POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Service-Agnostic Configuration and Control for eBPF Services

Supervisor

prof. Fulvio Risso

Candidate

Nico Caprioli

Academic Year 2018-2019

Contents

Chapter 1

Introduction

Nowadays, online service providers are moving from hardware-based network infrastructure to a software-based one. In particular, the *network function virtualisation* (NFV) decreases the cost of deployment of computer networks because it breaks the explicit coupling with specific equipment. Moreover, NFV enables to achieve better scalability and provide more agile services.

1.1 XDP and eBPF

One of the biggest problems regarding NFVs is the performance; in fact, in a software solution, each packet must traverse the whole network stack, producing significant and non-feasible delays.

In order to solve the problem, the Linux community proposed a solution named *eXpress Data Path* (XDP), which is an integrated fast path in the kernel stack. XDP goal is to provide a way to process incoming packets in the lowest point of the software stack, which makes it ideal for speed without compromising programmability.

XDP retrieves raw packets; it is then required the *extended Berkeley Packet Filter* (eBPF) which allows writing programs to manage the packets. Those programs run in a virtual machine inside the kernel space (e.g., parse the packet, drop, forward, manipulate, etc.).

1.2 Polycube

In this scenario comes the `polycube` framework, which aims to provide a uniform and simplified way to develop network functions based on XDP and eBPF, named services. One other important `polycube` feature is service chaining.

Figure 1.1 depicts the `polycube` architecture prior to this work. The most important component of the architecture is the polycube daemon, named `polycubed`, which is a service-agnostic user space daemon that is in charge of interacting with the different services. In particular, the daemon exposes a *Kernel Abstraction Layer* by means of which services can access the fast path in an easier way.

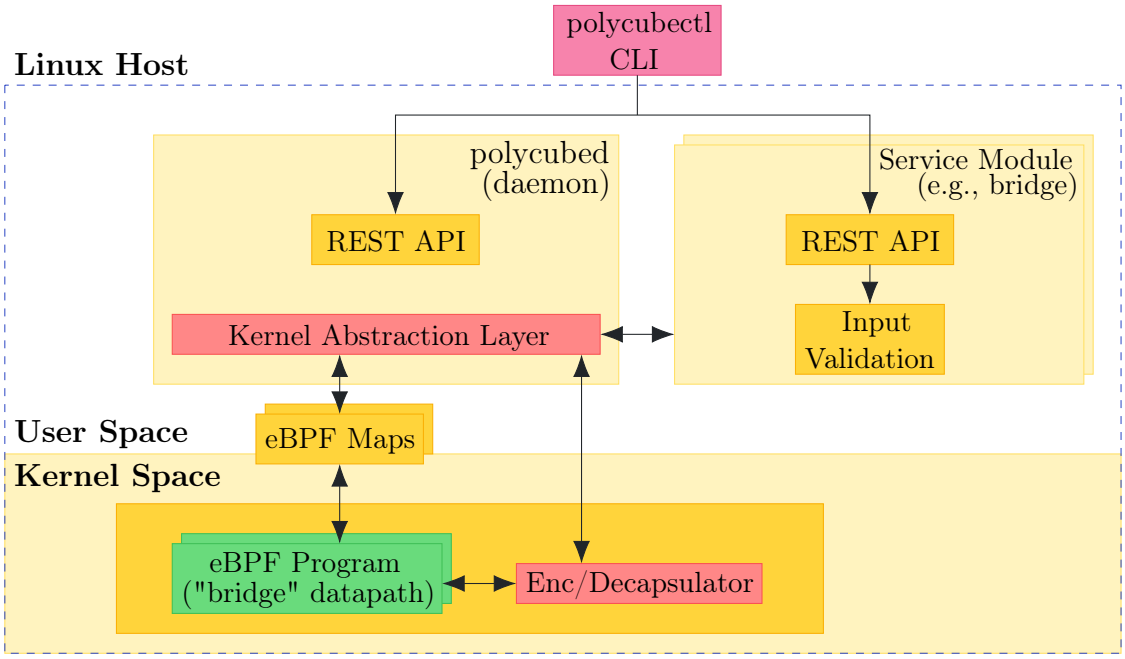

Figure 1.1. Old Architecture

One crucial aspect of this architecture is the *REST API*, which aims to enable the interaction to the framework. The REST API included in the daemon is in charge of providing web server functionalities (§3.7) and expose a set of endpoints to check the currently loaded services. Instead, the REST API included in each service is in charge of providing the endpoints to interact with the service itself, and their associated handlers to the web server contained in the daemon.

It is important to notice that each service is entirely contained within a single

shared object which, in turn, loads a library named `libpolycube` which leverages on `BCC` and thus provides the `polycube` functionalities to the service.

Chapter 2

Thesis Goal

This chapter will present how and why this thesis aims to improve the architecture presented in Figure 1.1.

2.1 Problem

Before talking about the improvement proposed and implemented with this thesis, it is crucial to see the problems it aims to solve in the bigger picture.

2.1.1 Service Structure

Most of the problems solved by this work derive from the internal structure of a general service, which is depicted in Figure 2.1. In particular, this figure shows the components making up a service and how they interact each other.

The REST API component oversees the interaction between the user and the service as it contains all the function for fetching the status of a cube and configure it. The REST endpoints and the associated HTTP are derived from the YANG data model (§3.1) and follow a well-defined structure which is described in §5.1.2.

The Input Validation component might be considered as a part of the REST API. It receives input data from the REST API and validates it across the restrictions defined within the YANG data model. If the validation succeeds, it invokes the Core Logic component.

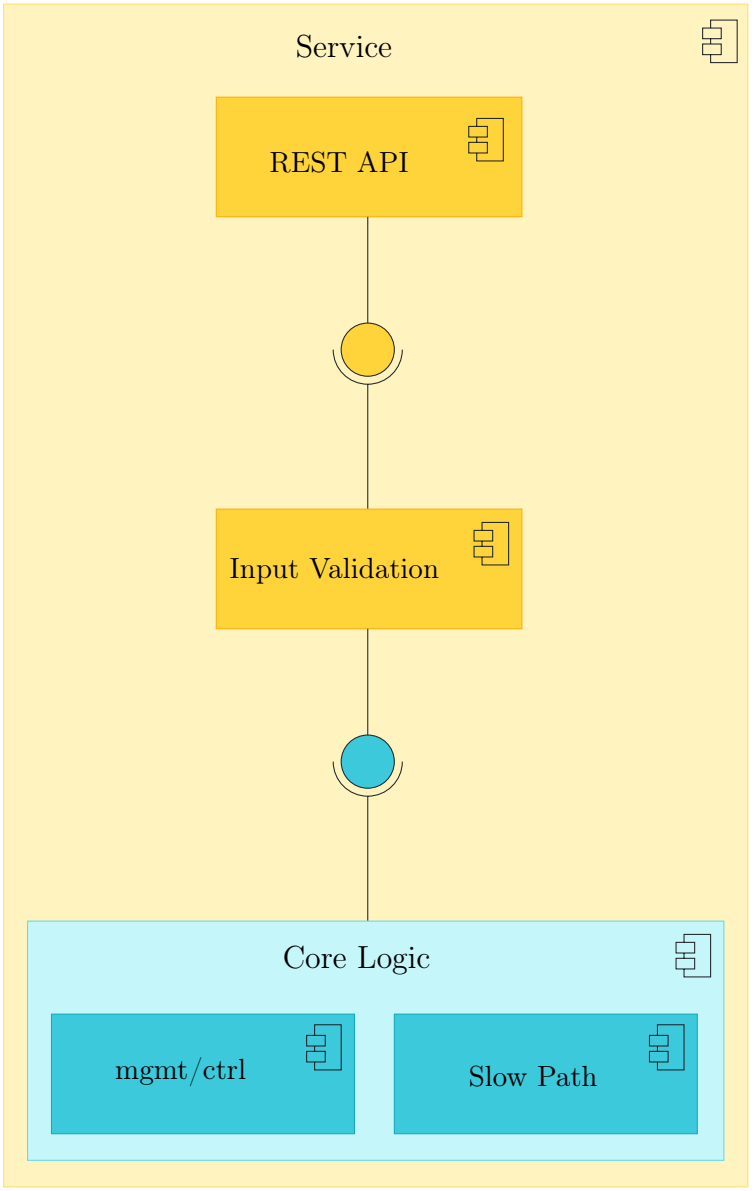

Figure 2.1. Service Component Diagram

Finally, the Core Logic component is where the actual service code resides. In particular, the slow path is standard user code, while the control plane exploits XDP and eBPF.

2.1.2 Automatic Code Generation

Along with `polycube`, it is shipped an automatic code generation tool which aims to generate all the REST APIs, the input validation component and a skeleton for the core logic. When it is possible, along with the skeleton, a partial default

implementation for the core logic is provided.

The automatic code generation tool surely helps the developer to develop a `polycube`-compatible service faster; nevertheless, it lays the foundations for a relevant drawback: the service developer freedom.

Before this work, a service developer had only two possibilities:

- Use the automatic code generation tool and then write the entire service in C++.
- Use any programming language (possibly one that allows writing eBPF programs) but implementing by hand the input validator and REST API in C++, since it requires pistache (§3.7).

It is clear that with the first case there is a strong limitation in available programming languages (namely, only C++), while in the second case there is a steeper learning curve since the service developer must understand how to map YANG nodes to REST endpoints.

Moreover, two other aspects must be taken into account with this architecture.

First of all, since third-parties develop services, the architecture presented so far is not very resilient to automatic code generation tool modifications. For instance, changes in the REST endpoint generation (e.g., to adhere the RESTCONF specification defined RFC-8040), or modification in supported REST request data type (such as supporting both JSON and XML), would require to each service developer to update their automatic code generation tool and re-generate the REST API.

Furthermore, the architecture above suffers from accidental modifications of the REST endpoint mapping by the service developers, thus breaking any automatically generated client such as the *polycube CLI*.

2.2 Solution

To solve the problem presented so far, a simple and yet effective solution has been designed.

The general idea is to move the REST API and Input Validation components from the service to the daemon. The immediate effect of this operation is that service code shrinks to only Core Logic, namely the only component on which service developer needs to focus.

2.2.1 Implications

Even though the initial idea was to remove unnecessary code from the service to allow service developers to focus only on core logic, moving the REST API and the Input Validation solved all the problems presented in 2.1.2.

In particular, the restrictions on programming language are coarser as there is no need to understand how REST endpoints are generated and then create them in C++ (§4.1.2). Nevertheless, the automatic code generation tool will still create a C++ skeleton and a partial implementation.

Secondly, moving REST API and Input Validation components makes the system more resilient to changes. For instance, changing the YANG to endpoint mapping from the actual one to a RESTCONF compliant one and updating the CLI accordingly, would not affect any service.

2.2.2 Further Features

The changes presented so far lays the foundations for some unforeseen but well-accepted features.

Let us consider a situation in which the `polycube` daemon must be moved into a new computer after a hardware upgrade, or more simply, it just crashes. In both situations, it is desirable restarting the daemon with the same configuration it has before the shutdown, without requiring to re-configure it from the ground up.

By moving the REST API into the daemon, it now has full control over all the services and can fetch data from them whenever deemed useful. This condition permits to implement `polycube` persistence regarding deployed services and their configuration directly into the daemon.

Chapter 3

Tools and Technologies

Before exploring how the solution (§2.2) to the problem in 2.1 was implemented, it is useful to provide some background on the tools used.

This chapter aims to provide all the required information about tools and languages deployed during the implementation of this work.

3.1 YANG 1.1 Data Modeling Language

Most of this work revolved around the YANG Data Modeling Language, version 1.1, which is defined in RFC-7950.

In a nutshell, YANG is a data modeling language used to model configuration data, state data, Remote Procedure Calls, and notifications for network management protocols.

YANG Nodes

Models defined through the YANG data modeling language have the form of a tree in which each node can have different properties, depending on its type. Even though YANG offers a vast number of nodes, only a subset of them can be currently used to define a `polycube` service.

The most external node in a YANG data model is the `module` one, which allows

defining the service-wide properties such as copyright or version. Ignoring the unique service-wide properties, it acts as a `container`.

The `container` node, as the same suggest, is a simple container for other YANG nodes. Due to its nature, a `container` node cannot be instantiated.

Another YANG node supported by the `polycube` framework is the `list` node. In some ways, the `list` node is similar to the `container` one, as it contains a set of nodes. The main difference between the `container` node and the `list` one is that the latter is instantiated as a collection of the elements that contains. Moreover, an arbitrary number (even zero) of `leaf` nodes contained inside the `list` can be set as `key`s for the `list` and then used to identify one instance across the entire collection uniquely. If no `key` is defined, one must be defined internally by the system.

The `leaf` node, as the name suggests, is a leaf in the data model tree. It maps to a scalar type and, as such, can be instantiated.

Finally, the last YANG node that can be used to define a `polycube` service is the `action` node. It can only contain two nodes: `input` and `output`. Both of them act as a `container`. The purpose of the `action` node is to enable the service developer to create an operation that works on other nodes, explicitly setting its input and output. For instance, an `action` may be the flushing of a bridge filtering database, which is defined elsewhere.

Configuration and Status Nodes

Currently, the `polycube` framework does not support either notifications nor Remote Procedure Calls. Nevertheless, it is possible to configure a node as a *configuration node* or *status node* using the `config` parameter.

To better understand the difference between status and configuration nodes, let us consider a routing table. Any static route, or the routing algorithm to use, are part of the configuration; instead, the whole list (including static routes, directly connected devices, and routes computed with the routing algorithm) is the status of the routing table.

YANG Types and Restrictions

As already mentioned, the `leaf` node represents a scalar type. In more details, a `leaf` can have a type (such as `integer`, `string`, or `enumeration`) and optionally some restrictions on it.

The restrictions purpose is to reduce the available value space following some well-defined rules. For instance, it is possible to define a regular expression for a string or set a limit on its length. It is possible to force a number in a given range, and so on.

For the full context, refer to [1].

3.1.1 Polycube Service YANG Structure

Within the `polycube` framework, YANG is the language used for defining services' structure. Each service is contained in precisely one `module`, where the service name is the `module` name.

In order to instantiate cubes of a specific service, configure it, or fetch status data, the framework exposes a set of REST endpoints. Since YANG was already the language used to define services, part of this work consisted in reverse-engineering the REST endpoint generation algorithm, as defined in 5.1.2.

For convenience, the content of the `module` is wrapped into a `list` in a user-transparent way. This `list` represents the cubes instantiated, and it has a single `key` representing the cube name. This silent wrapping can be summarised as follows:

```
1 module sample {
2   yang-version 1.1;
3   namespace "http://polycube.network/sample";
4   prefix "sample";
5
6   import polycube-base { prefix "basemodel"; }
7
8   organization "Polycube open source project";
```

```
9    description "YANG data model for the Polycube Sample service";
10
11   basemodel:service-description "Sample Service";
12   basemodel:service-version "2.0";
13   basemodel:service-name "sample";
14
15   list sample {
16     key "cube_name";
17     leaf cube_name { type string; }
18     [...]
19   }
20 }
```

Listing 3.1. Sample Service List Wrapping

Lines 2, 3, 4, 8, and 9 are simple descriptive information about the service. Line 6 imports a `polycube`-defined YANG file which is required to create a `polycube`-compliant service. Lines 11, 12, and 13 sets `polycube` specific service information.

Finally, lines 15, 16, and 17 are the implicitly added wrapping list. This is not present into the developer's YANG file but it is required to identify instantiated cubes uniquely.

3.2 libyang

`libyang` is a YANG data modelling language parser and toolkit written in C and is available at [2].

Even though it was possible to implement a YANG parser by hand, possibly achieving higher performances as it would generate REST endpoints and validators during the parsing process, the solution was discarded.

Relying on a third-party library has the advantage of updates such as bug fixes or performance improvements regardless of `polycube` development status. Moreover,

along with the standard compile-and-install configuration, `libyang` is shipped in pre-compiled packages for most common GNU/Linux distributions.

3.3 XPath

The *XML Path Language*, better known as XPath, is a language used to address parts of a data tree.

Even though it is designed to work with XML documents, the YANG data modeling language adopted it in with version 1.0 to address one node from one other, whose full specification is available at [3]. Additionally, some YANG-specific function has been defined in [1, section 10].

XPath is very useful for enforcing logical constraints by means of `when` and `must` statement. Listing 3.2 shows an example of usage.

Discarding the preamble which is useful only to give a context, what it is essential is line 11, which can be read as: the sibling node named `ifType` is valid only if `ifMTU` is at most 1500.

```
1  container interface {
2    leaf ifType {
3      type enumeration {
4        enum ethernet;
5        enum atm;
6      }
7    }
8    leaf ifMTU {
9      type uint32;
10   }
11   must 'ifType != "ethernet" or ifMTU = 1500' {
12     error-message "An Ethernet MTU must be 1500";
13   }
14   must 'ifType != "atm" or'
15        + ' (ifMTU <= 17966 and ifMTU >= 64)' {
```

```
16      error-message "An ATM MTU must be 64 .. 17966";
17    }
18  }
```

Listing 3.2. Example of XPath Restriction

3.4 Flex & Bison

As should be clear from Listing 3.2, XPath is a string into a YANG document.

`libyang` is capable of parsing the simple XPath inside a `leafref` node and replace it with the correct `leaf`. Nevertheless, when it comes to `must` and `when` statements, `libyang` will merely put the string into the correct structure, without any parsing involved.

For this reason, it was necessary to write a parser that can understand the XPath expression inside a `must` or `when` statement.

Since writing a parser by hand is non-feasible, the tools used to write an XPath parser are *Flex* and *Bison*.

Flex is a scanner generator. A function generated with flex will take a string as input (the XPath in this case) and outputs a set of tokens. The association between the input string and the output token is achieved with regular expressions. For every matched regular expression, the lexer will advance the position in the input string and emit a token.

Let us consider the following simple mapping as an example:

```
polycube → emit POL;
[a-zA-Z]+ → emit ALPHA;
[0-9]+ → emit NUM;
.  → ignore;
```

Given the input string `polycube v0.1`, the output tokens sequence would be `POL ALPHA NUM NUM`.

The main advantage of using flex instead of standard regular expression is that flex will generate *minimised deterministic finite automaton* ahead of time, thus improving the overall performances.

The flex output token stream is passed to Bison, which is a parser generator. Through Bison, it is possible to define formally the rules forming a grammar, and an action associated with each rule.

For instance, assuming flex emits the token `DOT` when it encounters a dot and `SEP` when it encounters a forward slash, a mapping may look like the following:

```
start_with :  path;
current = DOT;
parent = DOT DOT → goto parent folder;
path = current SEP path | parent SEP path |
     ε  → print folder content;
```

As it is easy to see, the `parent` rule has one action associated that moves to the parent directory (being the example a simplified grammar, it is not clear what moves to parent, but it is not relevant). The ε rule, also known as *empty rule* will match when no other rule can match.

Since the `path` rule is recursive, this parser can handle any combination of current and parent directory, such as `./../.` or `../../././..` and so on.

For the full context regarding flex and bison, refer to [4], [5].

3.5 Representational State Transfer

REpresentational State Transfer (REST) is a software architectural style originally described in [6] that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style, termed *RESTful* web services, provide interoperability between computer systems on the Internet. RESTful web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

In a RESTful web service, requests made to a resource's URI will elicit a response with a payload formatted in either HTML, XML, JSON, or some other format. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources. When `HTTP` is used, as is most common, the operations available are `GET`, `POST`, `PUT`, `DELETE`, and other predefined CRUD HTTP methods.

In the `polycube` framework, a stored resource is any YANG defined node that can be instantiated, such as a `leaf` or a `list`. In particular, for status nodes only read operation are defined, while for the configuration ones are defined the creation, update and deletion as well.

3.6 JSON

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is the format chosen for exchanging data through the REST API. In particular, service data is encoded according to RFC-7951 [13].

3.7 Pistache

Pistache is a modern and elegant HTTP and REST framework for C++. It is entirely written in pure C++11 and provides a clear and pleasant API.

The main reason to choose this library instead of another one is because it was already used in previous `polycube` versions, and thus it did not require further integrations. Nevertheless, this library did not entirely fulfil the requirements, and thus it as been modified, as described in 5.1.2.

Chapter 4

Architecture

This chapter will present a revisitation of the architecture presented in 1.2 to solve the problems proposed in chapter 2. Comparing the Figure 4.1 with Figure 1.1, it is easy to notice the addition of a *Service Controller* module.

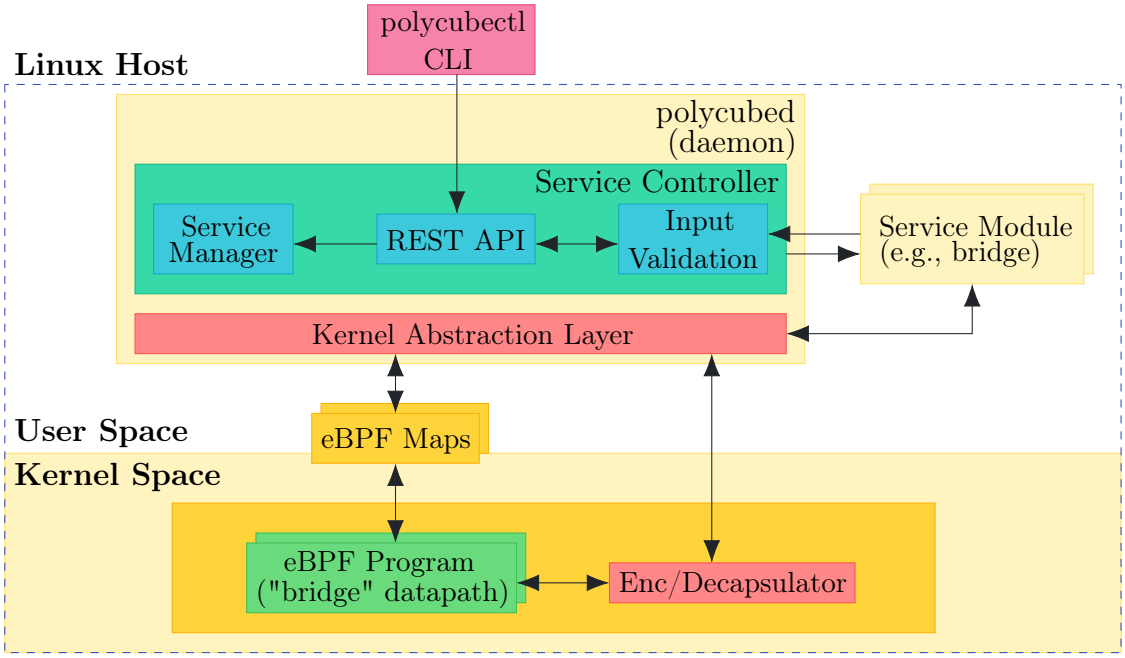

Figure 4.1. New Architecture

The purpose of this module is to provide a service-agnostic interface that allows users to interact with the services. It operates by means of three sub-modules:

- **REST API**: Receives the request from a REST client (e.g., the `polycube`

`CLI`).

- **Service Manager**: Enables services' run-time registration and unregistration (§5.2).
- **Input Validation**: Enforces YANG defined constraints on REST request path parameters and body.

The *Service Controller* will, in turn, forward the request to the correct service only if all constraints are satisfied, otherwise returns an error.

4.1 Services as Grey-boxes

`polycube` design lets anyone develop a custom service; nevertheless, the daemon cannot know services' internals. For this reason, it has been designed a uniform protocol that allows the daemon to exchange data with any service module.

This architectural model goes by the name of **grey-box**. In particular, with this work, only services contained in a shared object each are supported.

Let us consider the trivial data model defined in listing 4.1 as an example.

The shared object that implements this service must expose a set of well-defined entry points.

```
1  module example {
2    list ports {
3    key "name";
4      leaf name {
5        type string;
6      }
7      leaf ip {
8        type inet:ipv4-address;
9        config false;
10     }
11   }
```

```
12 }
```

Listing 4.1. YANG Data model example

First of all, it is defined a special purpose entry point that returns the YANG data model:

```
1 extern "C" {
2   const char *data_model();
3 }
```

Listing 4.2. YANG data model entry point

For all the other functions, since there is no guarantee of success, the daemon will know the outcome of the operation by means of a simple `struct` named `Response`, whose goal is to store the response body along with an enumerable value stating whether the operation was successful or, if there was an error, what kind of error was it.

The service-wide entry points can be considered special purpose ones, since they are always present, and they have fixed parameter known at compile time by the daemon:

```
1  extern "C" {
2    Response create_example_by_id(const char *cube_name,
3                                  const char *value);
4    Response update_example_by_id(const char *cube_name,
5                                  const char *value);
6    Response replace_example_by_id(const char *cube_name,
7                                  const char *value);
8    Response read_example_by_id(const char *cube_name);
9    Response delete_example_by_id(const char *cube_name);
10
11   Response read_example_list_by_id();
12   Response update_example_list_by_id(const char *value);
```

```
13    Response read_example_list_by_id_get_list();
14  }
```

Listing 4.3. Service wide entry points

The functions from line 2 to line 9 expose the operations on a single `Cube`, identified by its name; the value parameter is the validated request JSON body. The function on line 11 and 12 allow to operate on the whole list of `Cubes` within the service. The function on line 13 returns the list of the names of all instantiated `Cubes`.

Finally, all other nodes defined in the YANG data model take a set of functions each. Before showing how these nodes are mapped to the correct entry points, we need to show how `list keys` are managed. Since there is no control over YANG data models submitted by the user, each node might be a descendant of zero or more lists, each with a variable number of `keys`.

The `Key` structure overcomes the problem by specifying the type of the value so that it is possible to retrieve it without any casting required:

```
1  enum ElementType {
2    BOOLEAN, STRING, INT8, INT16, INT32, INT64,
3    UINT8, UINT16, UINT32, UINT64, DECIMAL
4  };
5
6  union ElementValue {
7    bool boolean;
8    const char *string;
9    int8_t int8;
10   int16_t int16;
11   int32_t int32;
12   int64_t int64;
13   uint8_t uint8;
14   uint16_t uint16;
15   uint32_t uint32;
```

```
16    uint64_t uint64;
17  };
18
19  typedef struct {
20    const char *name;
21    enum ElementType type;
22    union ElementValue value;
23  } Key;
```

Listing 4.4. Structure for passing a list key

The service implementation will then take the correct `value` depending on the `type`.

For the sake of simplicity, function names have been simplified using the placeholder `data_op` which can be one between `create`, `update` or `replace` and with `no_data_op` which can be `read` or `delete`.

```
1  Response
2  data_op_example_ports_by_id(const char *cube_name, Key *keys,
3                              size_t num_keys, const char *value);
4  Response
5  no_data_op_example_ports_by_id(const char *cube_name, Key *keys,
6                                 size_t num_keys);
7
8  Response
9  data_op_example_ports_list_by_id(const char *cube_name, Key *keys,
10                                  size_t num_keys, const char *value);
11 Response
12 no_data_op_example_ports_list_by_id(const char *cube_name, Key *keys,
13                                    size_t num_keys);
14
15 Response
16 read_example_ports_list_by_id_get_list(const char *cube_name, Key *keys,
17                                       size_t num_keys);
```

```
18
19 Response
20 read_example_ports_ip_by_id(const char *cube_name, Key *keys,
21                               size_t num_keys);
```

Listing 4.5. Generic YANG nodes entry points

As anticipated, each of these functions takes a C-style array of keys. Function from line 1 to line 17 are equivalent to the ones shown in listing 4.3. Instead, for `leaf` nodes are only defined single entry operations. In this particular case, since the `ip leaf` is `config false`, only the reading operation is provided.

4.1.1 Virtual Method Tables

The former architecture used a `Cube` interface declared in the `polycube` daemon and then implemented it in each service. The daemon would then have used this interface to interact with the cube.

Even though this approach may seem working, it has a significant drawback: the C++ standard states that virtual member functions must be called through virtual method tables, but it does not define how compilers should implement them. Therefore, compiling the daemon and a service with different compilers (or even different versions of the same compiler) may lead to undefined and possibly harmful behaviour. Instead, treating the service as a grey-box where the daemon interacts with the services by means of entry-points, does not have the same problem, since there is no longer a common interface.

4.1.2 Extensibility

Requiring services to be grey-boxes in which only the data exchange entry point is known in advantage, provides a great improvement over extensibility.

In particular, once the flow reaches the shared object entry point, any implementation is valid. The `polycube` provided automatic code generation [7] will still

generate a C++ implementation stub on request; nevertheless, the constraint on allowed programming languages for developing a service (i.e. C++) has been relaxed as it is now possible to use any programming language that allows developing of shared objects (e.g., python).

4.2 Workflows

This section provides information about the workflows changed with the new architecture. In particular, it will be presented a revisited version of the workflow to register a new service at run-time, and the revisited workflow to provide input data validation before it reaches the service.

4.2.1 Registering New Services at Run-time

Since `polycube` daemon is intended to be always running, requesting to stop it, register a new service and then restart it with a new set of services is not feasible. For this reason, it has been implemented a mechanism to provide run-time service registration.

Originally, the workflow to register a service at run-time was straightforward since after the request to register it by the user, the daemon would simply load the shared object which, in turn, would configure all the REST APIs. Instead, Figure 4.2 depicts how this mechanism changed with new architecture.

At first, the user must submit a new shared object to the daemon. For security reasons, instead of directly sending arbitrary binary code which will be run by the daemon with *root* permissions, the user must upload the file out-of-band (e.g., through SSH) and then provide the path to the file onto the server. A valid request might look like the following:

```
1 POST /services HTTP/1.1
2 Host: example.org
3 Content-Type: application/json
4 Content-Length: 82
```

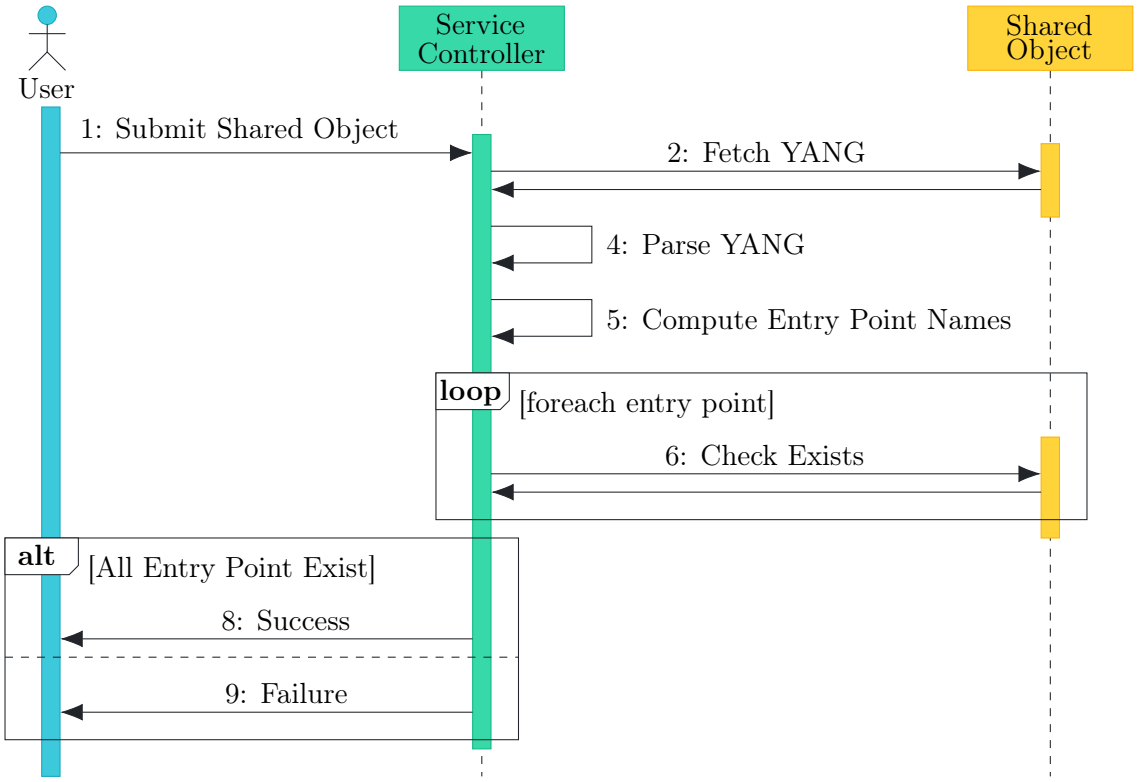

Figure 4.2. Run-time Service Registration

```
5
6  {
7    "protocol": "lib",
8    "service": "/usr/lib/polycube/services/myservice.so"
9  }
```

Listing 4.6. HTTP request for registering a service at run-time

For convenience, instead of sending both YANG data model and shared object path with the request in listing 4.6, a shared object must contain an entry-point containing the YANG data model, which the Service Controller will parse.

The parsing process, starting from the YANG data model will then compute all entry-point names and will check if the shared object exposes all of them.

The outcome of this operation will finally reach the user, and, in the case of success, the correct REST endpoints will be created.

Entry-point names' generation

The algorithm for synthesising an entry-point name starting from the YANG data model follows some trivial steps. At first, it is added the operation name, which can be one between `create`, `read`, `update`, `replace` and `delete`. At this point, the whole tree to reach the node of interest is appended using the *snake_case* convention. For instance, assuming a `leaf` inside a `container`, the tree to reach the leaf would be `containername_leafname`. Since a YANG node might contain both dashes and dots, while C does not allow these symbols in function names, they are both replaced with an underscore. Finally, for compatibility with the actual naming convention, a `by_id` is appended.

It is important to notice that there exist two other entry points defined for `lists`: `listname_list` and `listname_by_id_get_list`. The difference between the `_list` version and the standard one is that the former operates on the list as a whole, while the latter operates on a single element of the list. The `_get_list` variant is instead used for the help feature, and it is intended to return a list of instantiated `key` values.

4.2.2 Input Validation

Input data validation is a crucial aspect of the `polycube` framework and, as such, the former architecture already implemented it.

The most significant difference between the previous architecture and the current one resides in the responsibility of each component. In a nutshell, the input validation has now been moved into the `polycube` daemon instead of being part of the auto-generated code for each service.

Figure 4.3 depicts how the Service Controller handles the input validation process.

Upon HTTP request, the *REST API* module will request the *Input Validation* module to validate each path parameter and the body of the request across restriction imposed by the YANG data model.

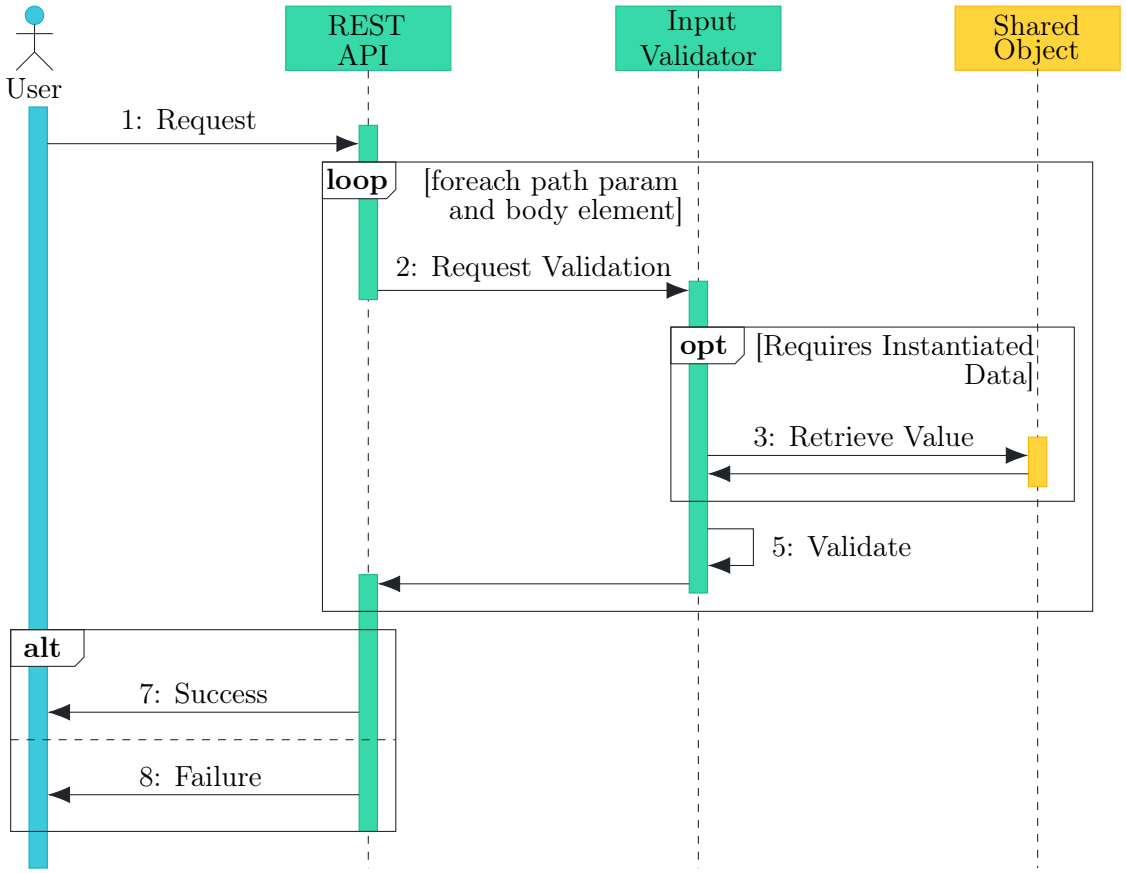

Figure 4.3. Input Validation

If the restriction requires instantiated data (e.g., a `when` clause), the *Input Validation* module will fetch it from the shared object and then proceed with validation.

Finally, the *REST API* will answer the user with a list of error messages (one for each check not passed) in the case of failure or will process the request and then answer accordingly.

Chapter 5

Implementation

Chapter 4 presents a high-level overview of a programming language agnostic architecture. This chapter aims to provide implementation details for the aforementioned architecture.

The programming language used for this work is C++ (§5), chosen for two main reasons:

1. Average better performance in terms of time and space, compared to other high-level programming languages.
2. Compilation dependencies. The rest of the `polycube` daemon is already implemented in C++, so no other compiler/interpreter is required.

Since the implementation is extremely complex, the component diagram in figure 5.1 presents a summarised view of the system, which will be further analysed along this chapter.

First of all, `libyang`, `flex`, `bison` and `pistache` are external libraries covered respectively in 3.2, 3.4 and 3.7.

The implementation revolves around the concept of *resource*, which is a 1:1 mapping to YANG data model nodes and that comes in two flavours: *body* resource and *endpoint* resource. The goal of the *body resources* is to process the body request whilst endpoint resources extend the corresponding body resource by providing REST APIs for manipulating it. Moreover, endpoint resources process request

path parameters. The separation between body and endpoint resources is required to handle resources without a mapped REST endpoint, such as the descendant of an `action` node.

Each resource internally holds a mapping between a field (i.e. a path parameter or a body field) and a set of *validators*, whose goal is to enforce YANG defined restrictions. Upon request, a resource will invoke the right *validator* for each path parameter and each body field. The validator will, if required, request to the shared object the instantiated value, then will proceed with the validation.

The *XPath Parser* is a special-purpose validator which exploits flex and bison to evaluate an XPath and then enforce the restrictions as a regular validator.

Finally, the *Yang Parser* takes as input a YANG data model, generating the right resources accordingly.

Programming Language

This work has been implemented using the C++ 17 standard. The main reason to use it is the `string_view` that is a non-owning object referring to a constant contiguous sequence of `char`-like objects with the first element of the sequence at position zero.

A typical implementation holds only two members: a pointer to constant `char` and a size.

The main advantage of the `string_view` over classical `string` is that the `substring` operation just performs two arithmetical operations: addition on the pointer and a subtraction on the size; thus, returning just a portion of the original data. Instead, `string` performs a copy of the characters in the substring. It is clear that the `string_view substring` has an $O(1)$ complexity, while the `string` one has an $O(n)$ complexity. Since both pistache and the JSON parsing library do use `string_view` when available, allowing it provides a great performance improvement.

Two other good reasons to use C++ 17 are a native reader-writer lock, useful to provide thread-safety with a higher throughput and the `if constexpr` construct

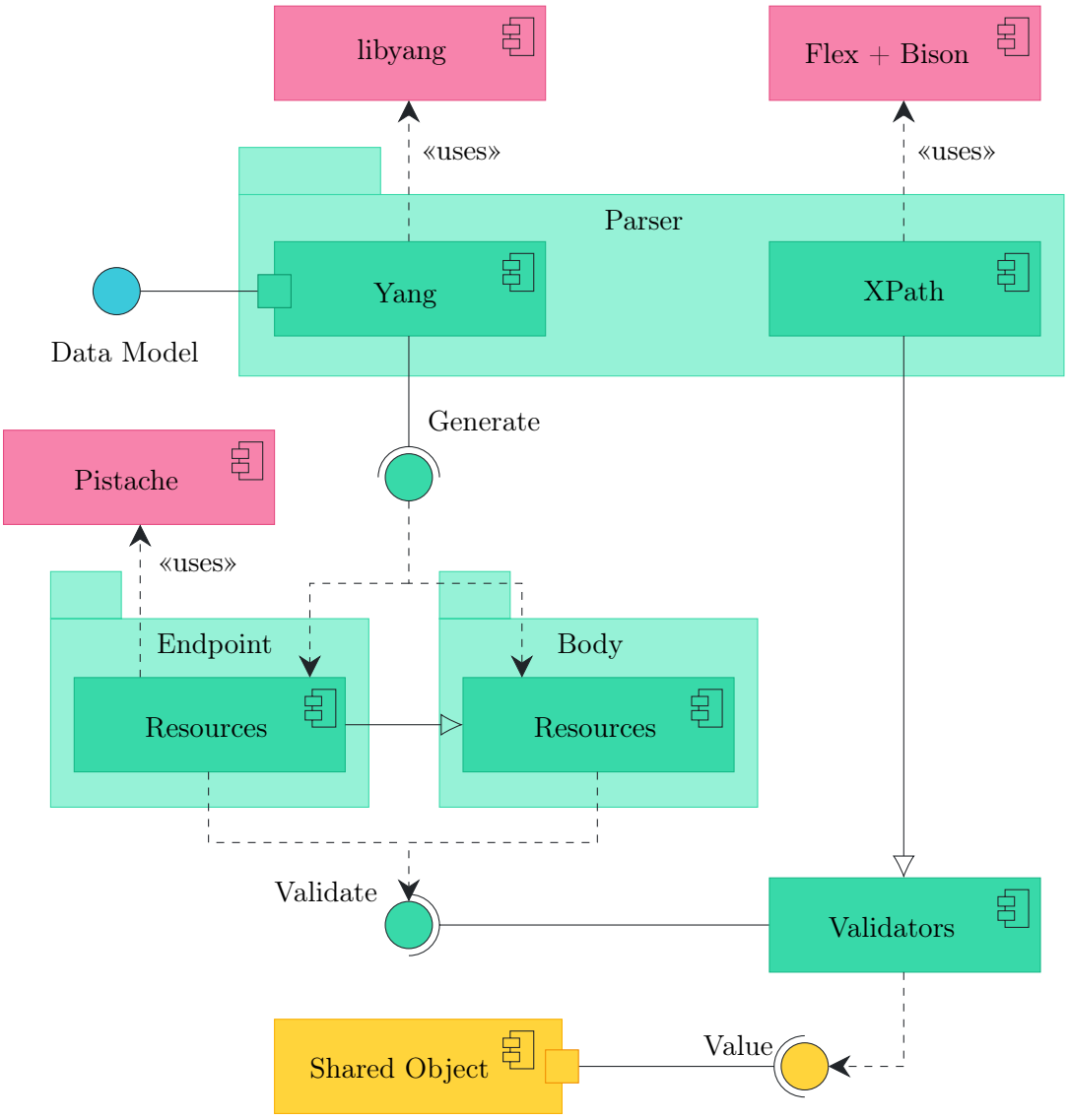

Figure 5.1. High-Level Implementation Overview

that allows the compiler to compile only the correct branch of the `if-else` statement.

5.1 Class Hierarchy

Now that it is clear what the main components are, it is possible to analyse in detail each component.

5.1.1 Body Resources

Each class in this component is abstract as it delegates the reading functionality to the concrete class implementing it in a protocol dependent fashion (§5.1.4).

Figure 5.2 depicts a simplified class diagram of the body resource component. It has been stripped of all internal operations and all non-relevant specialisations (such as `ListResource` that extends `ParentResource` and manages the YANG `list` node).

What it is important to notice is that the component is developed in a polymorphic way: all the components except the body resource one will refer to its object only through the `Resource` abstract class. The purpose of this class is to create a mapping between the parsed YANG nodes and the body of a REST request and to validate it. In particular, each node is a different named body field.

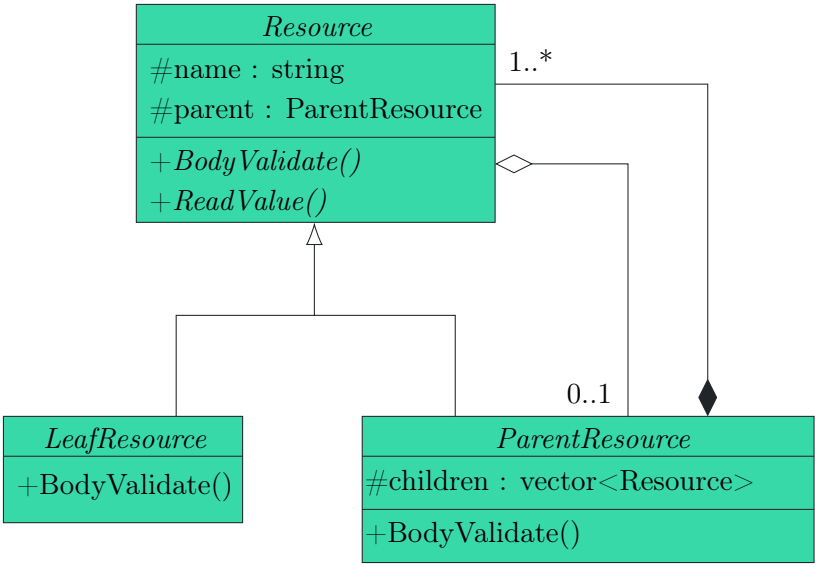

Figure 5.2. Body Resource simplified class diagram

The `Resource` abstract class exposes an operation to read the value upon the instantiated data; moreover, each resource has exactly one parent, with the exception of the root resource (i.e. the service).

Finally, the `LeafResource` is in charge of handling the YANG `leaf` scalar types, while the `ParentResource` is in charge of handling complex body requests by passing the body field of interest to the correct child.

Body Resources Implementation

Listing 5.1 shows how a `ParentResource` handles the body validation process.

First of all, it iterates over every child. For each child, it checks if it was marked as `mandatory` in the YANG data model and whether it is present into the request body. In the case of a missing mandatory field, an error is added to the response body. Similarly, if the field is present in the body request but it was marked as `config` in the YANG data model, an error is added since configuration nodes are meant to be read-only.

```
1  std::vector<Response>
2  ParentResource::BodyValidate(nlohmann::json &body,
3                               bool check_mandatory) const {
4  std::vector<Response> errors;
5  for (auto &child : children_) {
6    if (body.count(child->Name()) == 0) {
7      if (check_mandatory && child->IsMandatory()) {
8        errors.push_back({ErrorTag::kMissingAttribute,
9                          child->Name().data()});
10     }
11   } else {
12     // non configuration nodes are read only
13     if (!child->IsConfiguration()) {
14       errors.push_back({ErrorTag::kInvalidValue,
15                         child->Name().data()});
16     } else {
17       auto child_errors =
18            child->BodyValidate(body.at(child->Name()),
19                                check_mandatory);
20       errors.reserve(errors.size() + child_errors.size());
21       // remove current parsed element from the body.
22       // required for detecting unparsed elements,
23       // that may be typos
```

```
24          body.erase(child->Name());
25          std::copy(std::begin(child_errors), std::end(child_errors),
26          std::back_inserter(errors));
27        }
28      }
29    }
30
31    errors.reserve(errors.size() + body.size());
32    for (auto &unparsed : body.items()) {
33      errors.push_back({ErrorTag::kInvalidValue, unparsed.key().data()});
34    }
35
36    return errors;
37    }
```

Listing 5.1. `ParentResource` Body Validation

At this point, the request body field associated the current children is passed to it, which will, in turn, validate it.

Every validated body field, whether the validation was successful, is removed from the body request. Once the `ParentResource` has iterated over all of its children, if there is a remaining field into the body, it is marked with an error as it might be a typo.

As it is clear, the `ParentResource` only checks for mandatory and configuration children; then it delegates the actual validation to its children. If the child happens to be a `ParentResource` as well, the process is repeated recursively. Instead, when the child is a `LeafResource`, it uses the implementation in Listing 5.2.

```
1   std::vector<Response>
2   LeafResource::BodyValidate(nlohmann::json &body,
3                              bool check_mandatory) const {
4   std::vector<Response> errors;
5   if (body.empty()) {
```

```
6     errors.push_back({ErrorTag::kMissingAttribute, name_.data()});
7     return errors;
8   }
9
10  auto field = field_->Validate(body);
11  if (field != kOk) {
12    errors.push_back({field, name_.data()});
13  }
14  return errors;
15  }
```

Listing 5.2. `LeafResource` Body Validation

The body emptiness check is useful only if the function is invoked from the leaf REST handler, as there is assured to be existent when invoked by a `ParentResource`.

The most important part of the body validation process in a `LeafResource` is on line 10, in which it passes the body to a `JsonBodyField` (§5.1.3) that is in charge of performing the actual validation.

Finally, all body resources expose a `SetDefaultIfMissing` function which is invoked by the REST handlers. Its purpose is to add a leaf YANG defined default value if it is missing in the request body. If required, the whole tree for reaching the leaf from the resource handling the request is added.

5.1.2 Endpoint Resources

Similarly, to body resources, each class in this component is abstract as well. In particular, the top-most abstract class is `Resource`, which provides REST request validation functionality and delegates the writing and deletion functionalities to the concrete class implementing them in a protocol dependent fashion (§5.1.4).

The component will then exploit the *multiple inheritance* mechanism provided by the C++ programming language. As it is possible to see in figure 5.3, like in the body resource (fig. 5.2), there are the `LeafResource` and the `ParentResource`

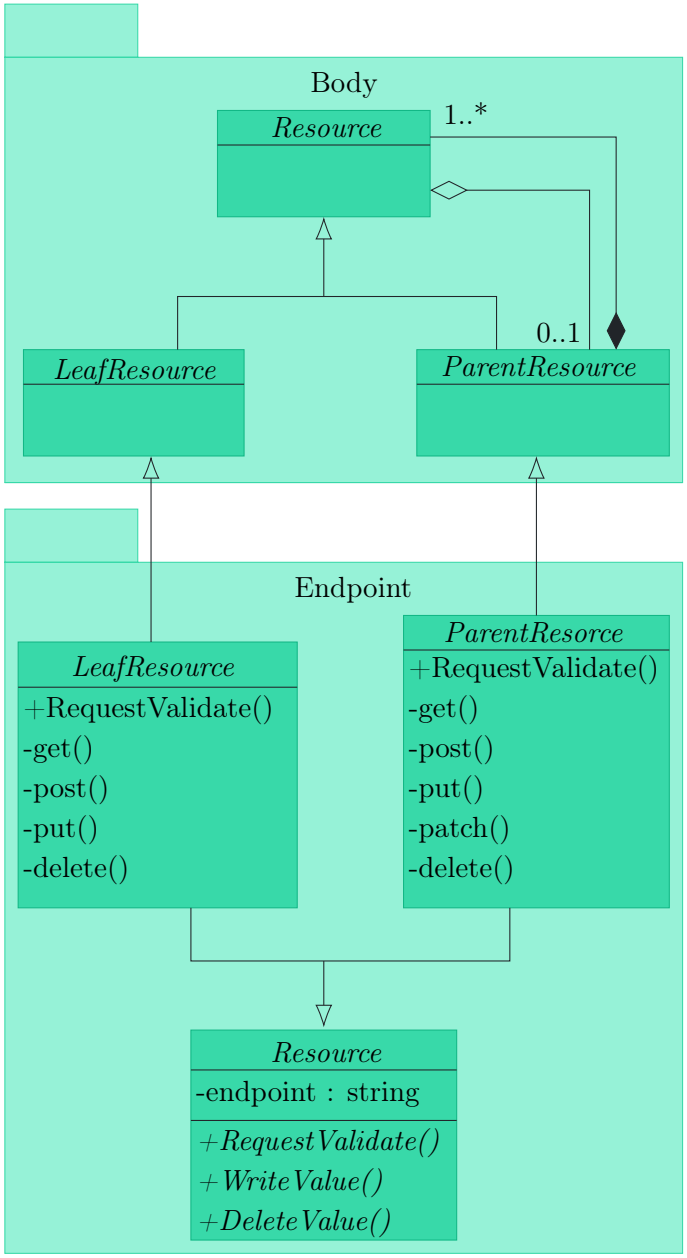

Figure 5.3. Endpoint Resource simplified class diagram

which extend the endpoint `Resource` and their body correspondent. Through this structure it is always possible to refer to a class by means of the `Resource` (either the body or the endpoint one), providing greater flexibility.

While the body validation functionality is inherited from the body resource, these classes use the *Pistache* library (§3.7) to provide REST APIs.

REST Endpoints Generation

The REST endpoints are derived from the YANG data model following a straightforward algorithm.

The first path segment is the service name, followed by a path parameter containing the requested instantiated cube name. At this point, the YANG data model is parsed with a top-down approach. The parser will produce a path segment using the name of any `container`, `rpc`, `action`, `leaf` and `leaf-list` encountered. Similarly, `list` nodes will produce a path segment corresponding to their name, followed by a path parameter for each `key`.

Taking the listing in 4.1 as an example, the endpoints would be:

```
/example/:cube_name/
/example/:cube_name/ports/
/example/:cube_name/ports/:name/
/example/:cube_name/ports/:name/ip/
```

where the segments starting with a colon are path parameters.

It is important to notice that, while the `HTTP GET` handler is always registered with each endpoint, the registration of the remaining handlers depends on whether the resource is a *configuration resource* or a *state resource*: if the node is *configuration* all manipulation operations (`POST`, `PUT`, `PATCH`, and `DELETE`) will be present, but will not otherwise.

Pistache

As already mentioned in section 3.7, for this work the Pistache library has been used as Web Server with REST features. Nevertheless, it has been modified with this work both for performance improvement and for missing features.

The performance improvement concerns the routing algorithm used by Pistache, meaning how it does select the correct handler for the incoming request. Before the modification, the Pistache library used a linear search, which can be summarised as follow:

1. Explode each registered route (request URI) and the ongoing one in path segments.
2. For each registered route, compare in order all path segments
3. If all path segments match, invoke the handler; otherwise go back to step 2.

While this approach is trivial to implement, it has an $O(n \cdot m)$ complexity, where n is the total number of registered routes and m is the number of path segments in the incoming request URI.

By reorganising the known routes in a tree of path segments with nodes containing the handler to invoke, an $O(m)$ complexity was achieved. Since in a real case scenario, the daemon must handle hundreds of routes, the performance achieved is significant.

The modification regarding missing features concerns the ability to add and remove routes at run-time, required for registration and un-registration of services. This result was achieved by allowing the Pistache library to handle a non-owning reference of the router and then add to it the operations to add and remove a route which, in turn, add or remove a node and the associated handler from the path segments tree.

Endpoint Resources Implementation

Both `ParentResource` and `LeafResource` implement a trivial request validation as they only pass the request to their parent.

Instead, it is interesting how `ListResource` handles the validation process, which is shown in Listing 5.3.

```
1 std::vector<Response>
2 ListResource::RequestValidate(const Pistache::Rest::Request &request,
3                                const std::string &caller_name) const {
4 auto errors = ParentResource::RequestValidate(request, caller_name);
5 for (const auto &key_param : key_params_) {
6   auto error = key_param.Validate(request);
```

```
7     if (error != ErrorTag::kOk) {
8       errors.push_back({error, key_param.Name().data()});
9     }
10  }
11  return errors;
12  }
```

Listing 5.3. `ListResource` Request Validation

First of all, the request is passed to the parent, which allows the errors to be in the correct order. Since YANG `list` `keys` are mapped to REST path parameter, it is required to validate each of them. In particular, each key is associated with a `PathParamValidator` (§5.1.3) which will, in turn, validate the `key` across a set of validators.

5.1.3 Validators

All the components presented so far rely on a last component named `Validators` which is presented here.

Figure 5.4 extends the class structure presented in Figure 5.2 and Figure 5.3. For the sake of simplicity, all the class relationships and private members not deemed useful for this section have been omitted.

Let us begin with the `Validators` component. It is easy to understand that it provides to the external world a simple interface named `Validator` that exposes a single operation to validate a string. The validation process is then delegated to the concrete class. In Figure 5.4 it is possible to see a `PatternValidator` and a `NumberValidator`; however, other validators such as `LengthValidator` exist. The main point is that, from a resource standpoint, it does not matter what the validator actually is.

The validators are stored in a list inside a `Field` abstract class, which is in turn implemented by `JsonBodyField` and `PathParamField`. As the name suggests, the

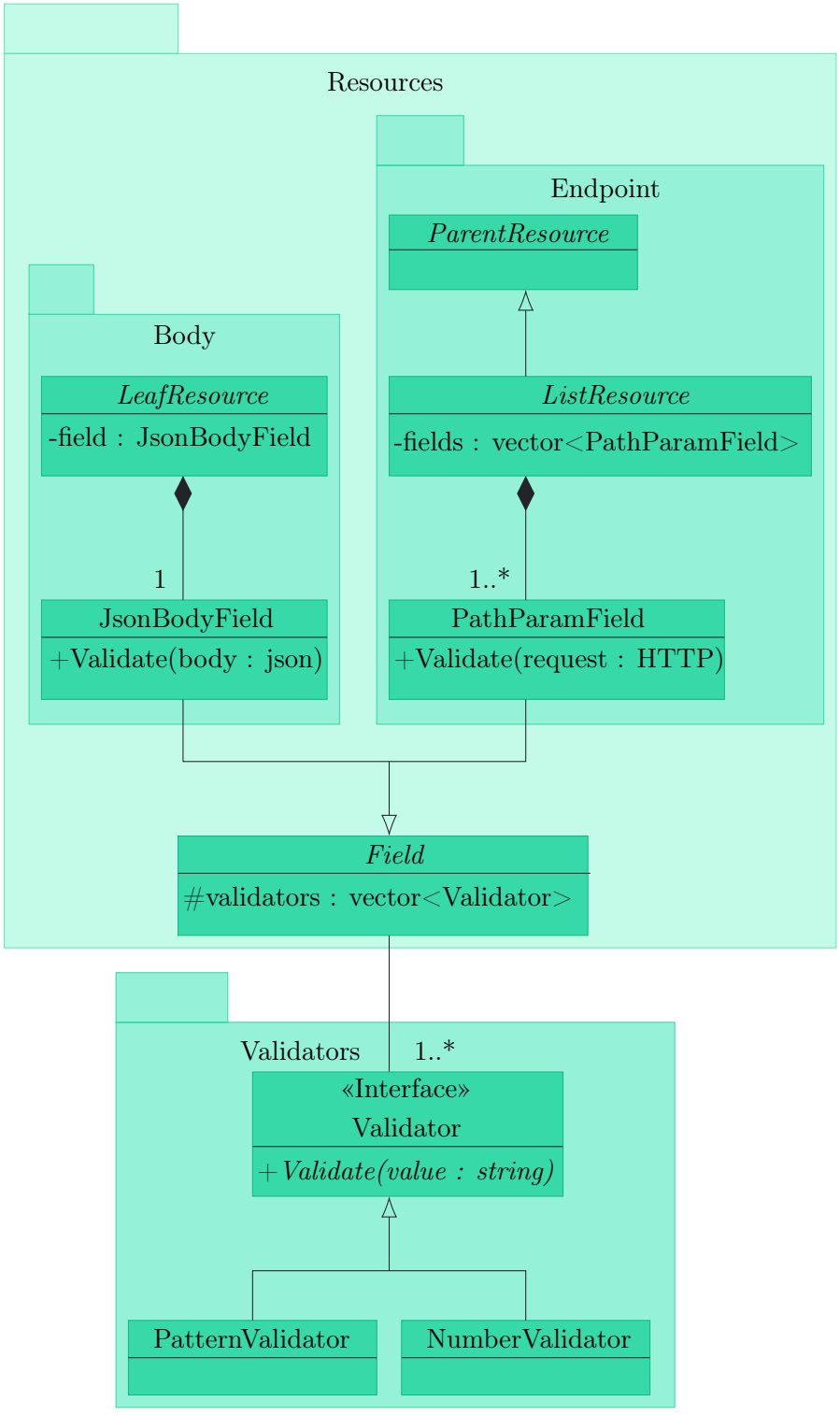

Figure 5.4. Validators simplified class diagram

purpose of these two classes is to respectively validate a JSON body field or a path parameter across the list of registered validators.

It is resource (either `LeafResource`, `LeafListResource`, or `ListResource`)

duty to handle the `Field` object lifetime. In particular, the `LeafResource` will handle a single `JsonBodyField`, the `LeafListResource` (not shown in the figure, but part of the body component) will handle a list of `JsonBodyField`s, while `ListResource` and `Service` (not shown in the figure, but part of the endpoint component) will handle a list of `PathParamField`s.

In order to implement the *Separation of Concerns* design principle, each resource can call only its directly connected validators. Upon REST request, the handler designated to manage the request (§5.1.2) will pass the request URI to its resource parent, which will validate it and then pass it to its parent until the flow reaches the top-most resource (i.e. the `Service`). Similarly, once the request URI validation is completed, the handler will pass to each child the body subsection of interest, which will, in turn, pass another subsection to its child. The descent continues until the flow reaches a `LeafReource` or a `LeafListResource` which can apply validation.

Let us take the YANG data model in listing 4.1 as an example. Let us remove the `config false` on line 9 and let us assume a POST request on `/example/ex1/ports/p1/` with body `{"ip":"1.1.1.1"}`. At first, the handler will check that `p1` is a valid port name. Regardless the outcome of the operation, it will pass the request URI to the example service, as it is its parent, which will check if `ex1` is a name for an instantiated cube of type `example`. Finally, the initial handler will pass to the `LeafResource` associated with the field `"ip"` the value `"1.1.1.1"`. The `LeafResource` will validate if `"1.1.1.1"` is a valid IPv4 address.

XPath Validator

As already mentioned at the beginning of the chapter, the XPath Validator exploits flex and bison to parse an XPath.

In a nutshell, there is an `XPathParserDriver` class holding the currently handled cube name (retrieved from the request URI) and the currently parsed resource, which at first is the one holding the REST handler. This class is passed as a parameter to the bison parser to avoid global scope pollution and thus provide thread safety.

The `XPathParser` implements a simple grammar supporting XPath path operators (`.`, `..`, `current()`, and `/`) and boolean operators. All those operators are retrieved using the `Flex` scanner. Each time the parser reduces a rule that changes the reference node in the XPath, the associated action will update it in the driver as well. For example, the action associated with the operator `..` will change in the driver the current Resource with its parent. Instead, when the parser reduces a boolean rule, it will fetch the value of the current resource by means of the `Value()` operation and will perform the requested boolean evaluation.

The parser will output a simple boolean stating whether the XPath was evaluated to `true` or to `false`, or it will throw an error if an invalid XPath is provided.

Validator Implementation

To better understand how `Validators` are implemented, Listing 5.4 shows the `PatternValidator` implementation by way of example.

Its implementation is trivial as it just compiles a regular expression to an NFA, which is then used to check whether the input request input string matches it.

```
1  namespace polycube::polycubed::Rest::Validators {
2  PatternValidator::PatternValidator(const char *pattern, bool inverse)
3      : pattern_(pattern, std::regex_constants::optimize |
4                          std::regex_constants::ECMAScript),
5        inverse_(inverse) {}
6
7  bool PatternValidator::Validate(const std::string &value) const {
8    return !inverse_ == std::regex_match(value, pattern_);
9  }
10 }  // namespace polycube::polycubed::Rest::Validators
```

Listing 5.4. `PatternValidator` Implementation

5.1.4 Extensibility

Even though this work comes with limited shared object support (namely, it supports YANG `container`, `list`, `leaf` and all types), its design allows any developer to complete the support for shared object and add any other protocol as deemed useful.

In order to achieve this result, the `Resources` component exposes an *abstract factory* whose goals is to create resources in a protocol independent fashion. It is then the protocol bridge responsibility to provide an implementation for each resource by implementing the `ReadValue`, `WriteValue` and `DeleteValue` operations and then providing a concrete factory implementing the abstract one.

The abstract will pick the correct concrete factory using the protocol field in the request body (see listing 4.6) which the parser will use to create all the resources. Figure 5.5 briefly depicts how the abstract factory is implemented. It introduces the `ServiceManager`, which is described in section 5.2.

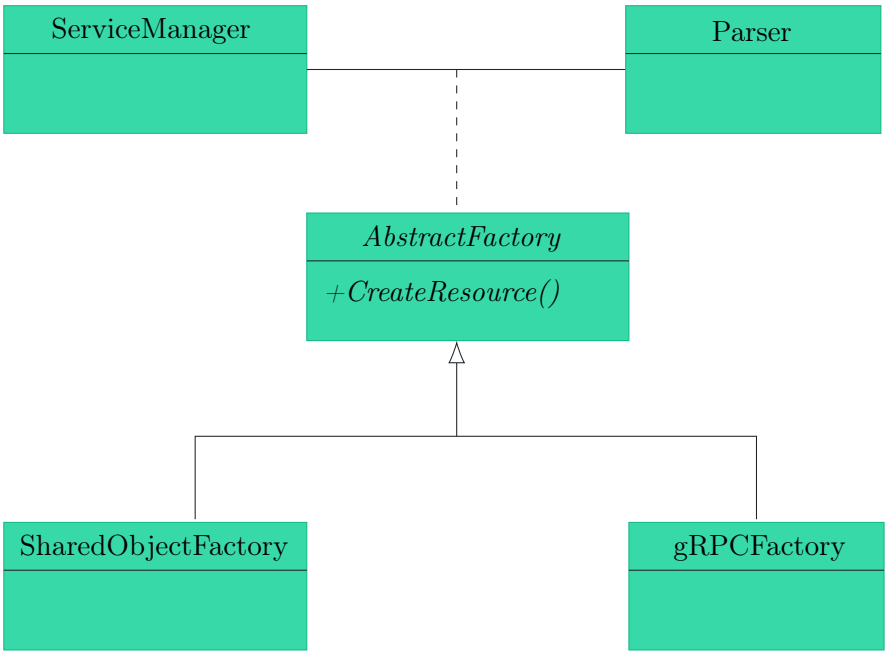

Figure 5.5. Abstract Factory

This approach provides the *Separation of Concerns* design principle as the parser must not know protocol details. Moreover, it provides improved extensibility as it is now easy to develop a new protocol, such as *gRPC*.

5.2 Service Manager

Section 4.2.1 shows the workflow to register a new service at run-time, while figure 5.1 depicts the component involved in YANG data model parsing and daemon-service communications. Still, there is one last component that merges it all and goes by the name of *Service Manager*.

The *Service Manager* exposes a REST endpoint on `/services` with two available operations: `POST` and `DELETE`.

The `POST` operation is in charge of registering a service at run-time and takes as input a `JSON` containing the protocol and service fields. Starting from this data, the *Service Manager* will create a concrete resource factory (§5.1.4) and will fetch the YANG data model from the service. Before passing the concrete factory and the YANG data model to the parser, it checks that there is not a service with the same name. If this check fails, the process fails as well, and the parser is never invoked.

The `DELETE` operation is in charge of de-registering a service starting from its name. The operation will only succeed when there is no cube instantiated of the requested service.

Since the `PUT` operation must be idempotent, it has been decided to avoid it and allow service updating by means of a succession of `DELETE` and `POST`.

Chapter 6

Validation

This chapter aims to provide a comparison between the old and new architecture in terms of required resources. In particular, it takes into account the differences in required storage as well as some relevant timings.

6.1 Storage Requirements

Both the new and the previous architecture relies on Pistache external library for providing REST APIs. At state of the art, Pistache compiled in release mode takes 2.4 MiB. Since it is a statically linked library, not all of its code gets copied into the main program by the linker but only the used one. The service controller provided with this work takes around 1.7 MiB (before the integration with the daemon).

In order to check the size of the Pistache library inside the executable, it has been checked the size of each Pistache symbol with `nm service_controller -B -S -size-sort -demangle | grep Pistache` and then summing it all. The result is that Pistache takes $\sim$ 300 KiB.

The previous architecture linked Pistache on every service, as the REST API was directly exposed by it. Moving the REST API into the daemon had the effect of shrinking the size of each service, thus reducing the total required amount of storage.

Figure 6.1 shows a comparison of storage requirements between the previous

and the current architecture, assuming 1 MiB services and no restriction imposed by the YANG data model.

As it is easy to see, when more than four services are deployed, the current architecture requires less storage.

Figure 6.1. Storage Requirements Comparison

It is important to notice that the graph regarding the previous architecture shows only the size of the services, while the graph regarding the current architecture shows both the size of the services and the size of the service controller (including `libyang`).

Moreover, the graph regarding the current architecture should be flatter as the code of the REST handler is no longer present. Finally, in a real case scenario, the difference between the two graphs would be more significant, since services would contain validators' code as well; in fact, while in the current architecture the requirement for REST handlers and validators is constant, in the previous architecture the required size increased linearly with the number of defined endpoints and YANG defined restrictions.

6.2 Services' Compiling Time

Since most of a service code resides in user implementation, removing the input validation and the REST API does not produce a significant reduction in compiling

time.

Nevertheless, the main advantage of the current architecture is an improved resilience to YANG data model changes. As a matter of fact, the following changes would produce a recompilation only of the translation unit containing the YANG data model shared object entry point:

- The allowed value space defined by a `range`, `length`, or `pattern` statement may be expanded.
- A `default` statement may be added to a `leaf` (either directly or indirectly through its type), or changed.
- A `must` statement may be removed or its constraint relaxed.
- A `when` statement may be removed or its constraint relaxed.
- A `mandatory` statement may be removed or changed from `true` to `false`.
- A `min-elements` statement may be removed, or changed to require fewer elements.
- A `max-elements` statement may be removed, or changed to allow more elements.
- A `description` statement may be added or changed without changing the semantics of the definition.
- A `type` statement may be replaced with another `type` statement that does not change the syntax or semantics of the type. For example, an inline type definition may be replaced with a `typedef`, but an `int8` type cannot be replaced by an `int16`, since the syntax would change.
- Any set of data definition nodes may be replaced with another set of syntactically and semantically equivalent nodes. For example, a set of `leaf`s may be replaced by a `uses` statement of a grouping with the same leafs.

6.3 Service Deploy Timing

The most time-consuming operation is the run-time service registration presented in section 4.2.1.

In GNU/Linux, shared objects use the ELF (*Executable and Linkable Format*). The `dlsym` function will use a hash function to resolve the bucket in which the function is placed. Each bucket contains a linked list of functions whose hashes are in conflict. The search average complexity on this structure is an amortised $O(1)$, which is a great result; nevertheless, it is significantly affected by cache misses. Therefore, the run-time service loading mostly depends on three factors:

1. Cache size
2. RAM speed
3. Storage speed

Maximising these three factors improves the service loading time sharply, along with better system overall performances.

Even though shared object loading is the slowest operation during run-time service registration, it takes $\sim 50\%$ of the total deploy time. As a matter of facts, it still is required to parse the whole YANG data model, register the REST endpoints and generate the validators. In particular, the construction of the `PatternValidator` is quite expensive as regular expression are compiled with the `optimize` flag. While on the one hand, this flag makes the matching faster, on the other hand, it involves the conversion from NFA (*nondeterministic finite automaton*) to DFA (*deterministic finite automaton*) during construction.

Taking the `NAT` YANG model shipped `polycube` as an example, inspecting the shared object with `readelf -dyn-syms libpcn-nat.so`, we get that it exposes 160 dynamic symbols.

Measuring the time required to deploy an empty `NAT` service (only the entry points are defined, no control plane) on an Intel® Core™ i7-4770@3.40GHz CPU (caches: 32 KiB L1, 256 KiB L2, 8 MiB L3) with DDR3@1600MHz CL11 RAM and 7200 RPM Hard Drive we get ~ 37 *ms* to load it completely.

As expected, the time to register a service at run-time is higher than the one of the previous architecture, which was $\sim 1\ ms$. This huge difference is due the fact that in the previous architecture YANG parsing was performed ahead of time, and most of the logic was in the service itself, thus reducing the number of entry-points.

6.4 Input Validation Timings

Input data validation timings are highly variable since they strictly depend on the restrictions defined in the YANG data model.

The trivial case in which there is no restriction has a negligible overhead since it would require to iterate on an empty list of validators.

Most of validators will still have a negligible overhead (e.g., `NumberValidator` performs comparisons between numbers, which are trivial operations). The validator with the highest impact is the `PatterValidator`, which may be in charge of validating complex regular expressions.

The `ipv4-prefix` in the IETF YANG module containing data types for Internet addresses (`ietf-inet-types`) is defined by means of the following regular expression:

```
(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}
([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])/
(([0-9])|([1-2][0-9])|(3[0-2]))
```

The time required to validate this regular expression on an Intel® Core™ i7-4770@3.40GHz CPU is $\sim 80\mu s$.

Chapter 7

Conclusions

The solution proposed with this thesis is far from complete. First of all, it is required feedback from service developers to understand whether the proposed architecture is deemed easy enough to use or another refining iteration is required. Still, this solution is headed in the right direction since it removes all REST handler and validation from the service, thus allowing the service developer to focus only on core logic.

Features vs Performances

This work mainly focused on features instead of performance; it is no accident that the run-time service deployment (§6.3) takes a significant amount of time, compared to the previous architecture in which the YANG parsing was made ahead of time.

Nevertheless, $\sim 37ms$ is a negligible time, considering that service deployment is a minimal time compared to the total service lifetime. Moreover, if in future the run-time service deployment will become a time-critical feature, there is a vast area for performance improvements.

For instance, at state of the art, the parser will re-generate the validators for each new service; however, some data-types such as the IETF INET ones are very common and reusable: parsing them at daemon startup and reusing them without recompiling all the regular expressions all the time would reduce the service deployment time. Moreover, pre-generating them would lead to a significant reduction of

main memory since they would be shared between services.

This improvement would require a minimal code change; nevertheless, it is possible to improve performances even more by writing `Validator` specialisations for INET types, thus moving the validator generation from run-time to compile-time.

Thread-safety

An issue that raised with the new architecture is the thread-safety. With the previous architecture, since the YANG data model was parsed ahead of time and the service code generated accordingly, it was possible to provide a very fine lock mechanism.

Instead, since the new architecture must handle all possible YANG data models, it can only provide a coarser locking mechanism. With this work, the only thread-safety guarantee is during service deploy by means of an exclusive lock into the `ServiceManager`. This means that services cannot safely accept multiple configuration commands at the same time; they must be serialized and executed in sequence, without concurrency.

Nevertheless, thread-safety should be analysed very carefully to provide the best solution. One possible solution is to implement a per-cube reader-writer lock which would read-locked by the `GET` operations inside a cube and write-locked by all the other operations. While this locking mechanism may provide high throughput, it may be deemed not enough; thus, the best locking mechanism must be designed carefully.

YANG Data Modeling Language Coverage

As already mentioned, this work was meant to provide an initial prototype to determine if it was headed in the right direction and, as such, it requires improvements. As a matter of fact, likewise the original implementation, it supports only a subset of the YANG nodes.

In particular, all YANG nodes are implemented as abstract classes, but the Shared Object protocol support is limited to `container`, `list`, `leaf`, and `action`. Similarly, the XPath parsing only supports boolean and path operations, so it is required to implement the remaining operation for full coverage. Those missing features have not deemed a priority during the implementation and testing phase of this work as they were missing in the previous version as well and thus not used in any `polycube` defined service.

Nevertheless, a full YANG coverage is a crucial requirement to allow services' developers to exploit the YANG data modeling language fully.

Persistent Configuration

At state of the art, when the daemon gets shut down and then restarted, it is initialised at an empty state in which it is required to reload and re-configure each cube. Implementing configuration persistence would lead the daemon to check whether there is any service loaded from a previous session and reconfigure them accordingly, thus solving the empty-state problem.

In the previous architecture, the services existed semi-independently from the daemon, thus it was not possible to persist cubes' configuration from the daemon; the only possible approach was to let each service to persist their data; nevertheless, since services were deployed ahead of time, not the whole daemon but only the services were in an empty state.

With the new implementation, the daemon is aware of the configuration of each cube, since it can extract data from them. It is therefore possible to add an ad-hoc REST endpoint to save the configuration of the whole `polycube` framework manually, possibly adding an automatic timed save explicitly set into the configuration file.

Multi-protocol

Since initial design, `polycube` was intend to support multiple protocols to let the daemon interact with the services; namely, the original idea was to support services

contained in a shared object and the gRPC protocol. However, the previous implementation neither did support multiple protocols, nor its design allowed a smooth integration with protocols different from the shared object one.

Even though the new implementation does not implement multiple services as well, its design takes into account this requirement, thus making multiple protocol support straightforward to implement.

Bibliography

[1] M. Bjorklund, E. Bjorklund, and Tail-f Systems,
«The YANG 1.1 Data Modeling Language»,
RFC Editor,
RFC 7950,
Aug. 2016.
[Online]. Available: `https://www.rfc-editor.org/rfc/rfc7950.txt`.

[2] Czech Educational and Research Network,
libyang,
[Online]. Available: `https://github.com/CESNET/libyang`.

[3] World Wide Web Consortium,
XML Path Language (XPath),
[Online]. Available: `https://www.w3.org/TR/1999/REC-xpath-19991116/`.

[4] The GNU Project,
GNU Bison,
[Online]. Available: `https://www.gnu.org/software/bison/`.

[5] V. E. Paxson,
Flex,
[Online]. Available: `https://github.com/westes/flex`.

[6] R. T. Fielding,
«Architectural Styles and the Design of Network-based Software Architectures»,
[Online]. Available: `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`,

PhD thesis, University of California, Irvine, 2000,
Pp. 76–106.

[7] The Polycube Authors,
Swagger Code Generator,
[Online]. Available: `https://github.com/netgroup-polito/swagger-codegen`.

[8] M. Stefani,
Pistache,
[Online]. Available: `http://pistache.io/`.

[9] The gRPC Authors,
gRPC,
[Online]. Available: `https://grpc.io/`.

[10] ECMA International,
The JSON Data Interchange Syntax,
[Online]. Available: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[11] S. Miano, F. Risso, B. Matteo, and M. Vásquez Bernal,
«Toward Flexible and Efficient In-Kernel Network Function Chaining»,
Internal Draft.

[12] IO Visor Project,
BCC,
[Online]. Available: `https://github.com/iovisor/bcc/`.

[13] L. Lhotka and CZ.NIC,
«JSON Encoding of Data Modeled with YANG»,
RFC Editor,
RFC 7951,
Aug. 2016.
[Online]. Available: `https://www.rfc-editor.org/rfc/rfc7951.txt`.

[14] A. Bierman, YumaWorks, M. Bjorklund, Tail-f Systems, K. Watsen, and Juniper Networks,

«RESTCONF Protocol»,

RFC Editor,

RFC 8040,

Jan. 2017.

[Online]. Available: `https://www.rfc-editor.org/rfc/rfc8040.txt`.

Acknowledgements

There many of people that I want to thank. Without them, today I would not be at this important goal of my academic career.

At first, I want to thank my supervisor, professor Risso, as he allowed me to work on such interesting and innovative topic. He has been very willing to help during all the phases of this work, giving me precious advice and opportunities.

I want to thank Sebastiano Miano and Mauricio Vásquez Bernal for their crucial support that let me understand the polycube framework architecture and identify where my work should have been placed.

A special thanks to all my friends, that made these years so far from home more affordable. To Mattia, the one who is always ready for a nerdy idea, even if he forgets it after a few minutes, and his *Nascondini*. To Uccio, also known as Vincenzo: the one who we all always miss, and to his (no longer) fish. To Gius(t)eppe, the righteousness made man. To Federico, an emotional punching bag whose only purpose is complaining about how much life is bad. To Piero, whose *odi et amo* relationship with Federico created the most hilarious situations. To Toppa, Francesco for friends, the one who works always claiming to be the poorest of us all so that he doesn't have to swipe his card. To our mum Vergona, the man who doesn't know the word anxiety. To Flavio, the only one that understands how important is to wear a helmet all the time, even though he is afraid of kites.

A heartfelt thanks to my beloved Benedetta, who always walked beside me, regardless of the distance separating us. She gave me the strength to keep going by supporting (and tolerating) me. If it weren't for her, this thesis would be half as good as it is, since she spent a lot of time reading it and telling me what was unclear.

Finally, I want to thank my family, and especially my father who assisted me along the way both financially, allowing me to focus only on my studies, and emotionally, always pushing me to do my best and never surrender to difficulties.