within the target infrastructure. Second, the specific execution time of the underlying firewall operation is measured, such as the insertion or deletion of rules.

By combining producer-side latency and consumer-side execution metrics, this evaluation provides a comprehensive view of system performance, highlighting potential bottlenecks in communication or rule enforcement.

As input for the performance test, all valid allow commands that can be issued to the SLPF Actuator Manager were used, constructed as described in Subchapter 6.8. This included every combination of parameters in target that involve more than one parameter, as well as all possible values for arguments that can assume multiple values.

Each generated *allow* command was sent from the producer to the consumer, executed, and all metrics described above were recorded.

Upon successful execution of each *allow* command, a corresponding *delete* command was issued to the consumer, and the same set of metrics was collected.

For each measured metric, a corresponding **box plot** was generated to visually summarize the distribution of values across all test runs. Box plots provide a compact representation of key statistical characteristics, including the median, interquartile range and potential outliers, allowing for a quick assessment of variability and consistency in performance. By examining these plots, it becomes easier to identify trends or occasional spikes in latency or execution times, offering valuable insight into the behavior of the SLPF Actuator Manager.
Figure 6.11 shows the performance test results for the *SLPFActuator_iptables*.

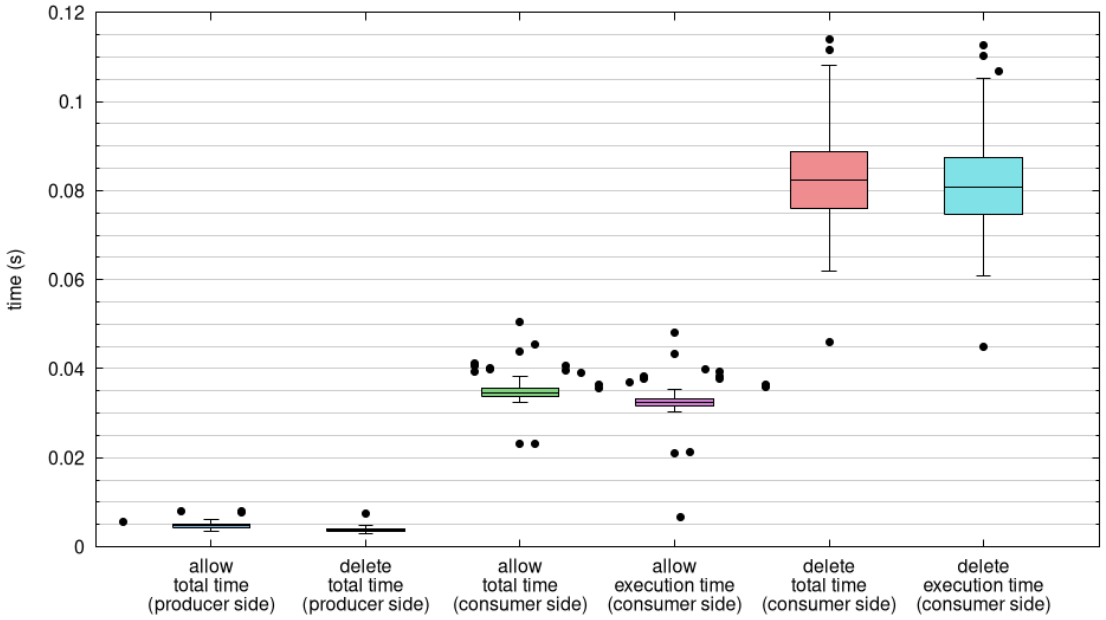

Fig. 6.11 Performance test results for the SLPFActuator_iptables

The plot presents multiple box plots, each representing the distribution of measured times for a specific metric. The boxes are arranged from left to right according to the type of measurement:

- **Producer-side latency:** the first two boxes show the time elapsed between sending a command and receiving the corresponding response. The first box corresponds to the *allow* command and the second to the *delete* command.
- **Consumer-side execution times for the allow command:** the next two boxes report, respectively, the total execution time (from command reception to completion) and the time required to perform the actual allow operation in the iptables environment.
- **Consumer-side execution times for the delete command:** the last two boxes show the total execution time for the delete command and the time specifically spent performing the delete operation in iptables.

By reading the box plots from left to right, it is possible to compare the latency and execution characteristics of allow and delete commands at both the producer and consumer sides and to identify potential variability or occasional spikes in processing times.
In particular, it can be observed that producer-side latencies are significantly shorter than consumer-side execution times.

As explained in Section 5.1, this is expected: the actuator executes the command asynchronously in the background, and the response is returned to the producer before the underlying operation has actually completed.
It is also evident, from the consumer-side measurements, that the total execution time of each command closely mirrors the time required to perform the corresponding firewall operation in the iptables environment. This indicates that the majority of the consumer-side processing time is dominated by the execution of the underlying iptables action itself, with only a minimal overhead attributable to the OpenC2 abstraction.

Figure 6.12 presents the performance results for the *SLPFActuator_openstack*.

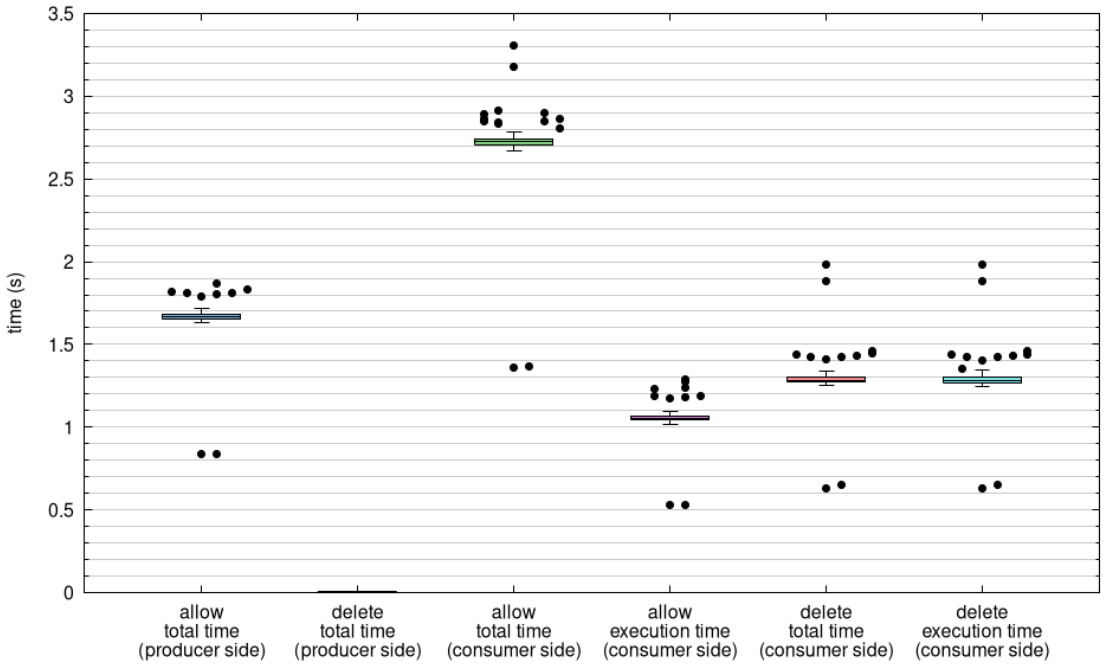

Fig. 6.12 Performance test results for the SLPFActuator_openstack

As shown in the figure, the average values of all measured metrics are significantly higher than in the iptables case and generally exceed one second, except for the producer-side total time of the *delete* command. This behavior is due to the inherent slowness of OpenStack in performing management operations and it explains the delays observed during the functional tests for the OpenStack actuator, where *allow* and *delete* commands were actually applied within the OpenStack environment only after more than one second.
It can also be observed that the consumer-side total time of the *allow* command is considerably higher than the time required to perform the allow operation itself within the OpenStack environment. As explained in Section 5.10.2, this difference is caused by the additional processing required to translate the OpenC2 abstraction into the OpenStack-specific model, such as identifying the relevant OpenStack ports (network interfaces) involved in the command and creating the corresponding Security Groups.

Since these steps occur before the action is scheduled by the SLPF Actuator Manager, the producer-side total time is similarly increased.

Figure 6.13 illustrates the performance test results obtained for the *SLPFActuator_kubernetes*.

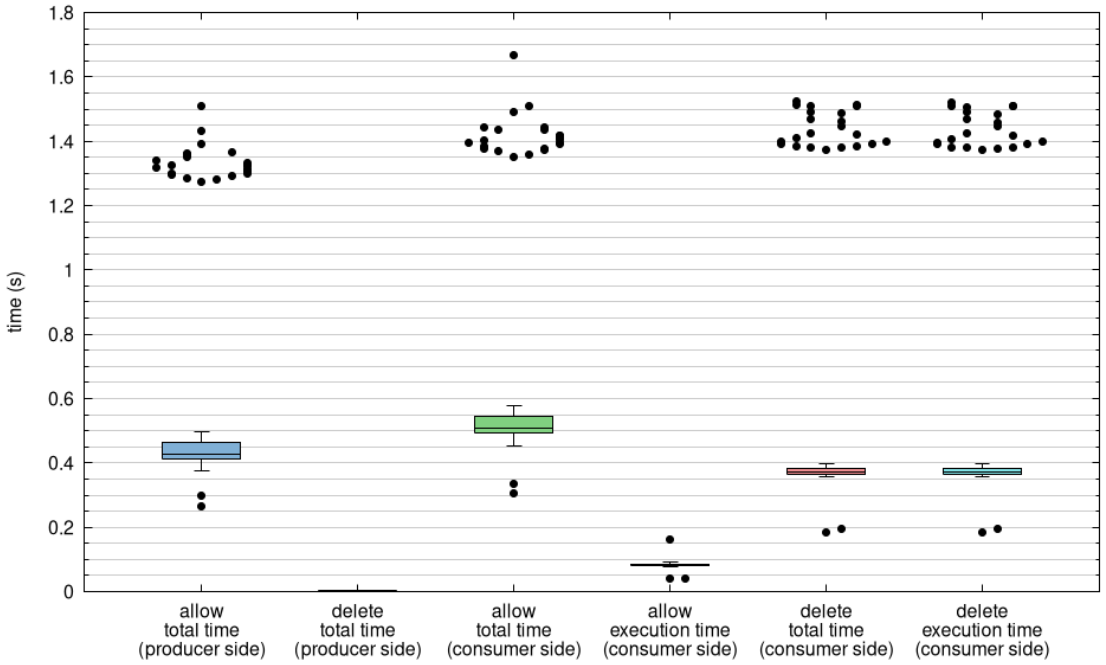

Fig. 6.13 Performance test results for the SLPFActuator_kubernetes

As can be seen from the figure, the average of the measured metrics does not exceed approximately half a second. This result highlights the significantly higher responsiveness of the Kubernetes environment compared to OpenStack, confirming the efficiency of its Network Policy management and the actuator's handling of rule updates.
Similar to the OpenStack case, the consumer-side total time for the *allow* command is greater than the time required to perform the allow action itself in the Kubernetes environment. This difference is due to the additional processing needed to translate the OpenC2 abstraction into the Kubernetes-specific model, such as identifying the pods involved in the command and generating the labels to be associated with the respective pods and network policies, as explained in Subchapter 5.11.2.
Since these operations take place before the SLPF Actuator Manager schedules the action, the total producer-side time is also correspondingly increased.

7 Conclusions

In this thesis, a fully functional SLPF Actuator Manager compliant with the OpenC2 Stateless Packet Filtering (SLPF) Profile has been designed and implemented. The implementation targets four heterogeneous firewall technologies, iptables, OpenStack Security Groups, Kubernetes Network Policies and Microsoft Azure Network Security Groups (NSGs), providing a concrete realization of the SLPF Specification [6] across diverse environments.

The primary objective was to validate the interoperability and flexibility of OpenC2 by creating a standardized command interface capable of consistently controlling packet-filtering capabilities across multiple platforms. To achieve this, a modular and extensible architecture was developed, consisting of a generic Actuator Manager and a set of backend-specific actuators responsible for translating OpenC2 Commands into the native syntax and semantics of each firewall system.

By adopting this approach, the Actuator Manager provides all the shared mechanisms required for proper actuator functioning, including initialization and shutdown procedures, validation of received commands, management of the internal database that tracks active filtering rules and the scheduling infrastructure responsible for executing time-based commands. Likewise, common functionalities, such as handling standard arguments like *start_time*, *stop_time*, *duration*, *response_requested*, *persistent* and *insert_rule,* are implemented once within the Actuator Manager and automatically made available to all backend modules.

Thanks to this architecture, integrating a new firewall platform requires implementing only the backend-specific logic. In particular, each backend must define the methods responsible for translating and executing SLPF Commands, comprising the action, the target and the backend-specific arguments *direction* and *drop_process*, within the corresponding firewall environment, as well as a small set of auxiliary methods required for proper interaction with the Actuator Manager's workflow.

This significantly reduces the effort required to support additional technologies and reinforces the extensibility and long-term maintainability of the overall system.

The SLPF Actuator Manager was validated through a combination of semantic and syntactic checks, functional tests and performance measurements. Semantic and syntactic validation ensured that OpenC2 commands, including actions, targets and arguments, were correctly interpreted and compliant with the SLPF Specification [6].

Functional tests verified the correct enforcement of allow, deny and delete operations across all supported backend firewalls, while performance tests measured command latency and execution times to assess responsiveness and efficiency.

The implementation and evaluation carried out in this work provide concrete evidence of the validity and practicality of the OpenC2 SLPF Actuator Profile. Throughout the integration of four heterogeneous firewall technologies, the SLPF Specification [6] proved sufficiently expressive to capture the essential capabilities of stateless packet-filtering systems while remaining generic enough to be applied across platforms with fundamentally different architectures and enforcement mechanisms.
This demonstrates that the SLPF Profile provides an effective foundation for interoperability, allowing higher-level automation logic to remain independent of the underlying filtering technology.

Despite its overall effectiveness, the SLPF Profile also presents some limitations that emerged during the integration of the four backend technologies.
A first constraint concerns the representation of network-level filtering through the *ipv4_net* and *ipv6_net* targets. In their current form, these targets allow specifying only a single CIDR block, which must be interpreted either as the source network or as the destination network [6]. This forces implementers to choose one of the two roles and prevents the expression of rules that simultaneously constrain both source and destination networks. Allowing both to be specified explicitly would enable more precise and expressive filtering semantics.
A second limitation relates to how traffic endpoints are expressed in the *ipv4_connection*, *ipv6_connection*, *ipv4_net* and *ipv6_net* targets. The profile mandates that both source and destination identifiers be expressed exclusively as IP addresses expressed in CIDR notation. While sufficient for basic filtering, this representation is restrictive when applied to modern firewalling systems that support higher-level abstractions.
As observed in the OpenStack and Kubernetes actuators (Sections 5.10.2 and 5.11.2), CIDR ranges must first be mapped to the underlying logical entities of the platform, namely ports in OpenStack and pods in Kubernetes. For inbound rules, all ports or pods whose IP addresses match the specified CIDR must be identified as destinations, for outbound rules, the same process applies to determine the corresponding sources.
In OpenStack, the selected ports must then be associated with appropriate Security Groups, which act as the enforcement mechanism for ingress and egress policies.
In Kubernetes, the corresponding pods must be labeled and those labels are subsequently referenced in the Network Policies that enforce the filtering behavior.
For this reason, it would be beneficial for the SLPF Profile to allow targets to be expressed not only as IP-based address ranges but also as string identifiers representing higher-level objects known to the actuator. For example, in OpenStack this could correspond to the specific OpenStack ports (network interfaces) targeted by the command, while in Kubernetes it could refer to the pods affected by the filtering action. Such an extension would increase expressiveness, reduce the reliance on IP-based matching and better align OpenC2 with the abstractions used by contemporary cloud-native and virtualized firewall technologies.

Additionally, this approach would help mitigate issues related to the dynamic assignment of IP addresses to virtual machines and pods, as commonly encountered in OpenStack and Kubernetes environments.
The last limitation concerns the definition of the *update* action in the SLPF Specification [6]. As currently described, update allows an actuator to replace its active filtering rules with the contents of a file referenced through a *file* target, provided that the file format is natively supported by the underlying firewall technology. While this approach works for systems that natively allow updating active rules from rule files, it becomes restrictive for environments that do not offer such functionality, as observed in the OpenStack and Microsoft Azure actuator implementations.
To address this limitation, it would be beneficial to abstract the semantics of the *update* action so that it does not depend on implementation-specific rule files. Instead, the file target could contain a list of SLPF allow and deny commands, each with its associated targets and arguments, expressed in a standardized format, such as JSON. This approach would increase the applicability and usefulness of the *update* action in several ways. First, it would enable the use of all implementation-independent arguments (*start_time*, *stop_time*, *duration*, *response_requested*, *insert_rule* and *persistent*). Second, it would allow actuators whose native platforms do not support updates to implement the update action consistently and correctly, thereby improving interoperability across heterogeneous firewall technologies.

Future developments could focus on extending the SLPF Actuator Manager to integrate additional actuators supporting other firewall technologies not covered in this work. Thanks to the modular and extensible architecture of the Actuator Manager, new actuators can be implemented by defining the backend-specific methods responsible for translating and executing SLPF Commands in the target environment, while relying on the shared core functionalities provided by the Actuator Manager.
This approach would allow the SLPF Actuator Manager to manage a broader range of packet-filtering systems, further validating the interoperability and flexibility of the SLPF Profile across diverse platforms.
Moreover, future work could consider extending the SLPF Profile itself to support allow and deny targets identified by higher-level entities rather than only by IP addresses, enabling more expressive and platform-aware filtering rules across heterogeneous environments.
Finally, the potential of using artificial intelligence and large language models (LLMs) to automatically generate OpenC2 commands is currently being investigated in an ongoing thesis, aiming to further simplify command creation and enhance automation in network security management.

Bibliography

[1] OASIS, *Open Command and Control (OpenC2) Architecture Specification Version 1.0*, edited by D. Sparrell, 30 Sept. 2022. Available: https://docs.oasis-open.org/openc2/oc2arch/v1.0/cs01/oc2archv1.0-cs01.html

[2] OASIS, *Open Command and Control (OpenC2) Language Specification Version 1.0,* edited by J. Romano and D. Sparrell, 24 Nov. 2019. Available: https://docs.oasis-open.org/openc2/oc2ls/v1.0/cs02/oc2ls-v1.0-cs02.html

[3] OASIS, *OpenC2 Actuator Profile Development Process Version 1.0*, edited by D. Lemire and D. Kemp, 17 Jan. 2024. Available: https://docs.oasis-open.org/openc2/cn-appdev/v1.0/cn01/cn-appdev-v1.0-cn01.html

[4] OASIS, *Specification for Transfer of OpenC2 Messages via HTTPS Version 1.1*, edited by D. Lemire, 30 Nov. 2021. Available: https://docs.oasis-open.org/openc2/open-impl-https/v1.1/cs01/open-impl-https-v1.1-cs01.html

[5] OASIS, *Specification for Transfer of OpenC2 Messages via MQTT Version 1.0*, edited by D. Lemire, 19 Nov. 2021. Available: https://docs.oasis-open.org/openc2/transf-mqtt/v1.0/cs01/transf-mqtt-v1.0-cs01.html

[6] OASIS, *Open Command and Control (OpenC2) Profile for Stateless Packet Filtering Version 1.0*, edited by J. Brule, D. Sparrell, and A. Everett, 11 July 2019. Available: https://docs.oasis-open.org/openc2/oc2slpf/v1.0/cs01/oc2slpf-v1.0-cs01.html

[7] O. Andreasson, *Iptables Tutorial 1.2.3*, Netfilter Documentation Project, 2006. Available: https://www.frozentux.net/iptables-tutorial/iptables-tutorial.html

[8] OpenStack Foundation, *OpenStack Networking Guide*, 2025. Available: https://docs.openstack.org/neutron/latest/admin/index.html

[9] OpenStack Foundation, “Security group rule” in *Python OpenStackClient Documentation*, 2025. Available: https://docs.openstack.org/python-openstackclient/latest/cli/command-objects/security-group-rule.html

[10] The Kubernetes Authors, *Concepts*, Kubernetes Documentation, The Linux Foundation, Cloud Native Computing Foundation (CNCF). Available: https://kubernetes.io/docs/concepts/

[11] Microsoft Corporation, "What is a Network Security Group (NSG)?", *Microsoft Learn Documentation*, 2025. Available: https://learn.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview

[12] M. Repetto, "Otupy: a Flexible, Portable, and Extensible Framework for Remote Control of Security Functions", *Computers & Security*, vol. 158, Nov. 2025. Available: https://www.sciencedirect.com/science/article/pii/S016740482500286X?via=ihub

[13] SQLite Consortium, *About SQLite*, *SQLite Official Website*, 2025. Available: https://www.sqlite.org/about.html

[14] APScheduler Documentation, *User Guide*, 2025. Available: https://apscheduler.readthedocs.io/en/stable/userguide.html

[15] OpenStack Foundation, *OpenStackSDK Documentation*, OpenStack, 2025. Available: https://docs.openstack.org/openstacksdk/latest/

[16] Kubernetes Python Client, *Kubernetes Python Client Documentation*, 2025. Available: https://kubernetes.readthedocs.io/en/latest/

[17] Microsoft Corporation, *Azure SDK for Python Documentation*, The .NET Foundation & Contributors. Available: https://learn.microsoft.com/it-it/azure/developer/python/