Deliverable 4.4-pub: Public DevOpsPro code base

Dissemination level	PU
Version	1.0
Due date	N/A
Version date	30.05.2016

This project is co-funded
by the European Union

Document information

Editors

Felicián Németh (BME), Wolfgang John (EAB)

Contributors

Per Danielsson (SICS), Jan Ekman (SICS), András Gulyás (BME), Per Kreuger (SICS), Shaoteng Liu (SICS), Guido Marchetto (Polito), Felicián Németh (BME), Bertrand Pechenot (ACREO), István Pelle (BME), Sachin Sharma (iMinds), Riccardo Sisto (Polito), Pontus Sköldström (ACREO), Serena Spinoso (Polito), Rebecca Steinert (SICS), Matteo Virgilio (Polito)

Coordinator

Dr. András Császár
Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH)
Könyves Kálmán körút 11/B épület
1097 Budapest, Hungary
Fax: +36 (1) 437-7467
Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme
FP7-ICT-2013-11
Collaborative project
Grant Agreement No. 619609

Summary

This deliverable contains the publicly shared code developed during the prototyping activities in task T4.4. These prototyping activities led to DevOpsPro, the integrated prototype of Work Package 4. DevOpsPro is based on multiple tools that are useful on their own, but together they support the monitoring and management processes and are part of the realization of the UNIFY Architecture. For detailed description of the tools we would like to refer to Deliverable 4.3, whereas this auxiliary document list requirements, provides installation instructions, gives usage examples and howtos, and presents sample results for each tool. Note that this version of the deliverable (D4.4-pub) is the shortened, public version of D4.4. It only documents the SP-DevOps tools and software components that have been publicly released.

Contents

List of Figures

List of Tables

1 Introduction

Besides theoretical research results, the UNIFY project has also put emphasis on prototyping activities to verify the design of the UNIFY Architecture and its components, as well as to disseminate its results. The publicly available tools[1] allow easy exploitation of the results for everybody, not just the project parters. Deliverable D4.4 primarily contains the code developed during the prototyping activities in task T4.4. These prototyping activities led to DevOpsPro, the integrated prototype of Work Package 4. DevOpsPro is based on multiple tools that are useful on their own, but together they enable novel, efficient, and automated management processes as part of the realization of the UNIFY Architecture. Deliverable 4.3 contains the detailed description of the tools [D4.3], whereas D4.4 lists requirements, provides installation instructions, gives usage examples and how-tos, and presents sample results for each tool. This deliverable (D4.4-pub) is a shortened, public version of D4.4. It only documents the SP-DevOps tools and software components released publicly.

1.1 List of SP-DevOps tools and functions

The tools follow the ordering presented in D4.3, as Table 1 shows. During our prototyping task, we followed an agile development approach, and released different iterations of the tools which are part of the SP DevOps Toolkit. However, not all of the tools developed during the project are released publicly and therefore the private ones are omitted from the public version D4.4-pub.

As discussed in D4.3, the Two-Way Active Measurement Protocol (TWAMP) [RFC5357] is used to measure different network performance parameters with the help of probe packets. However, the lack of standard management frameworks for TWAMP resulted in various proprietary mechanisms. The UNIFY project contributed to the TWAMP data model [Civ+16] that avoids vendor lock-in by standardizing the management of TWAMP measurements. The data model is defined in YANG and does not really represent a program code contribution as the other components

Table 1: List of SP-DevOps tools as documented in D4.3 and D4.4-pub

Implemented SP-DevOps tools and functions	D4.3	D4.4-pub
VeriGraph	Section 4.1	Section 2
Verification of Path Functionality	Section 4.2	(removed)
Consistent Network Updates	Section 4.2	(removed)
Network Watchpoints	[D4.2, Section 5.12]	(removed)
MEASURE	Section 4.3	Section 3
TWAMP	Section 4.4	[Civ+16]
RAMON	Section 4.5	Section 4
E2E delay and loss monitoring	[D4.2, Section 5.6]	(removed)
Probabilistic delay monitoring	Section 4.6	(removed)
EPLE	Section 4.7	(removed)
IPTV Quality Monitor	Section 4.8	(removed)
AutoTPG	Section 4.9	Section 5
Double Decker	Section 4.10	Section 6
Recursive Query Engine & Language	Section 4.11	(removed)
EPOXIDE	Section 4.12	Section 7

[1] Find the UNIFY OpenSource Software contributions here: https://www.fp7-unify.eu/index.php/results.html#OpenSource

desribed in this deliverable. Therefore, the model and its description are omitted from this document but are available at https://tools.ietf.org/html/draft-ietf-ippm-twamp-yang-00.

1.2 Final DevOpsPro

The UNIFY DevOpsPro covers several prototypes developed to evaluate the SP-DevOps concept and integrated SP-DevOps tools and functions (see Table 2)[2]. Specifically, as a first prototype instance, VeriGraph has been integrated with ESCAPE to add online pre-deployment verification support to the UNIFY use-case of multi-domain deployment of complex service graphs. This project-wide integration work has only implications on the VeriGraph tool and has thus been documented in D4.3 (Section 6.1).

As second prototype instance part of DevOpsPro, the state migration use-case related to the FlowNAC prototype is supported by programmable and efficient observablity through DoubleDecker messaging (DD, Section 6) and MEASURE (Section 3). Besides the integartion of DD, the FlowNAC demonstartor allowed us to verify the applicability of our definition of monitoring functions (MF) in MF control application and multiple Observability Points (OP), in turn consisting of local control plane and data plane compontents (LCP and LDP, resp.). In FlowNAC, VNF internal information served as the relevant metric to trigger state migration, which means that one microservice component composing the FlowNAC had to act as internal MF. The ongoing evaluation of the FlowNAC demo shows that the SP-DevOps MF concept can be nicely generalized to include also VNF-internal monitoring. However, we learned that our current definition of MEASURE for automatic programmablity of MFs has deficiencies in describing this type of internal MFs and will require a refinement, which we plan as future work beyond the scope of UNIFY. The FlowNAC state migration demo and a discussion related its implications on MEASURE can be found in D3.5.

Table 2: Location of source code releases

SP-DevOps tools and functions	Source code location
VeriGraph – core	https://gitlab.com/mettiu/rest-verigraph/tree/unify
VeriGraph – Neo4JManager	https://gitlab.com/mettiu/neo4jmanager/tree/unify
MEASURE - Parser	https://github.com/Acreo/MEASURE/archive/master.zip
MEASURE - MF-IB	https://github.com/Acreo/MEASURE-MFIB/archive/master.zip
MEASURE - MMP	https://github.com/Acreo/MEASURE-MMP/archive/master.zip
MEASURE - Aggregator	https://github.com/Acreo/MEASURE-Aggregator/archive/master.zip
TWAMP data model	https://tools.ietf.org/html/draft-ietf-ippm-twamp-yang-00
RAMON	https://github.com/nigsics/ramon/archive/unify.zip
AutoTPG	http://users.intec.ugent.be/unify/autoTPG-CONTROLLER.zip
Double Decker	https://github.com/Acreo/DoubleDecker/archive/D44.zip
EPOXIDE	https://github.com/nemethf/epoxide/archive/unify.zip

As last and main instance of DevOpsPro, examples of two SP-DevOps processes have been integrated with the elastic router demo. Specifically, SP-DevOps supports the elastic router scenario with programmable observablity to trigger dynamic scalaing, as well as an automated troubleshooting process. In this demo, individual SP-DevOPs tools are mainly integrated in the infrastructure layer realized by a Universal Node (UN) domain (see Figure 1). The Local Orchestrator cooperates with a Monitoring Management Plugin (MMP), which is responsible for operating the monitoring and troubleshooting components of the UN. When the Local Orchestrator sends an update of the Network Function

[2]Note that Table 2 points to snapshots of the code that correspond to the documentations in this deliverable. You might find updated versions of most tools by browsing to the master branch of the repsective repository.

Forwarding Graph (NF-FG) to the MMP, the MMP derives its tasks by interpreting the MEASURE annotation of the NF-FG. Interpreting the MEASURE language and MMP coupled together so tightly that Section 3 discusses them together.

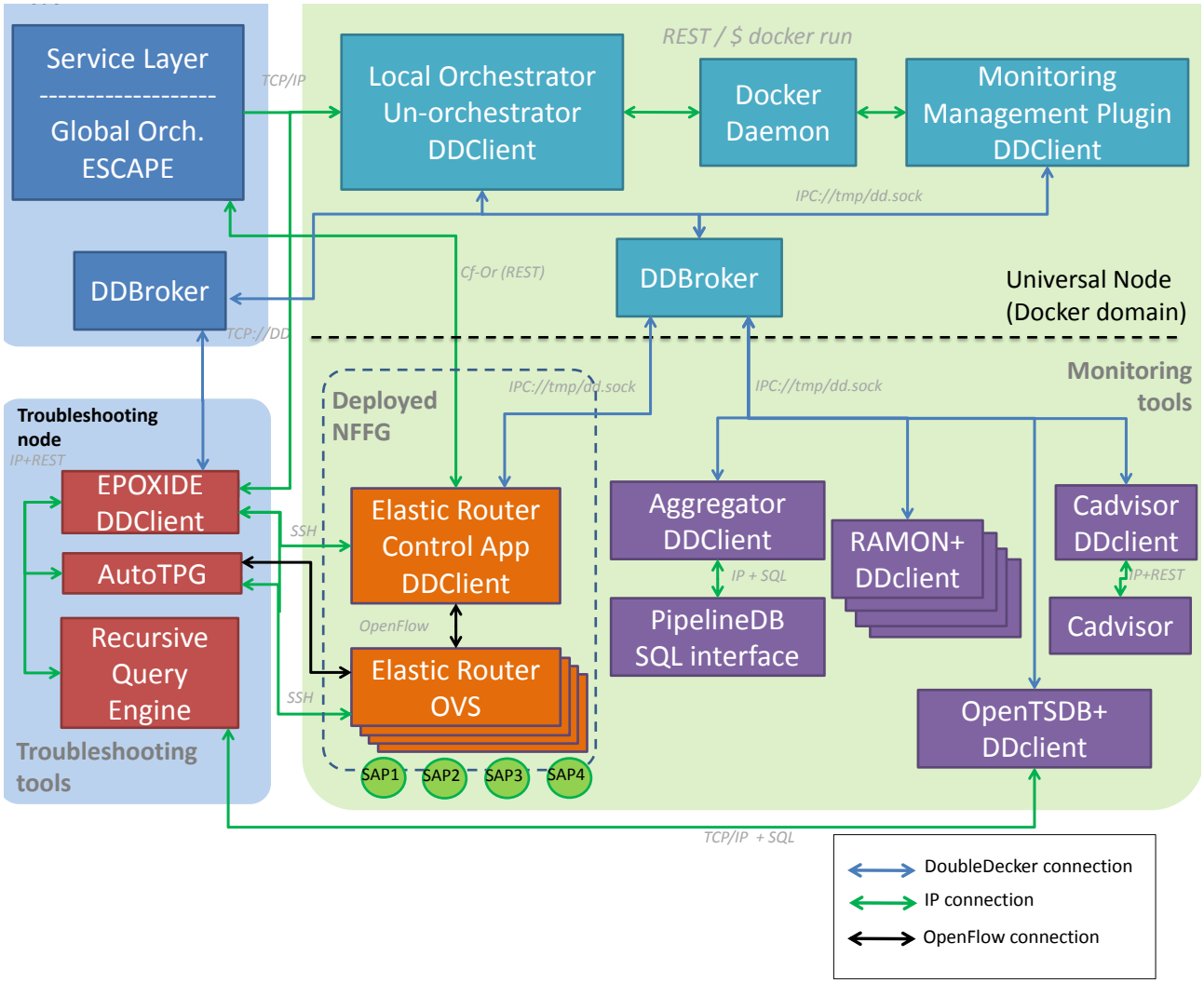

Figure 1: System view of the prototyped elastic router, depicting integrated functional blocks and their interfaces. [D4.3]

While a more detailed description of the demonstrator can be found in D4.3, we will hereby briefly provide an overview of the interfaces between the components. DoubleDecker is the distributed messaging system that helps the interprocess communication between various components. The MMP instantiates and configures RAMON (our efficient rate monitoring function); the Aggregator (a general local aggregation function for measurement result), which the the second main part of the MEASURE tool; and other publicly available components such as cAdvisor, PipelineDD, OpenTSDB.

The elastic router demonstrator also showcases a complex semi-automatic troubleshooting process. At some point during the demo, the Aggregator using the MEASURE annotation of the NF-FG detects an erroneous imbalance of resource usage and sends a trigger to EPOXIDE via the DoubleDecker message system. EPOXIDE is the multicomponent troubleshooting framework that can orchestrate a series of hypothesis tests by interconnecting different troubleshooting tools. In the demo, EPOXIDE first queries the Global Orchestrator for deployment information and delegates fault localization tasks to the Recursive Query Engine (RQE) and AutoTPG. RQE narrows down the fault location to one UN by

calculating aggregated performance metrics derived from data queried from hierarchical time series databases. AutoTPG in turn generates test packets to examine candidate switches of the UN determined by RQE and finds a flow entry responsible for the imbalance. As Figure 1 shows, the communication between the components happens either via DoubleDecker or REST API calls. The actual information is always encoded JSON objects. The interface definitions are given in the section of the tool that provides the API. For example, EPOXIDE orders AutoTPG to check a specific OpenFlow switch via the REST API of AutoTPG, therefore the communication interface is documented with AutoTPG (Section 5).

2 VeriGraph – a formal verification tool for complex networks containing active NFs

2.1 Overview

VeriGraph is a formal verification tool that is able to automatically verify networks containing a wide category of Virtual Network Functions (VNFs), also called active network functions. The main features of such data-plane functions are: (i) ability to modify a forwarded packet (or part of it) and (ii) ability to take traffic history into consideration (i.e. stateful functions). It is easy to observe that a huge number of VNFs fall into this category (NAT, web cache, firewall, IDS and so on) and can be combined together to form a complex Network Function-Forwarding Graph (NF-FG).

In the UNIFY context, VeriGraph is implemented as a layer that lies in between the mapping phase of the Orchestration Level (OL) and the following deployment steps. This verification layer receives the NF-FG along with the policies and the configurations from the upper levels of the architecture and it performs all the required verification processes to certify the graph consistency with respect to the provided policies.

The adopted verification logic is an extension of the approach proposed in [Pan+14], enhanced to support the above-mentioned active network functions and the use cases envisioned in the UNIFY environment. Essentially, a model of the network and the involved network functions, consisting of First Order Logic (FOL) formulas, is dynamically built by our tool in response to a verification request from the UNIFY architecture and the overall set of formulas are checked according to the provided policies. The resulting problem is fed as input to and analyzed by an off-the-shelf SAT solver which determines whether the considered policies are satisfied or not. An exception is raised to the upper layer in order to signal possible failures in the verification phase.

2.2 Requirements

VeriGraph has the following requirements:

- Linux Ubuntu 14 LTS
- Git
- ESCAPEv2 and all of its dependencies (available here: https://sb.tmit.bme.hu/escape/index.html)
- Python 2.7
- Java 7 (tested with version 1.7.0_91)
- MS Z3 SAT solver (available here: https://github.com/Z3Prover/z3)
- Apache Tomcat 8 (available here: http://tomcat.apache.org/download-80.cgi. Newer versions are not supported since they require newer version of Java)
- Neo4J graph database (available here: http://neo4j.com/download/)

In the following sections, we will detail the overall verification architecture and all the steps required to properly install the different components and get the system up and running on a Linux machine.

2.3 Tool architecture

The overall architecture of the verification module and its interaction with the ESCAPEv2 framework is depicted in Figure 2.

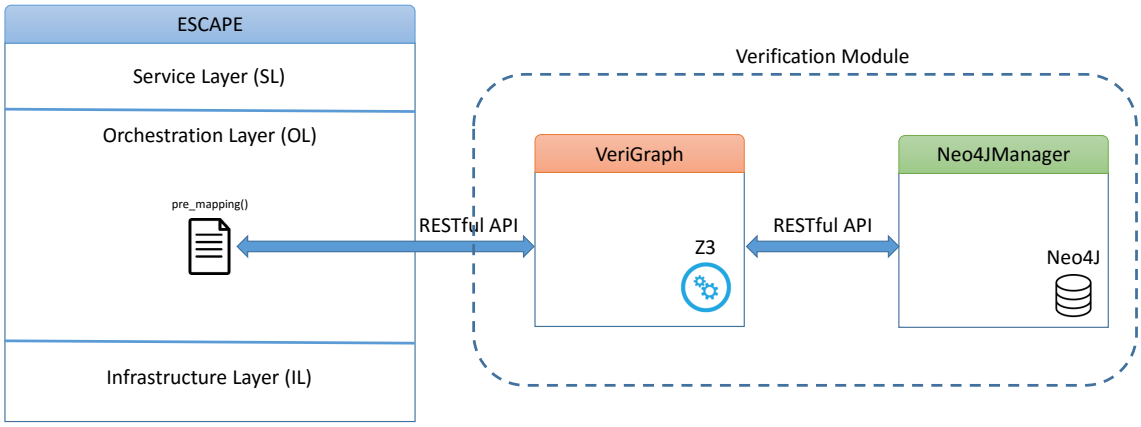

Figure 2: Verification module Architecture

Essentially, it consists of two logical modules:

- **VeriGraph** is the core component and it is externally exposed by means of a RESTful API. Its fundamental role is that of receiving the graph under deployment from the ESCAPE Orchestration Layer and decompose it into different function chainsin order to perform policy verification on each involved chain. In this context, the term "function chain" is used to indicate a non-cyclic sequence of middleboxes that starts from one source node and ends into a different destination node. As a consequence, starting from a given NF-FG, it is generally possible to extract multiple chains with the same source and destination nodes. VeriGraph also receives the high level policies that are requested by the upper layers. The full source code of VeriGraph is available at https://gitlab.com/mettiu/rest-verigraph/tree/unify;
- **Neo4JManager** is used internally by Verigraph to store the under deployment graph onto the Neo4J database and extract all the required chains from the graph based on the provided policies. This module is also able to perform some basic topological reachability checks on its own, thanks to the sophisticated API provided by Neo4J database. Notice that the reachability checks performed by Neo4JManager do not take into account nodes behaviour and their configurations, i.e. it only considers the raw topology and properties verified using the standard graph theory. Further complex checks are implemented by VeriGraph exploiting FOL formulas and VNF models, as described in Section 5.10 of [D4.2] and Section 4.1 of [D4.3]. The full source code of Neo4JManager is available at https://gitlab.com/mettiu/neo4jmanager/tree/unify.

Once VeriGraph has stored the received graph on Neo4JManager and it has queried the service to retrieve all the required chains in the graph, it generates the FOL formulas for the Z3 solver (one set of formulas for each chain) and produces a global verification result for ESCAPE in order to trigger an exception if any violation occurred.

In order to properly integrate the verification tool with ESCAPE, we implemented a GraphVerify class into the framework, extending the available AbstractMappingDataProcessor abstract class (see https://sb.tmit.bme.hu/escape/util/mapping.html#escape.util.mapping.AbstractMappingDataProcessor) contained in ESCAPE. Essentially, it provides

an extensible way to customize the pre_mapping behavior of the deployment process, i.e. the operations to be executed before ESCAPE actually maps the request onto the underneath layers in the OL. We redefined the pre_mapping_exec() polymorphic method in order to process the input_graph provided by ESCAPE and call our VeriGraph verification module.

2.4 Installation and usage

In this section we describe all the steps needed to install the verification tool and all of its dependencies inside the ESCAPE framework. We will detail how to build a self-contained virtual machine (under VirtualBox) and how to launch a service graph deployment with the verification enabled.

2.4.1 ESCAPEv2 Installation

- download the Mininet VM image at https://github.com/mininet/mininet/wiki/Mininet-VM-Images;
- import .ovf file in VirtualBox;
- launch the VM (username/password: mininet/mininet);
- type `cd /home/mininet`
- create the .ssh folder (if not existing) with the command mkdir .ssh
- copy your private RSA key to be used for the Escape Gitlab in the .ssh folder:
  - `scp <your_user>@<host_ip>:~/.ssh/<your_ssh\_key> ~/.ssh/id_rsa`
  - `sudo chmod 700 .ssh && sudo chmod 600 .ssh/id_rsa`
- clone the escape repository into a folder named "escape"
  - `git clone git@gitlab.fp7-unify.eu:Balazs.Sonkoly/escape-shared.git escape`
- `cd escape`
- type the following command `./install_dep.sh`
- run `./escape.py -help` to see all the available arguments
- Run Escape with a static example file:
  - `cd /home/mininet/escape`
  - `./escape.py -dfi -s pox/escape-mn-req.nffg`
  - to print the nffg type the following at the prompt "`ESCAPE>`":
  - `core.adaptation.controller_adapter.domainResManager._dov.get_resource_info().dump()`
  - type CTRL+D to gracefully shutdown Escape

2.4.2 Create an ESCAPEv2 custom configuration

- Create a new configuration file to enable pre-mapping code:
  - nano /home/mininet/escape/pox/ext/escape/config.json
  - paste the following snippet (note that the following configuration enables mapping only at the orchestration layer, feel free to enable it at the service layer as well; the module which is in charge of the pre-mapping code is called verification.py and will be created in the next step):

```
{
        "service": {
                "MAPPER": {
                        "mapping-enabled": false
                },
                "PROCESSOR": {
                        "module": "escape.verify.verification",
                        "class": "GraphVerify",
                        "enabled": false
                }
        },
        "orchestration": {
                "MAPPER": {
                        "mapping-enabled": true
                },
                "PROCESSOR": {
                        "module": "escape.verify.verification",
                        "class": "GraphVerify",
                        "enabled": true
                }
        },
        "adaptation": {},
        "infrastructure": {}
}
```

  - save and exit with CTRL+O, CTRL+X

2.4.3 Create module for the pre-mapping code

- `cd /home/mininet/escape/pox/ext/escape`
- get the verification library from git:
  - copy the verify directory and all of its content.
    Source here: https://gitlab.com/mettiu/verigraph/tree/unify/escape/pox/ext/escape/verify
    Destination here: `/home/mininet/escape/pox/ext/escape/verify`

2.4.4 Add custom components

- `nano /home/mininet/escape/examples/escape-mn-topo.nffg`
- CTRL+W search "supported" and add the name of the custom components in the supported list (repeat the operation for all the occurences of "supported")

- Add nat, firewall, dpi in order to correctly run this procedure

- Create a custom topology file
  - `nano /home/mininet/escape/examples/verify.nffg`
  - copy and paste the example NF-FG available at https://gitlab.com/mettiu/verigraph/blob/master/doc/topology_example.nffg

- Create a template for each custom component
  - `cp /home/mininet/escape/mininet/mininet/templates/headerCompressor.jinja2 /home/mininet/escape/mininet/mininet/templates/{CUSTOM_COMPONENT}.jinja2`
  - Replace CUSTOM_COMPONENT with firewall, nat and dpi

- Edit the ESCAPE catalog
  - `nano /home/mininet/escape/mininet/mininet/vnfcatalog-reset.py`
  - CTRL+W Catalog().add_VNF
  - after all the other components, add the following:
    * ```
      Catalog().add_VNF(vnf_name='CUSTOM_COMPONENT',
      vnf_type='Click',
      description='CUSTOM_COMPONENT DESCRIPTION',
      icon='decompress_small.png')
      ```
      where CUSTOM_COMPONENT has to be replaced with nat, firewall and dpi
  - CTRL+W del_VNFs
  - Add the custom component name to the list of elements
  - CTRL+O, CTRL+X to save and close the file
  - reset the database:
    * `sudo chmod +x /home/mininet/escape/mininet/mininet/templates/vnfcatalog-reset.py`
    * `python /home/mininet/escape/mininet/mininet/templates/vnfcatalog-reset.py`

2.4.5 Install Z3

- Follow the instructions at https://github.com/Z3Prover/z3. In essence the required steps are:
  - `cd /home/mininet`
  - `git clone https://github.com/Z3Prover/z3.git`
  - `cd z3`
  - `python scripts/mk_make.py --python --pypkgdir=/usr/lib/python2.7/dist-packages`
  - `cd build`
  - `make`
  - `sudo make install`
      ```

2.4.6 Deploy verification and Neo4JManager web services

- Get the web services WARs:
  - Get the Neo4jManager WS from:
    https://gitlab.com/mettiu/rest-verigraph/blob/unify/dist/neo4jmanager.war
  - Get the Verification WS from:
    https://gitlab.com/mettiu/rest-verigraph/blob/unify/dist/verify.war
- Install Tomcat:
  - `cd /home/mininet`
  - `wget http://mirror.nohup.it/apache/tomcat/tomcat-8/v8.0.32/bin/apache-tomcat-8.0.32.tar.gz`
  - `tar xf apache-tomcat-8.0.32.tar.gz`
  - `cd apache-tomcat-8.0.32/conf`
  - replace the tomcat-users.xml file content with the following snippet:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              si:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">
    <role rolename="tomcat"/>
    <role rolename="role1"/>
    <user username="tomcat" password="tomcat" roles="tomcat,manager-gui"/>
    <user username="both" password="tomcat" roles="tomcat,role1"/>
    <user username="role1" password="tomcat" roles="role1"/>
</tomcat-users>
```

  - `cd ../bin`
  - `./startup.sh`

2.4.7 Run ESCAPE with custom configuration and components, with verification enabled

- `cd /home/mininet/escape`
- `./escape.py --config pox/ext/escape/config.json -dfi -s examples/verify.nffg`

Inspect the output to see the verification result. In Figure 3 we report an execution of the overall service request deployment attempt, where the verification module captured a policy violation thus aborting the deployment process itself.

2.5 Full documentation of services

In this section we more closely detail the specific RESTful interface of the two main components, namely Neo4JManager and VeriGraph which have been integrated into the ESCAPEv2 framework but can also be used in other contexts given the modular approach they are based on.

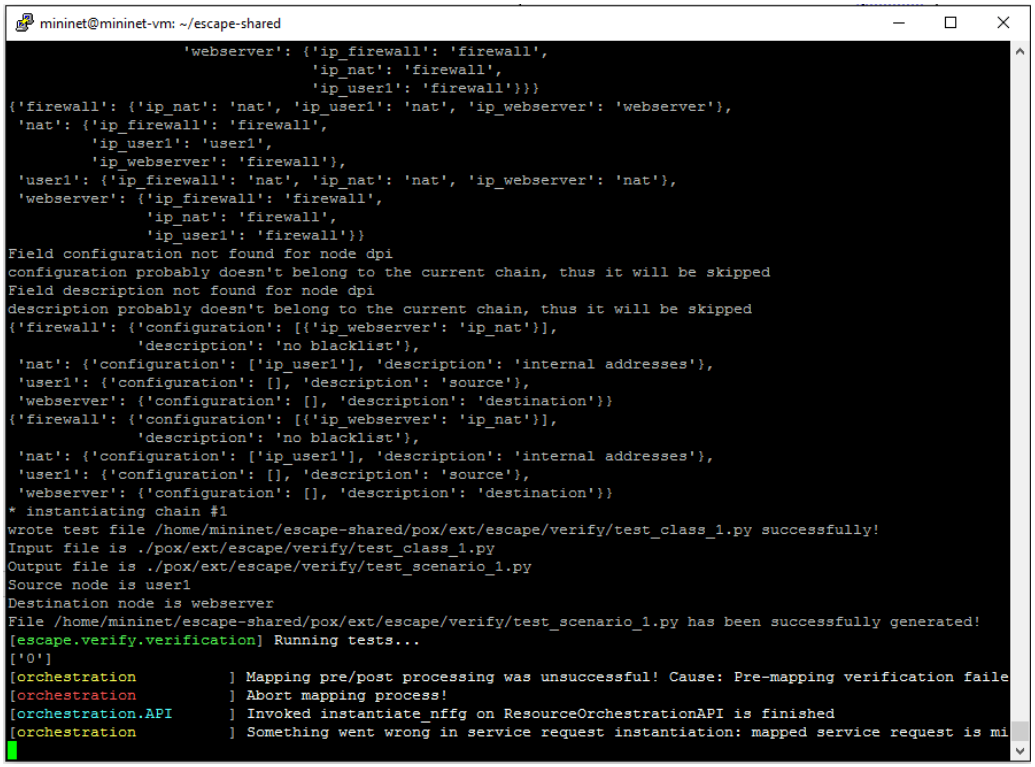

Figure 3: Policies verification in ESCAPE

2.5.1 Neo4JManager

The main operations supported by Neo4JManager are summarized in the following:

1. METHOD: POST
   URL: `${BASE_URL}/graphs`
   DESCRIPTION: This method creates a new resource representing the provided graph
   REQ. BODY: Json/XML description of the graph
   RESP. BODY: In case of success, the service replies with an OK message and the ID of the newly created graph

2. METHOD: GET
   URL: `${BASE_URL}/graphs/{graphId}/topologyProperty`
   DESCRIPTION: This method checks whether the topological property specified by means of query parameters is satisfied or not
   PARAMETERS: type -> a string specifying the type of properties. It can be: REACHABILITY, ISOLATION or NODE_TRAVERSAL
   sourceNode -> the source node
   destNode -> the destination node
   middleBox -> the node that must be traversed (used if type is NODE_TRAVERSAL or ISOLATION)
   RESP. BODY: Json/XML including the verification results

3. METHOD: GET
   URL: `${BASE_URL}/graphs/{graphId}/paths`

DESCRIPTION: This method retrieves all the possible paths from one node to another.
PARAMETERS: sourceNode -> the paths root, destNode -> the destination node
RESP. BODY: A Json/XML list containing all the paths between the two nodes

The full documentation of the tool can be found at `http://{TOMCAT_IP}:8080/Project-Neo4jManager/index.html` where `{TOMCAT_IP}` must be replaced with the actual hostname or IP address of the machine running Tomcat.

2.5.2 VeriGraph

The main operations supported by VeriGraph are summarized in the following:

1. METHOD: POST
   URL: `${BASE_URL}/graphs`
   DESCRIPTION: This method creates a new graph
   REQ. BODY: XML description of the graph
   RESP. BODY: In case of success, the service replies with an OK message and the ID of the newly created graph

2. METHOD: GET
   URL: `${BASE_URL}/graphs/{graphId}/nodes`
   DESCRIPTION: This method returns all the nodes of a single graph
   RESP. BODY: List of available graphs

3. METHOD: GET
   URL: `${BASE_URL}/graphs/{graphId}/nodes/{nodeId}/neighbours`
   DESCRIPTION: This method returns the neighbours of a given node
   RESP. BODY: List of neighbours

4. METHOD: GET
   URL: `${BASE_URL}/graphs/{graphId}/property`
   DESCRIPTION: This method checks whether the property specified by means of query parameters is satisfied or not
   PARAMETERS: type -> a string specifying the type of properties. It can be: REACHABILITY or ISOLATION
   sourceNode -> the source node
   destNode -> the destination node
   middleBox -> the node that must be traversed (used if type is ISOLATION)
   RESP. BODY: The verification result

The full documentation of the tool can be found at `http://{TOMCAT_IP}:8080/verify/api-docs` where `{TOMCAT_IP}` must be replaced with the actual hostname or IP address of the machine running Tomcat.

2.6 Known limitations and issues

The current implementation supports only reachability policies. However, other network properties such as flow isolation or the policies described in Section 4.1 of [D4.3] can be easily expressed in terms of single or multiple reachability properties thus allowing the verification module, with a trivial extension, to support them.

3 MEASURE - Automated monitoring intents and aggregation system

In this section, we discuss monitoring intents allowing automated deployment of monitoring functions (MFs), and the realization as well as automated configuration of a local aggregation function, implemented as a prototype called MEASURE -"Measurements, States and Reactions". The prototype shows how intents expressed in the MEASURE language (see D4.2) can be automatically translated into infrastructure deployment and configurations of a scalable monitoring system. The prototype is the combination of four separate components each with a very specific role. The four components are a Monitoring Management Plugin (MMP), acting as a monitoring controller in the context of a Universal Node; the MEASUREParser to parse MEASURE annotations; the Aggregation Point for monitoring data, realized with help of a Backend DB, and finally the Monitoring Functions Information Base MF-IB to list the available monitoring functions in a given infrastructure domain.

3.1 Overview

One of the main overall goals of UNIFY SP-DevOps is observability of heterogeneous and dynamic environments. Several monitoring and verification functions are available but they need to be instantiated, configured and the results exploited in terms of appropriate (re)actions. To do so in a generic manner, we introduced MEASURE for Measurements, States and Reactions. At the core lies the MEASURE annotation, a simple JSON formatted grammar to define the metrics to be monitored, information about the expected values and what to do it those values reach pre-defined thresholds (see the Figure 6 for an example). The be more specific, the Measurements part defines the metrics and a minimum set of parameters to configure them. This configuration will be handled by the MMP with the help of the MF-IB. The States are defined as zones - as long as the value of one metric or a logical combination of several metrics stays in the interval, the zone remains the same. Finally, a state-machine like mechanism reacts to the migration form one zone to another and contains information on the reaction to adopt, e.g. notify an orchestrator or control entity that resources are running short.

3.2 Motivation

The main goal of the monitoring system is to automate the instantiation and configuration of monitoring functions, as well as reactions to the monitoring results they generate. To do this the MEASURE NF-FG annotation language has been defined, which allows definitions of which metrics are to be monitored, and where to monitor them, in a technology agnostic fashion.

By specifying intented metrics to provide, rather that the tools that provides them, MEASURE allows the system to operate in heterogeneous environments, where the exact infrastructure domain for VNFs is decided dynamically by orchestration layers and can change during operations due to migration or scaling operations. MF instances are usually domain specific, which means that they for instance only operate in a Docker container based environment, but not in a virtual machine environment based on OpenStack. Upon receiving the request for a certain metric, the system can thus request a list of tools able to deliver the intented metric from the Monitoring Function (MF-IB). A tool suitable for the local environment can then be selected from the available tools.

Within the MEASURE description, measurements refer to abstract entities, such as nodes, ports, links or NF instances as specified in the corresponding NF-FG (NF-FG format see D3.3), rather than to what they are finally instantiated as (e.g. a VXLAN tunnel or a Docker container). This allows the system to remain agnostic of the actual components

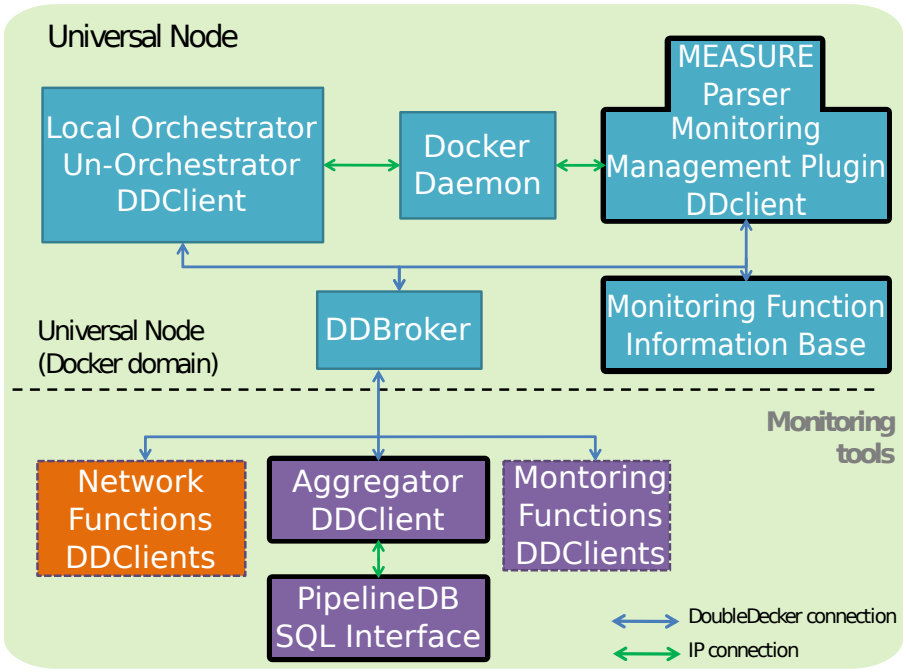

Figure 4: View of the MEASURE monitoring and aggregation system in the infrastructure layer (here a UN). Components with bold borders are described in this section.

it has to monitor until they have been instantiated. Once instantiated the orchestrator informs the system about the mapping from abstract entities to actual implementations, so that monitoring functions can be configured accordingly.

The main goal of the local aggregation system is to increase scalability of network management by reducing the overhead of monitoring processes. This is done in two ways: i) by utilizing the hierarchical DoubleDecker messaging system (described in section 6), and ii) by aggregating data as locally as possible. For this reason, a local Aggregation Point receives raw monitoring results from MFs. The aggregation logic can be configured dynamically by the MMP, based on the initial MEASURE description (specifically the Zones and Reaction parts). Acting as a sink for monitoring results, it continuously evaluates the results and upon traversing the defined zones sends or publishes events to interested components, typically local network functions utilizing the Cf-Or interface (e.g. VNF control apps), orchestrations or control layer modules, or network management systems. It is then up to these entities to take appropriate response, e.g. by requesting additional resources or handling failures.

An additional goal of the system is to be able to handle multi-tenant scenarios in a secure and confidential manner. This is provided by the DoubleDecker messaging system which allows customers or tenants groups to be defined and messages within these groups to be encrypted in an end-to-end fashion. This isolates tenants and prevents them to either intercept or inject control messages into another tenants services (i.e., NF-FG), or even in a non malicious case, avoid collision of names of different components managed by one tenant.

3.3 Features

A simplified view of the interactions between the different components in a complete setup is shown in Figure 5, with the components described here highlighted in blue. Most of the communication between the components go through the DoubleDecker broker, either as direct point-to-point Notifications via the pub/sub interface. On top of the Dou-

bleDecker layer JSON-RPC [Gro+12] is used to define the message format and the various commands used between the components.

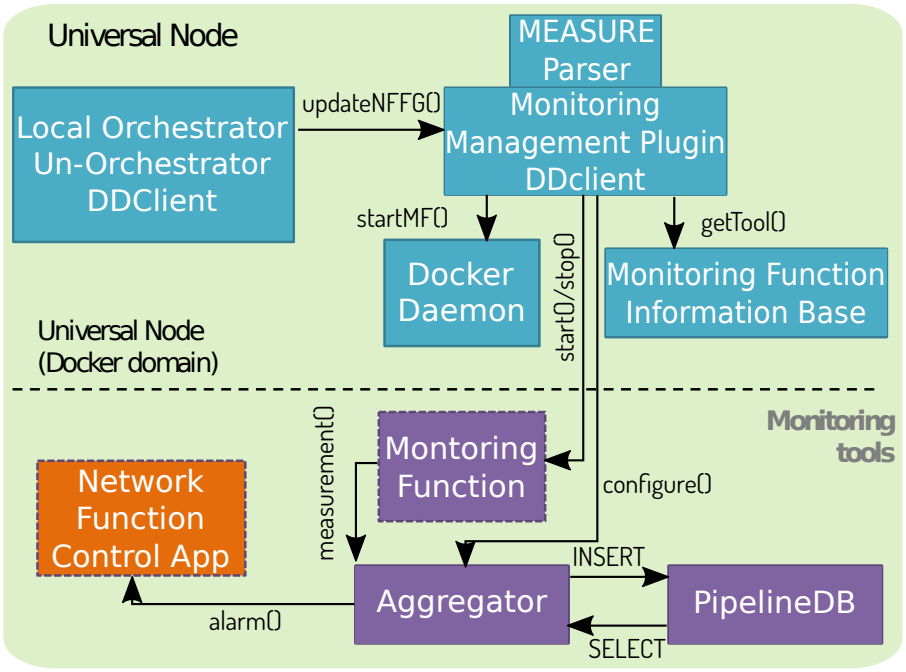

Figure 5: Logical view of interactions between components.

3.3.1 Monitoring Management Plugin (MMP)

The MMP extends the Python DoubleDecker client, and its main mode of communication is through a DoubleDecker broker. However, the MMP is also connected to the local deployment system, in this case a Docker daemon.

The MMP exposes one JSON-RPC command for receiving abstract to instantiated mapping information from the Orchestrator, together with the MEASURE descriptions, called updateNFFG. This is called whenever an NF-FG is deployed, modified, or removed, in order to either instantiate, update, or remove the associated monitoring. This triggers the MMP to parse the MEASURE description, using the MEASUREParser described in section 3.3.2, and:

1. Lookup of appropriate Monitoring Function to generate the requested metric. This is done by calling the getTool method of the MF-IB.
2. Generate configuration commands for the Aggregator and transmit it using the configure command of the Aggregator.
3. Startup of Monitoring Functions (if not already running) as Docker containers. This is done through the Docker-py interface to the docker daemon.
4. Configure Monitoring Functions by calling the start command of the started or running Monitoring Functions.

To generate the Aggregator configuration commands different backends could be plugged in. Currently only a single backend is supported, PAPBackend, which generates configuration for the PipelineDB-based Aggregation Point. 3

3.3.2 MEASUREParser

The MEASUREParser is a Python3 module implementing a parser for the MEASURE description. To use it, it has to be imported into the Python interpreter, where it provides 5 different methods:

- parse(string) - Returns a parse-tree in PyParsing named dictionary format.
- parseToDict(string) - Returns a parse-tree as Python Dictionaries.
- parseToXML(string) - Returns a parse-tree as XML.
- parseToJSON(string) - Returns a parse-tree as JSON.
- parseToYAML(string) - Returns a parse-tree as YAML.

These different outputs can be used to transform the MEASURE description into another format, e.g. for inclusion in a data model, or to generate the Aggregator configuration et.c. mentioned in section 3.3.1. If the MEASURE description cannot be parsed the methods raise an exception with information of where the parsing failed. The grammar in the parser is defined using PyParsing [McG07], which is relatively easy to extend in order to support additional features in the language.

3.3.3 Aggregator

The Aggregator extends the DoubleDecker Python implementation to communicate with other components, additionally it uses the psycopg2 Python module in order to communicate with PipelineDB [Nel+16]. PipelineDB is a so called Streaming SQL database, able to process streams of incoming data and discard the data once it has been used for a calculation. It is used to process incoming measurement results as well to calculate which zones are currently active.

The Aggregator receives configuration from the MMP using the configure call. This call configures the following functionalities :

1. Insert measurement results into the database, e.g. :

   ```
   CREATE STREAM stream_m1 (data JSON);
   ```

   Creates a stream for inserting JSON formatted data.

2. Configure the database, e.g. :

   ```
   CREATE CONTINUOUS VIEW view_z1 WITH (max_age = '5 minute') AS
       SELECT AVG(CAST(data->>'overload.risk.rx' as float)) AS
       "overload.risk.rx" FROM stream_m1;
   ```

   Creates a view, reading the overload.risk.rx JSON data field, as float, from stream_m1, and taking the average value over 5 minutes.

3. Configures how the data should be inserted in the database, e.g. :

   ```
   INSERT INTO stream_m1 (data) VALUES
       ('{results: {'overload.risk.rx':0.002 'overload.risk.tx':0.06}}')
   ```

   Insert the received JSON representation of a monitoring result.

4. How results should be retrieved, e.g. :

```
SELECT "overload.risk.rx" FROM view_z1 WHERE
    "overload.risk.rx" < 0.5
```

Selects the value from view_z1 and compares with the threshold, if a value is returned, zone z1 is active.

The Aggregator also supports a command, measurement, that is called by Monitoring Functions to process monitoring results. This command inserts the results in the database, and evaluates the results, potentially triggering associated actions. Data provided to the measurement command must follow the json syntax inspired by the mPlane format [TK16]. A result could be e.g. formatted as:

```
{
   "version" : 0,
   "results" : {
      "rate.rx" : 0.1234,
      "overload.risk.tx" : 0.2,
      "overload.risk.rx" : 0.002,
      "rate.tx" : 55.88
   },
   "label" : "ratemon",
   "parameters" : {
      "interface" : "veth0"
   }
}
```

3.3.4 Monitoring Function Repository / MF-IB

The MF-IB also extends the Python DoubleDecker client in order to provide a mechanism to resolve which tools can be used to provide a certain metric. These tools are registered in a local database, currently a json file , which can be queried using JSON-RPC calls. See Figure7 for the entry in the MF-IB corresponding to the Rate Monitoring. The call getTool(metric) returns a list of tools providing the requested metric, together with information on which configuration parameters the tool supports and a list of metrics it generates. In the prototype, one MF-IB is located locally per UN. The MF-IB could also be places on a more central localtion, e.g. within the global orchestration layer (similarly to the NF-IB in Escape).

3.4 Requirements and Installation

3.4.1 MEASUREParser

- The parser is implemented in Python3.
- We use pyparsing to parse the raw text and create the parsed tree, you can install this module using the command pip3 install pyparsing
- The parser uses the default json library available with python3
- A possible output for the MEASURE annotation is yaml, you will need to install the corresponding module with either pip3 install PyYAML or apt-get install python3-yaml
- The code is available at https://github.com/Acreo/MEASURE

- The parser must be installed as a python package using $ python3 setup.py install

3.4.2 MMP

- The parser is implemented in Python3.
- The MMP uses the DoubleDecker to communicate with other components, see 6.6 for more instructions on how to install the DoubleDecker.
- For the MEAUSUREParser module, see 3.3.2
- The MMP uses the default json library available in python3
- In the use-case of the elastic router the domain used is the Universal Node, this domain rely on Docker containers, the MMP needs access to the Docker-py library in order to spawn the monitoring containers.
- The format of the messages sent through the DoubleDecker follow the Jsonrpc style, we use two modules to handle the client/server sides, you can install the modules using pip3 install jsonrpcserver and pip3 install jsonrpcclient
- The MMP is used as a container itself, you can find the code to build this container at https://github.com/netgroup-polito/un-orchestrator/tree/elastic-router/use-cases/elastic_router/mmp.

3.4.3 Aggregator

- The current code of the aggregator is in Python3
- It uses the DoubleDecker to receive the raw monitoring data, see 6.6
- The format of the messages sent through the DoubleDecker follow the JSON-RPC style, we use two modules to handle the client/server sides, you can install the modules using pip3 install jsonrpcserver and pip3 install jsonrpcclient
- As the sink for data we use PipelineDB (for PAPBackend)
- The aggregator runs in a Docker container, you can find the code to build this container at https://github.com/netgroup-polito/un-orchestrator/tree/elastic-router/use-cases/elastic_router/aggregator

3.4.4 MF-IB / Registry

- The MF-IB uses JSON-RPC over DoubleDecker to provide information required about the available monitoring containers.
- Currently the containers available are described in a static json file, different alternatives are possible exploiting a Docker registry or a standard database but the interface would remain the same.

3.5 How-tos / Examples

Testing MEASURE is more complex than starting one prototype with the correct parameters. Tools are rather connected to their surrounding environment and it only makes sense if all other components are there. It is also more relevant if actual monitoring can be performed to demonstrate the usefulness of the Aggregator. However, to ease the development and debugging we implemented ways to test each component individually.

In the MMP, the function `testNFFG()` can generate and install and configure a dummy NF-FG. This can then be verified using direct calls to the Docker daemon to see the existence of new containers created.

The `test-mmp.py` script can generate random monitoring data. This is useful to test the aggregation but also the connection to the databases.

Finally all the components are connected to the DoubleDecker, it is always possible to use an interactive client such as `ddclient` or `ddclient.py` provided during the installation of the library. The advantage is to enable visual report of the messages passing through the messaging system. It is also possible to subscribe to the orchestration topics to check if the alarms are sent in a correct form.

```
measurement {
  m1 = oneway_latency(SAP1, SAP2);
  m2 = cpu_load(FW1);
} zones {
  z1  = Avg(m1, '5 minutes') > 10.0;
  z2 = Avg(m1, '5 minutes') < 10.0;
  z3 = Avg(m2, '1 minute') < 90%;
  z4 = Avg(m2, '1 minute') > 90%;
} reaction {
  z3->z4: Publish(topic=alarm, msg="Warning CPU");
  z2->z1: Publish(topic=alarm,msg="Warning latency");
}
```

Figure 6: Example of the MEASURE annotation

```
{
  "tools":{
    "ratemon":{
      "version":0,
      "label":"sics/ratemon",
      "parameters":{
        "interface":"string"
      },
      "results": {
        "rate.rx":"float",
        "rate.tx":"float",
        "overload.risk.rx":"float",
        "overload.risk.tx":"float"
      }
    }
  }
}
```

Figure 7: Entry corresponding to the RateMon in the MF-IB

4 Ramon – A distributed probabilistic rate monitoring function for node-local traffic rate modelling and congestion detection

4.1 Overview

The rate monitoring function implements a link utilization tool for assessing the risk of congestion. The method is based on estimating the parameters of a log-normal distribution modeling observed byte rates. The parameters are estimated locally at the node at longer regular intervals (seconds to minutes), given observations of byte counters and updates of the statistical moments at significantly shorter time intervals (milliseconds), which allows for detecting persistent micro-congestion episodes. Inspection of the percentiles of the cumulative density function corresponding to 99% of the link capacity allows for indicating the risk of congestion at a predefined threshold. The result is a more precise way of estimating the risk, compared to the common practice of using e.g. SNMP for assessing the 5-min average traffic rates, in which micro-congestion episodes cannot be easily detected. Moreover, the the benefit with a local estimate is the reduction in overhead and the possibility to represent link utilization in a compact form (i.e. the parameter estimates), compared to forwarding raw measurements for further processing, which is the case in the centralized monitoring solution offered by SNMP [KS15a]. Querying counter statistics locally at high rates allows for accurately capturing important aspects the traffic behaviour with significantly lower overhead than in a centralized setting. By varying the querying rate, we can achieve flexible high quality monitoring without the cost of constant high rate sampling of the counters [Ném+15].

4.2 Motivation

Existing approaches (e.g. SNMP and sFlow) normally involve forwarding of raw measurement information to dedicated monitoring equipment for further processing, which impacts the scale at which monitoring can be efficiently performed and thereby the overall network observability. For this reason, standard practice for identifying increased bandwidth consumption is based on low-frequency counter inspections and reporting when the average exceeds a fixed threshold. Using such low-resolution averages often leads to missed congestion episodes as well as false alarms, as these averages usually are far below the link capacity and determination of suitable detection thresholds is difficult [Joh+15]. Performing computationally lightweight node-local analytics on the traffic rates, does not only enable a richer model of the observed data, but also allows for a more compact representation (in terms of estimated parameters) that with significantly lower overhead can be disseminated to another network management function for further processing.

As described in [KS15b], the method-of-moments is used to estimate the parameters of a lognormally distributed link rates. The benefit of MoM is that it requires only storage of the first two statistical moments of the observed data, i.e. two counters for storing the sum and sum of squared observations. Computationally, only simple arithmetic is required for the updates, i.e. integer addition and multiplication.

4.3 Features

The implementation of the rate monitor will sample the traffic on the specified network interface and estimate the parameters of a log-normal distribution.The monitor can be run in two modes:

1. A standalone program recording the data from the monitor (i.e. traffic rates and estimation parameters) in a text file in JSON format.

2. Running in an OpenStack environment recording the monitor data in Ceilometer. (Se section "Storing the monitor data in Ceilometer" below).

The rate monitor has also been integrated with other tools of the Unify project in the Universal Node. This code is not included in the public release at the time of this writing.

4.4 Dependencies

4.4.1 Linux

The monitor has been developed and tested for Ubuntu 14.04, but it will probably work on other modern Linux versions and distributions.

4.4.2 Python version

The monitor runs in Python 2.7. You must to change the first line in the files run_monitor.py and monconf.py if the path to your default Python installation does not match the one specified.

4.4.3 Python pre-requisites

- Requests

  The Requests HTTP library is used for communication with Ceilometer.

  Installation instructions:

  http://docs.python-requests.org/en/latest/user/install

- Scipy

  Installation instructions:

  http://scipy.org/install.html

- sh

  Documentation: http://amoffat.github.io/sh/

  Install with pip:

  ```
  $ pip install sh
  ```

4.5 Known limitations and issues

- The timing for the sampling of the interface data counters is not very sophisticated. It is possible to sample about 1000 times per second, but the sampling rate will most likely vary somewhat.
- The rate monitor has not been tested for wireless interfaces.
- The rate monitor configuration port is hardcoded to 54736. This means that setting the –port option when running monconf.py will have no effect.

- Bad time synchronization between the rate monitor and Ceilometer may cause data loss. If the clocks on the computer running the rate monitor and the computer running Ceilometer differ too much, data may not be recorded in Ceilometer, and no error will be propagated to the rate monitor.

4.6 Installation

The monitor code can be downloaded from the NIGSICS Github page. The releases are at

https://github.com/nigsics/ramon/releases

The necessary files to run the rate monitor are all the Python files included in the installation package. Simply unpack the downloaded zip or tar file and start the monitor according to the instructions below.

4.7 Data stored by the rate monitor

The data is stored in a JSON object with the following names.

'timestamp'	Date and time when this data was recorded.
'interface'	The network interface being monitored.
'linerate'	The line rate in bytes per second of the interface being monitored.
'alarm_trigger_value'	The trigger value for the overload risk over which to set the alarm flag to True.
'cutoff'	The percentage for the link speed to use in the overload risk estimation.
'tx'	The mean transmission rate during the last estimation period.
'var_tx'	The variance of the transmission rate during last estimation period.
'mu_tx'	The location parameter of the estimated log-normal distribution for outgoing traffic.
'sigma2_tx'	The scale parameter of the estimated log-normal distribution for outgoing traffic.
'overload_risk_tx'	The overload risk in percent: (1 - cdf) * 100, (cdf = cumulative density function) for outgoing traffic.
'alarm_tx'	True or False.
'rx'	The mean reception rate during the last estimation period.
'var_rx'	The variance of the reception rate during the last estimation period.
'mu_rx'	The location parameter of the estimated log-normal distribution for incoming traffic.
'sigma2_rx'	The scale parameter of the estimated log-normal distribution for incoming traffic.
'overload_risk_rx'	The overload risk in percent: (1 - cdf) * 100 (cdf = cumulative density function) for incoming traffic.
'alarm_rx'	True or False.
'sample_rate'	The sample rate of the monitor.
'estimation_interval'	The estimation interval of the monitor.
'meter_interval'	The metering interval of the monitor.

When the monitor is used in mode 1 it will store a sequence of such JSON objects in a file the user specified at startup and/or send the JSON objects data to a local TCP port.

See the next section for how data is stored when the monitor is used in mode 2.

4.8 Storing the monitor data in Ceilometer

A minimal OpenStack installation of one controller node, running Ceilometer and its necessary support services, is sufficient for testing. The monitor can run on any computer that can connect to the Ceilometer node; on a physical computer or on a VM.

We use the web API of Ceilometer to store metering data in Ceilometer. This is documented at http://docs.openstack.org/developer/ceilometer/webapi/v2.html#user-defined-data. More hints can be found at http://docs.openstack.org/admin-guide-cloud/content/section_telemetry-post-api.html and here: https://www.mirantis.com/blog/openstack-metering-using-ceilometer

Storing the data in Ceilometer is done by a HTTP POST, with a JSON object as the data provided. See the code in the file ceilocomm.py, specifically the method putMeter(), for details.

4.8.1 Required fields

The required fields of the JSON data to store a meter are (values in this example are arbitrary):

"counter_name"	"test"
"user_id"	"admin"
"resource_id"	"76799085-e0ff-4620-9b7f-120d3c51cc49"
"counter_unit"	"%"
"counter_volume"	10.57762938230384
"project_id"	"855f014353ec48d98ef7b887fc6980e1"
"counter_type"	"gauge"

Contrary to the documentation mentioned above, project_id is a required field.

4.8.2 Data stored by the meter

All data from the rate monitor is stored in the resource_metadata field of the JSON data. See section 4.7 for details of what data will be stored.

Example of meter data stored in Ceilometer

```
{
    "counter_name": "sdn_at_edge",
    "user_id": "855f014353ec48d98ef7b887fc6980e1",
    "resource_id": "76799085-e0ff-4620-9b7f-120d3c51cc49",
    "timestamp": "2015-05-07T14:17:47.665000",
    "counter_volume": 4711.0,
    "resource_metadata": <a JSON object as described above>,
    "source": "76799085-e0ff-4620-9b7f-120d3c51cc49:openstack",
    "counter_unit": "b/s",
    "recorded_at": "2015-05-07T14:17:47.678000",
    "project_id": "76799085-e0ff-4620-9b7f-120d3c51cc49",
    "message_id": "d62753ec-f4c3-11e4-9642-080027486ab1",
    "counter_type": "gauge"
}
```

timestamp, source, recorded_at, and message_id are added by the Ceilometer API, and should not be specified by the user of the API.

4.9 Configuring the rate monitor

Configuring the rate monitor is done at start-up. Do 'run_monitor -h' for a list of configuration parameters.

4.9.1 Start parameters

-b	–meter_port	[1]	Port to send metering data to. Setting this option will start the monitorn in mode 1. Omitting both the -b option and the -f option will start the monitor in mode 2.
-f	–meter_file	[1]	Name of a file to append metering data to. Setting this option will start the monitorn in mode 1. Omitting both the -f option and the -b option will start the monitor in mode 2.
-i	–interface		Interface to monitor; default "eth0".
-s	–sample_rate		How often to sample; default 1000 samples per second.
-e	–estimation_interval		How often to estimate; default every 10 seconds.
-m	–meter_interval		How often to meter; default every 30 seconds.
-k	–link_speed		Set the link speed value for the monitored interface (in Mbits per second).
-a	–alarm_trigger		The overload risk which will trigger an alarm; default 95%.
-o	–cutoff		A percentage of the link speed to use in the overload risk calculation; default 99%.
-n	–meter_name	[2]	The name of this meter in Ceilometer.
-r	–resource_ID	[2]	Resource identifier for the resource to associate with the meter in Ceilometer.
-p	–project_ID	[2]	Project identifier for the resource to associate with the meter in Ceilometer.
-c	–controller	[2]	The IP adress of the controller running Ceilometer; default 10.0.0.11.
-u	–username	[2]	OpenStack User name; default "admin".
-w	–password	[2]	Password.
-t	–tenantname	[2]	OpenStack Tenant name; default "admin".
-d	–debug		Debug flag
-l	–log		Log debug messages to a file of the form monitor_meter_name_%Y-%m-%dT%H.%M.%S.log.
-v	–version		Show version and exit.

[1] Mode 1 only.

[2] Mode 2 only.

4.9.2 Configuration at run-time

The rate monitor can be configured at run-time with 'monconf.py'. Do 'monconf.py -h' for a list of configuration parameters.

These parameters can currently be configured:

–pause		Pause the rate monitor.
–resume		Resume the rate monitor.
–status		Show the status of the rate monitor.
–exit		Tell the rate monitor to exit.
		Setting this option will cause all other options to be ignored.
-i	–interface	Interface to monitor.
-s	–sample_rate	Sample rate in samples per second.
-e	–estimation_interval	Estimation interval in seconds.
-m	–meter_interval	Meter interval in seconds.
-k	–link_speed	Set the link speed value for the monitored interface (in Mbits per second).
-a	–alarm_trigger	The overload risk which will trigger an alarm (a percentage).
-o	–cutoff	A percentage of the link speed to use in the overload risk calculation; default 99%.

Other options for monconf.py:

–host	IP or name of the computer running the rate monitor. Default is 127.0.0.1.
–port	Port of the computer running the rate monitor. Default is 54736.

The default port number is the port number currently hardwired into 'monitor.py' so no other portnumber will work.

4.10 Starting the rate monitor

Simply start run_monitor.py from a shell with the appropriate configuration parameters. With the debug flag the monitor will only display the metering data on in the controlling terminal and will not store data persistently.

4.11 Stopping the rate monitor

Use the –exit option of monconf.py or type ^C in the terminal running the rate monitor.

5 AutoTPG – Verification of Flow-Matching Functionality in OpenFlow

5.1 Overview

Our tool performs verification of flow-matching functionality in OpenFlow switches/routers. A Flow-Match Header fault occurs when matching of a packet with the Flow-Match Header gives an incorrect result (i.e., a packet gets matched with the Flow-Match Header which it should not, or a packet does not get a match although a matching Flow-Entry is present in the FlowTable). These faults can be present due to: (1) bugs in OpenFlow switch implementation and (2) errors in FlowTable configuration. Bugs in OpenFlow switch implementation may be caused by bugs in the hardware or software part of switches. The errors in FlowTable configuration may be caused by: (1) bugs in controller software for the addition of a Flow Entry and/or (2) presence of a high priority error-prone Flow Entry (added manually or by a controller) that gives a match with the incoming packets. The objective of verification is to find this incorrect or no matching and hence, to find the packet-headers that cannot be matched or can be matched incorrectly with the Flow-Match Header of a Flow Entry. In the absence of this verification, it may be difficult to find which packets cannot be delivered or can be delivered incorrectly by a switch. The tool verifies the flow-matching functionality by transmitting test packets.

5.2 Requirements and Installations

5.2.1 Requirements

AutoTPG is implemented on the top of the floodlight controller (OpenFlow 1.3 package), which uses java to implement the controller. Therefore, like floodlight, AutoTPG can be installed on the operating system such as Ubuntu and Mac.

For the Ubuntu operating system, build essential, jdk, ant, and python-dev are needed to be installed. The linux command for installing these utilities are:

```
sudo apt-get install build-essential default-jdk ant python-dev
```

For Mac, Xcode and java development kit are needed to be installed. Xcode contains all necessary build essentials for installing the autoTPG controller and the java development kit installs automatically the jdk if it is not already present.

Currently, our software is tested in the Ubuntu operation sytem and can be run to test OpenFlow switches which run OpenFlow version 1.3.

5.2.2 Installation

There are following four steps to install autoTPG on the linux operating systems:

1. Download the floodlight-plus (which supports OpenFlow 1.3 version) through the following command:

   ```
   git clone https://bitbucket.org/sdnhub/floodlight-plus.git
   ```

2. Download the autoTPG patch file and apply the patch file in the floodlight folder by:

   (a) `cd floodlight-plus`

   (b) `wget http://users.intec.ugent.be/unify/autoTPG/autoTPGpatch`

   (c) `patch -s -p2 < autoTPGpatch`

3. Download the autoTPG code folder and unzip it:

   (a) `wget http://users.intec.ugent.be/unify/autoTPG/autoTPG.zip`

   (b) `unzip autoTPG.zip`

4. Install the autoTPG controller by: `sudo make (or sudo ant)`

5. Run the autoTPG controller by: `sudo ./floodlight.sh`

5.3 Internals

Our tool implements three steps of verification: (1) flow duplication, (2) test packet generation, and (3) matching error calculations. In the flow-duplication step, the tool copies all the Flow Entries from a table to another table. In the test packet-generation step, the tool transmits test packets. In the matching error-calculations, the tool calculates errors by either the binary search method or by the packet-reception method (See D4.3). We now discuss the implementation of all these steps in detail.

5.3.1 Flow-Duplication Step

AutoTPG uses a header field (such as EtherType or VLAN ID) for differentiating test packets from data packets and assumes that this header field (e.g., EtherType) is wildcarded in the Flow-Match Header part of Flow Entries. In the flow duplication step, for verifying FlowTable x (Figure 8), the controller copies all the flows from FlowTable x to FlowTable y (x and y are ≥ 0 and $y > x$). The Flow-Match Header in Figure 8 is an IP address field, but it can be any field (one or more) of the Flow-Match Header of Flow Entries. Figure 8 shows that the Flow-Match Header of all the duplicated flows in FlowTable y is same as the Flow-Match Header part of the original flows in FlowTable x. However, the difference

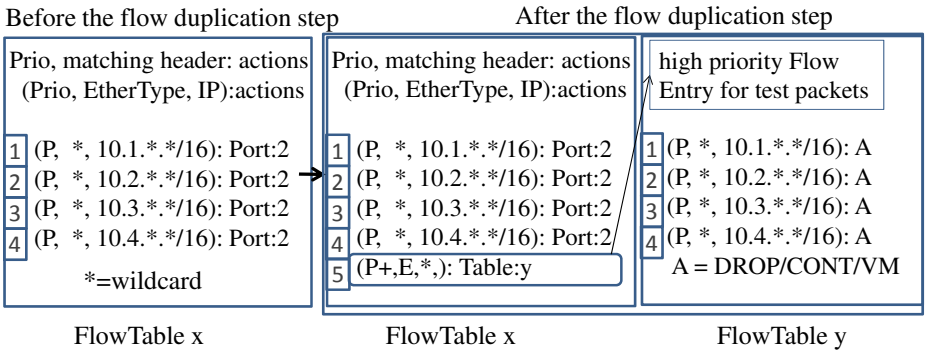

Figure 8: Flow Entries before and after the flow duplication step. E is the EtherType of the test packets, P+ is the priority number higher than P. DROP, CONT or VM are actions for the duplicated FlowTable

lies only in the action part. The action of all these duplicated flows is DROP, CONT, or VM. "DROP" means drop all the packets that match with the Flow Entry. "CONT" or "VM" means send all the matched packets to the controller or to the VM respectively. Moreover, the controller inserts an additional Flow Entry (5^{th} entry in FlowTable x in Figure 8) in FlowTable x to forward all the test packets to FlowTable y (action=Table:y). To match all test packets with this entry, the

entry contains a higher priority (P+) number than the priority (P) number of the existing Flow entries in FlowTable x and the Flow-Match Header contains the Ethertype (E) of test packets and all other fields as wildcarded fields.

5.3.2 Test Packet Generation Step

In this step, the controller or VM (running the autoTPG software) transmits test packets to the switch whose entries are needed to be verified. For this, the controller or VM can transmit all the test packets that can give a match with the Flow-Match Header of a Flow Entry. Using our software, test packets are generated in the form of packet-out messages.

5.3.3 Matching Error Identification

For this step, we implement two methods: (1) binary search and (2) packet-reception. The binary search method applies the well known binary search algorithm to find the matching errors. The packet-reception method receives the sent test packets and from the unreceived/received test packets, the method finds the matching errors. The advantage of the binary search method is that it reduces the upstream bandwidth to receive the test packets. However, the disadvantage is that it takes more time to find matching errors.

5.4 How-to and Examples

In these examples, we will install a wildcarded flow in a switch and will verify the flow for the flow-matching issue that may be present in the switch. Using our software, a Flow Entry or a complete FlowTable (or all FlowTables) can be verified for flow matching issues. We show the working our tool using following examples:

1. Install a wildcarded Flow Entry in a FlowTable (an example of a wildcarded Flow Entry is given below)

```
curl -d '{"switch":"00:00:00:00:00:00:00:01", "tableid":  "0", "priority":"1",
"inport":"1", "etherType":"0x800", "ipDst":"11.0.0.2/24", "output":"1"}'
http://controller-IP:8080/wm/autoTPG/qpc/json
```

   (a) Here, the switch datapath id is 00:00:00:00:00:00:00:01 ("switch":"00:00:00:00:00:00:00:01").
   (b) The ID of table is 0 ("tableid": "0").
   (c) The priority of the entry is 1 ("priority":"1").
   (d) The incoming port is 1 ("inport":"1").
   (e) The EtherType is 0x800 ("etherType":"0x800").
   (f) The network address to match with packets is 11.0.0.2/24 ("ipDst":"11.0.0.2/24").
   (g) The output port is 1 ("output":"1").

2. Verify a FlowTable

   We can run following commands to verify either all FlowTables or a single FlowTable:

   (a) An example of the command to verify a FlowTable with the packet-reception or binary search method:

```
curl -d '{"switch":"00:00:00:00:00:00:00:01", "tableid":  "0", "method":"rec"}'
http://controller-ip:8080/wm/autoTPG/qpvf/json
```

i. Here, the switch datapath id is 00:00:00:00:00:00:00:01 ("switch":"00:00:00:00:00:00:00:01").

ii. The ID of table is 0 ("tableid": "0").

iii. The method is the packet-recetion method ("method":"rec") or the binary search method if "method:"bin".

(b) An example of the command to verify all FlowTables:

curl -d '{"switch":"00:00:00:00:00:00:00:01", "method":"rec"}' http://controller-ip:8080/wm/autoTPG/qpvf/json

Here, "method":"rec" is running the packet-reception method and "method":"bin" is for running the binary-search method.

3. Results

The results will be displayed in the same terminal as an xml format. An example of the results is shown below:

"errors":2,"exitcode":"SUCCESS"",culprit":["11.0.0.2", "11.0.0.3"],"logfile":"/tmp/error.txt"

(a) Here, the number of errors is 2 ("errors":2).

(b) The success or failure of the command is given by "exitcode":"SUCCESS" or "exitcode":"FAILURE".

(c) The matching error is given by a packet header having the destination IP address as "11.0.0.2" or "11.0.0.3" ("culprit":["11.0.1.2", "11.0.0.3"]).

(d) Additional information about errors can be found at /tmp/error.txt ("logfile":"/tmp/error.txt").

6 DoubleDecker – distributed messaging system

6.1 Overview

DoubleDecker is an hierarchical distributed messaging system based on ZeroMQ which can be used to provide messaging between processes running on a single machine and/or between processes running on distant machines. It is hierarchical in the sense that message brokers are connected to each-other in a tree like topology (see 9). A client can send a message to any other client knowing only its identifier within the system.Messages are routed through the architecture based on routing tables stored in brokers.

The DoubleDecker is built as a library so it can be integrated with a wide range of tools. We provide the library in several languages to support the majority of the tools developed within Unify. The DoubleDecker can be used for various purposes such as publishing monitoring results from a VNF, sending alarms to the orchestration components or remotely configuring VNFs without allocating a specific IP address.

From an integration point of view the DoubleDecker is used as a glue between all the other components. It is used to transport raw monitoring data to aggregation points and databases, send monitoring alarms to the orchestrator, send runtime configuration commands to various components. It has been demonstrated at EWSDN15 [PMM15]

6.2 Motivation

As network environments become richer, the number of tools involved explodes. In a features rich environment like the Unify architecture the diversity becomes a challenge. Each tool has specific needs but still has to communicate with others. The Unify architecture allows and encourage the addition of third party tools, including monitoring tools. It is then necessary to provide a unified way to connect all the components without limiting the possibilities.

A very interesting feature offered by ZeroMQ is the possibility to use inter-process communication (IPC) sockets instead of TCP which solve IP addressing questions. A combination of both TCP and IPC is possible. It makes sense if IPC is used between all the components on the same node and the communication with components on higher layers is done via TCP.

The DoubleDecker also solves dynamicity related questions. Clients can dynamically join and leave the messaging

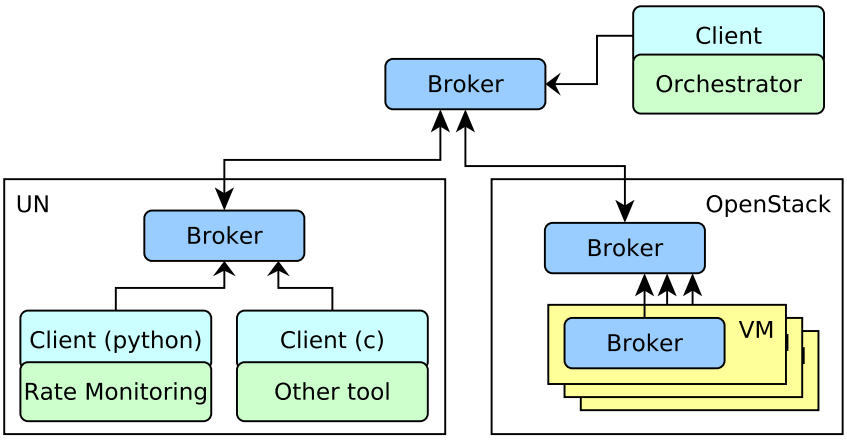

Figure 9: Example of an architecture using the DoubleDecker

system, the routes are kept up to date on real time. As clients are identified by an id they can be migrated with no consequence for the rest of the architecture.

Another constraint is the increased development speed of SDN related tools, as a positive result of the softwarization. Service providers can develop, deploy and update network functions at a much higher pace than in a legacy network. It is then important to provide an easy to use solution for the developers of VNFs to integrate their tool in the architecture. The core of the DoubleDecker is the open communication protocol between clients and brokers more than the code itself. It makes our solution viable at long term as it can be adapted to any new language.

6.3 Features

The goal of the DoubleDecker was to offer a resilient and efficient messaging system. Most of the goals are met by design, but some important features for developers can be identified.

6.3.1 Point to point messaging

The main feature of the DoubleDecker is the point to point messaging. The messaging system gives the abstract representation of the actual topology as if all the other clients were one hop away. Using a single socket, a client is able to send messages to all the other clients connected. Every client provides its name at registration and can be contacted knowing only this identifier.

6.3.2 Publish/Subscribe mechanism

In addition to the normal point to point messaging, the DoubleDecker offers a publish/subscribe mechanism. Clients can subscribe to any topic, identified by a string, no pre-configuration is needed. The subscription does not affect the other clients directly. The reciprocal functionality, publish, is also available for any client. They can decide to send messages not to another client identified by it's name but to a topic. Any client which previously subscribed to this topic will receive the message. This mechanism has important consequences on how the messaging system can be used. It gives an additional abstraction to the identifier. The name of the client is not necessary. It becomes very useful when a component has to contact many others but doesn't keep track of which client is connected. For instance the orchestrator might be interested in sending a command to all the monitoring tools. This can easily be achieved if all the monitoring components subscribe to a given topic like "monitoring-tool". This can be used as a service discovery mechanism, upon registration a client can publish on a topic, "discovery" for example, and provide informations about its role and capabilities. The other components now have access to its identifier and can send point to point messages without any static pre-configuration.

6.3.3 No message size limit

When using classic UNIX sockets several implementation choices constrain the possibilities. The DoubleDecker takes advantage of the abstraction offered by ZeroMQ. In particular when it comes to messages size. The size of the content of the message is completely abstracted by the library on both the sender side and the receiver side. The limit of the messaging system is very close to the limit of the link or links separating the two clients. For monitoring results, configuration or administration this limit will most likely never be reached.

6.3.4 Heartbeating mechanism

The DoubleDecker implements a simple heartbeating mechanism. The heartbeat happens at two different levels, first between brokers and then between a client and the broker it is attached to. This mechanism allows to know in real time if a link is broken or if a component failed. On the client side a specific callback is used in case of disconnection, this allows a local save of the data until the broker is up again for instance. The clients will also automatically try to reconnect at a regular interval so a developer using the library has almost nothing to care about.

6.3.5 Authentication

The DoubleDecker features security mechanisms, authentication at registration and point to point encryption of messages content. The tenant of the client being attached to the DoubleDecker messaging system provides a public/private keys pair to the client. Upon registration the client is challenged by the broker to validate its identity. To do so the system administrator has to provide the public key to the broker. This is a reasonable overhead since the key belongs to a tenant, an unlimited number of DoubleDecker clients can be started per tenant. As clients are authenticated per tenant, the same brokers can be used by different tenants. This authentication is completely transparent to the clients, everything happens as if there was only one tenant. Internally identifier and pub/sub topics are specific to a tenant so there is no risk of collision.

6.3.6 End to end encryption

Clients sue the public/private keys pair to encrypt messages from end to end. This was an important requirement as sensitive data such as orchestration commands can transit through the DoubleDecker. The brokers forward messages without touching the encrypted content. Again, everything is transparent for a VNF developer using the library.

6.4 Dependencies

The DoubleDecker is an implementation relying on the ZeroMQ library. The library needs to be available on the system to build and run the DoubleDecker. As the library has been ported over several architectures and several languages the exact version needed depends on the specific case. The Brokers depend on the C version of ZeroMQ. The client side has been ported so far into C, python3 and Java, depending on the version you use the corresponding library will be needed. All the other libraries needed are open sources and available in various languages and for the most common platforms. Security part of the DoubleDecker uses the ECC algorithms, this security mechanism is standard. We currently use libsodium in C, PyNaCl in python and kalium in Java. These libraries can be exchanged for a different implementation.

6.5 Known limitations and issues

The main limitation of the DoubleDecker as a library is the limited number of languages supported for now. This is not a hard limitation as a port is always possible and there is alternative solutions to connect an existing code with the client library. This would however slow down the integration process.

Albeit testings in various setups we couldn't reach a level of complexity where the DoubleDecker was failing. In a very diverse environment with Docker containers, virtual machines, unstable links combined with thousands of clients stability problems are likely to appear. We have simply not been able to test our tool enough to establish a clear limitation.

A known issue comes from the unlimited messages size. If by mistake, a message of several gigabytes is pushed through the system, the library will do it's best to transfer everything but could starve other components. The link would be saturated first but also the cpu resources as encrypting the message would require very heavy computation.

The abstraction offered by the publish/subscribe mechanism can also make the debugging difficult as there is no guaranty that any client is listening to the topic. Messages sent on a topic with no subscriber will be silently discarded by the brokers. A reception acknowledgement system can however be implement of top of the DoubleDecker.

6.6 Installation

The code of the DoubleDecker can be found at https://github.com/Acreo/DoubleDecker

Depending on the language you need the installation process will be different.

6.6.1 Building the C version broker/client

The broker is to be used as is so you simply need to follow the building instructions and use the generated binary.

If you have your own VNF and want to make it a client then you have to include the correct headers and change your building script to include dependencies to the DoubleDecker code. You can find examples of implementations on the repository. You then have to implement 5 callbacks and the other functionalities will be available for your prototype.

6.6.2 Building the python module for clients

In the python directory we provide all the scripts necessary to build and install the DoubleDecker as a python module. You will need to have all the dependencies installed, either from system packages or using pip. Then it can be imported just like any python module. The installation will also provide two scripts directly usable. The first one is an interactive implementation of the client side of the DoubleDecker. You can start it using the command ddclient.py. The second script provided helps for the creation of the encryption keyrs, see 6.9

6.6.3 Building the Java library

The Java implementation in only available for clients. We have implemented an interactive client that you can use to as a test client. In order to build the test client, you need to have Maven installed on your machine and run the following commands :

`# mvn compile` This command will build the client

`# mvn exec:java` This command will execute the main program

On the other hand, if a developer is interested in integrating the DoubleDecker with its own source code the package needs to be imported using import se.acreo.doubledecker and add following dependences on class path:

$rg.zeromq$ For the underlying transport library ZeroMQ

$org.abstractj.kalium$ package used for encryption

6.7 Starting a DoubleDecker broker

The broker accepts several startup parameters :

$-r < routeraddr >$	e.g. tcp://127.0.0.1:5555
	Multiple addresses with comma tcp://127.0.0.1:5555,ipc:///file.
	This is where clients will connect
$-k < keyfile >$	JSON file containing the broker keys
$-s < scope >$	scope "1/2/3"
Optional	
$-d < dealeraddr >$	e.g tcp://1.2.3.4:5555.
	Dealer should be connected to Router of another broker.
	If no dealer is provided the broker will be the root of the system
$-l < loglevel >$	e:ERROR,w:WARNING,n:NOTICE,i:INFO,d:DEBUG,q:QUIET
$-m < name >$	open a monitoring socket at /tmp/<name>.
$-v$	Verbose
$-h$	Help

Depending on the logging level the broker will print more or less information. It can be very useful do start a broker in interactive mode to troubleshoot an architecture or debug a client. The broker will print a messages for instance: when a new client register, when a client is disconnected and when clients subscribe/unsubscribe to publishing topics.

6.8 Starting a DoubleDecker client

The example implementation in python accepts the following parameters, they are all required. They represent the arguments needed to create an instance of the client. In a different implementation these parameters could be stored in a config file, provided manually at runtime or any other mean depending on the specific need.

$-c < customer >$	Name of the customer owning the client
$-k < keyfile >$	File containing the pair of pub/priv keys
$-n < name >$	Name of the client to be used in the messaging system
$-d < tcp/ipcurl >$	Address of the broker

6.9 Generating key pairs

The security features are now mandatory. A client will need a keys pair in order to register to a broker before doing anything. A script is available in python to generate the keys and format the a JSON file correctly formatted. The script will generate the keys pair for the client and a file containing only the public key which will be used by the broker to authenticate clients. To generate the keys you can simply execute the command ddkeys.py and follow the instructions.

6.10 Docker version of the Broker

A container with the broker already built and configured is publicly available at https://hub.docker.com/r/acreo/ddbroker/. It is the easiest and fasted way to set up a testing environment. The container can either expose TCP ports or share a volume where the IPC socket file can be created and made accessible for clients. We try to keep to container updated and re-build it with the latest version of the DoubleDecker but the Dockerfile is also available if you want to build your own broker container.

7 Epoxide – Multicomponent troubleshooting framework

7.1 Overview

Epoxide is an Emacs based modular framework, which can flexibly combine network and software troubleshooting tools in a single platform. Epoxide does not try to be a complex troubleshooting software that fully integrates all available tools, but rather a lightweight framework that allows the ad-hoc creation of tailor-made testing methods from predefined building blocks to test troubleshooting hypotheses.

Epoxide is a tool built around the concept of connecting existing network and software troubleshooting tools in a way that they can transfer data among each other. To describe such scenarios we introduce troubleshooting graphs.

Troubleshooting graphs (or TSGs in short) consist of nodes and links. Nodes are functional elements either implementing a wrapper for an external tool or providing entirely new functionality. Links connect nodes by relaying data between them.

For defining TSGs we use a Click-inspired language [Koh+00] and store them in .tsg configuration files. The nodes are implemented in Emacs Lisp and we chose human readable text data to be transfered between them (where possible).

The creation, execution and manipulation of TSGs are controlled by a framework that maps nodes and links to Emacs buffers and then helps the data distribution among nodes. Node buffers hold node specific attributes and potentially display state information; while link buffers are the inputs and the outputs of the nodes containing human readable text data (e.g. the output buffers of the Iperf wrapper node contain the intact output of the wrapped tool). The framework with the help of its event scheduler is responsible to call the execution function of a node if one of its input buffers got modified, afterwards it is the responsibility of the execution function to process the input data, and to write into its output buffers.

Epoxide offers [Lév+15]:

- Easy extensibility: framework and node functions are implemented in separate Emacs Lisp files and the framework provides a programming interface to node developers. This allows creating third party node implementations and also external node repositories. Developers only have to write node initialization, execution, and termination functions. Optionally, they can add documentation functions to provide information about node configuration parameters.
- Context-aware buffer switching keeps track of the created buffers and provides support to move among them in an orderly fashion by applying key combinations. When multiple branches of the TSG are available, possible choices are offered as a selectable list, or they can be selected by the same key combination prefixed with the output link's number. Users can also switch to buffers associated with the current Epoxide session by selecting from a list of buffer names grouped together based on their types and positions in the TSG.
- TSG visualization offers a graph representation of the interpreted .tsg file, navigation options for graph traversal, and quick access to node and link buffers.
- Syntax highlighting and context-aware code completion is provided for .tsg files. Additional, context-aware help is also shown with the short documentation of the current node, in which the currently typed parameter is highlighted. Intelligent code completion is offered: candidate lists are populated with node classes and phrases from other buffers.

- Views are special TSG elements that can create bundles of link and node buffers to be displayed together by splitting the Emacs window to different sub-windows. In a complex TSG, views provide quick access to a subset of the buffers. Moreover, in case of a long chain of nodes not only the final result is shown but every intermediate partial result is also available for inspection in the corresponding link buffer.
- Run-time node creation, reconfiguration and re-linking provides two possibilities to make modifications on a TSG that has already been started. The first one allows reinterpreting a modified .tsg file and then reconfigures the TSG based on the modifications. The second one interactively guides the user to add a new node or reconfigure existing nodes or links between them.
- Customization of major Epoxide parameters is integrated in Emacs's own customization interface and can be invoked via M-x customize-group and then typing the epoxide keyword.

7.2 Requirements and Installation

Epoxide can be installed via Emacs's package manager or by downloading the source from the project's github page. Epoxide can be used with Emacs version 24.4+.

7.2.1 Via the Emacs package manager

Add the following lines to your emacs initialization file:

```
(package-initialize)
(add-to-list 'package-archives
'("epoxide" . "http://nemethf.github.io/epoxide/") t)
```

Then install the epoxide package with M-x list-packages RET.
To utilize graph visualization install Graphviz from your distribution's package manager or from any other source.

7.2.2 From github

First install COGRE either from the CEDET git repository or from the package archive of Epoxide (see above), then clone the project from github by issuing:

```
git clone https://github.com/nemethf/epoxide.git
```

After this add the following lines to your initialization file:

```
(add-to-list 'load-path "path-to-the-epoxide-repository/src")
(autoload 'epoxide-tsg-mode "epoxide")
(autoload 'epoxide-topology-mode "epoxide")
(add-to-list 'auto-mode-alist '("\\.tsg\\'" . epoxide-tsg-mode))
(add-to-list 'auto-mode-alist '("\\.topo\\'" . epoxide-topology-mode))
(autoload 'tramp-mininet-setup "tramp-mininet")
(eval-after-load 'tramp '(tramp-mininet-setup))
```

7.3 Internals

Building on the buffer concept, Epoxide maps the nodes and links of a troubleshooting graph to Emacs buffers [Pel+15]. There are several advantages of this approach. First of all, it provides a uniform interface for nodes. Secondly, link

buffers have similar functionality and convenience as Unix pipes, but with additional features: while output buffer contents are visible to troubleshooters, pipes are hidden from their sight. Lastly, users can take advantage of the core features of Emacs. For example, they can easily navigate among links and nodes; by selecting a buffer they can observe or modify the state of a node or data of a link; or they can redirect the output of shell commands into a link buffer using the built-in shell interpreter called eshell. Additionally, node developers can also rely on available program libraries.

The functionality provided by Epoxide can be split into two categories:

- framework functionalities and
- node functionalities

The first provides the interpretation of the .tsg files and the life cycle management of the nodes.

When opening a .tsg file the Epoxide framework parses the TSG described by it, sets up buffers for the nodes and their outputs, makes the necessary connections between the nodes and executes the code contained in the nodes.

An event scheduler resides in the heart of the framework implemented as a combination of a simple event-queue containing node buffer names and a timer-function processing the head of the queue when Emacs is idle. Nodes are inserted into this queue when a related event occurs – typically when new data arrives on a node's input link. When Epoxide processes an event it calls the mandatory execution function of the corresponding node (more on node functions later). Then the framework collects the modified link buffers and schedule calling the execution functions of the associated nodes, hereby implementing a coordinated communication between the nodes.

The second group of functionalities provided by Epoxide helps in the process of integrating preexisting troubleshooting tools.

Every node has to implement an initialization, an execution and a stop function. The first one takes care of setting up node attributes that needs to be set before first running the tool wrapped by the node. The second function is responsible for calling the wrapped tool or processing the node's inputs, making transformations on the results (when necessary) and relaying that to the node's outputs by optionally appending text to the output buffers. The last function provides options to release resources when the node is cleared up.

7.4 How-to and examples

For an illustration of how Epoxide can be used to test networks and how to extend it if the available nodes are not enough for the task at hand a simple example is presented that takes you through the process of creating a troubleshooting graph, executing it and making observations during its run.

In the following section the following sample scenario is going to be used. Let's assume we have a host and want to test its connectivity to the public Internet. First we want to examine if connection can be established then we would like to measure the latency to some well-known websites. To accomplish this, the simplest tool we can use is ping: it tells us whether the target website is available, and if so, the latency is displayed also.

Let's further assume that we would like to mark those measurements where the round trip time exceeds a certain limit. To achieve this we would need a tool that can process the output of ping.

7.4.1 Constructing the TSG file

To set up Epoxide to perform the required tests, we have to create a troubleshooting graph that describes the scenario as a troubleshooting graph. Epoxide uses a Click-inspired language for describing such graphs. Here we provide a simple

one of that. TSGs are recognized by Emacs/Epoxide as being stored in .tsg files. For the current task we can come up with a .tsg shown in Figure 10 (it can be found in the project examples by the name of pingview.tsg).

In this simple example let's split the definition into two main logical parts: the first is the description of nodes and views of the troubleshooting graph and the second is the description of the links between them.

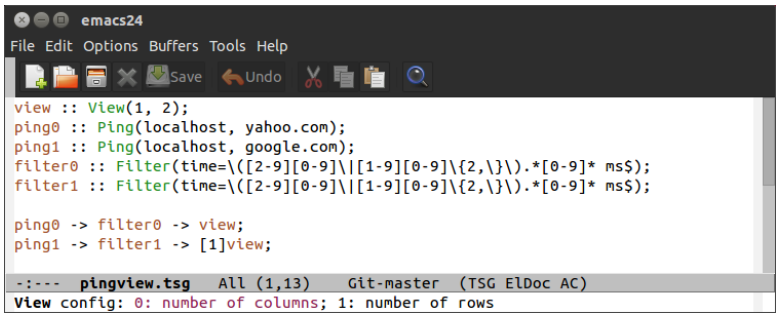

Figure 10: pingview.tsg with Eldoc support.

We define nodes for pinging yahoo.com and google.com and filters that highlight response times longer or equal to 20 ms and a view that would serve as a means to provide a solution for the common need to see multiple outputs aligned at once in a simple yet flexible way. Views can be added to TSGs as they were nodes.

A node is specified by its instance name (e.g. ping0) and its class (e.g. Ping). A class is assigned to the instance by using the :: operator. Following the class declaration, node configuration parameters can be specified in parentheses.

Links are then created that tie the nodes together. The output of a node is connected to the input of another by adding the -> operator. Nodes can have multiple in/outputs. To distinguish between these write the number of the appropriate in/output in brackets. Outputs should be written after the instance name of the node, inputs before them. When using the 0^{th} in/output, you can use the simplification of omitting the brackets and numbers altogether. Multiple outputs can be linked to multiple inputs between two nodes by listing the appropriate numbers separating them with a comma (e.g. node1[1, 2, 3] -> [4, 5, 2]node2 would link the 1^{st}, 2^{nd} and 3^{rd} output of node1 to the 4^{th}, 5^{th} and 2^{nd} input of node2 respectively). Note: you should list equal number of outputs and inputs.

Since views are a special kind of nodes, one can use the -> operator to link outputs of a node to a view. In case of a node's own buffer contains the data of interest the --> operator should be used. E.g. node --> view means that the node's own buffer will be displayed when switching to view.

When creating views one can define the layout of the view by specifying the row and column counts of the view: it then splits the Emacs window to sub-windows in the specified layout. If no layout specification is given then the the Emacs frame is split into as many sub-windows as many inputs the selected view has and each such input gets displayed in a sub-window. When using this setup, the class definition of the view can be omitted: an object in the .tsg file is assumed to be a view by default when it does not have a class assignment. When no view was defined in the .tsg file, a default view is created that always splits the Emacs window into two halves and uses these to display the .tsg file and the output buffer that has been used in the last assignment in the .tsg file.

An expression (instance declaration and linking) should always be terminated by a semicolon. A node instance declaration can be written inline in a linking expression. If you use a node only in one expression you can omit the instance name part: by doing so the framework will generate a unique name for this node.

It is really easy to create new TSGs or modify existing ones due to the convenient TSG editing features of Epoxide:

- syntax highlighting visually separates node classes and instances (see Figure 10),

- Eldoc integration provides short documentation for node configuration parameters (Figure 10),
- autocomplete support brings intelligent code completion capability (Figure 11),
- and keyboard shortcuts control the execution of the TSG and the nodes.

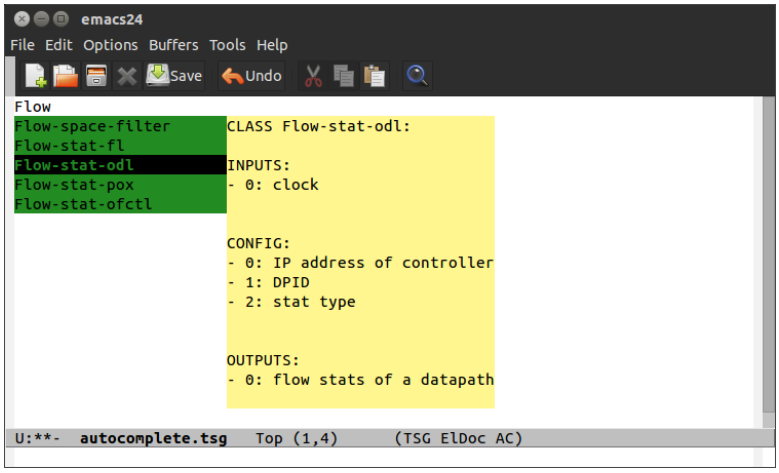

Figure 11: Autocomplete support in Epoxide

Since now we are familiar with the syntax of the TSG definition file we can take a look at a simple execution scenario.

7.4.2 Executing the TSG

Executing a TSG describing the troubleshooting process itself is as simple as opening the corresponding .tsg file. For starters let's see an even simpler scenario then described before that uses only one Ping and one Filter node. Let's open it now by issuing C-x C-f <path to project examples>/ping.tsg RET. Depending on your setup the execution starts automatically or you have to press C-x C-e then hit y at the confirmation question to start the execution. (This behavior can be customized in the epoxide customization group by changing the value of the variable epoxide-start-tsg-on-file-open.) As the execution starts, the Emacs window splits horizontally and you can see the .tsg file and the output of the last link in the TSG (see Figure 12). Currently this link is the output of the ping with lines emphasizing greater or equal than 20 ms response time.

Suppose we want to check the connectivity with a different target. All we have to do is modify the second parameter of the Ping node and reevaluate the .tsg file with C-x C-e y. (Or you can move to the node's buffer and change the parameter using the runtime node reconfiguration option as described later.)

At this point we have a running TSG which we want to examine. The main purpose behind examining a TSG in details is to see the data-flow itself and to modify node variables on-the-fly. In order to aid traversing a TSG, Epoxide can provide a graphical representation of the TSG by pressing M-g t or C-x g (see Figure 13). Nodes can be easily accessed from this view by clicking on their name or using key bindings (C-x p and C-x n).

Every node has its own buffer where a small help is provided and the node's variables can be accessed by pressing v; these variables can be modified on-the-fly. Links between the nodes can be altered this way also.

It is also possible to navigate through the TSG via the links. Press C-x n or M-n to jump to the following link/node, and C-x p or M-p to the previous one. If you want to jump back to the .tsg file, press M-g e.

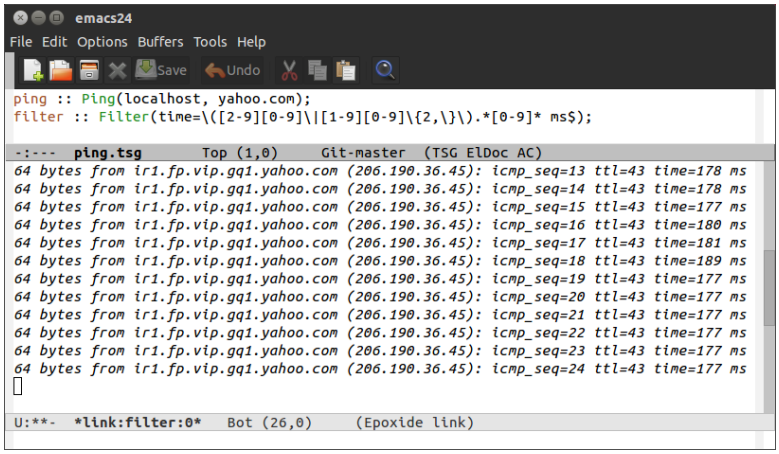

Figure 12: The source file of ping.tsg and its output

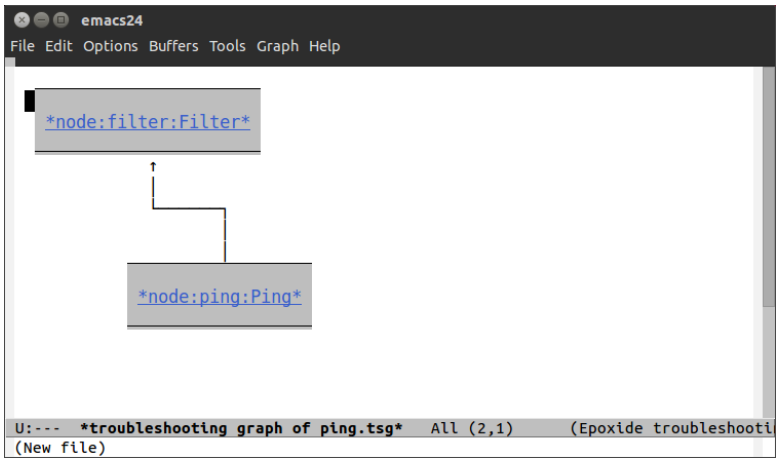

Figure 13: Graphical representation of ping.tsg

One might want to perform two pings at the same time. The modifications of the TSG definition file to achieve this were discussed in the previous section. Comparison of the simple case and the one using parallel pings is left to the reader.

If there are views in a TSG, the last one will be activated at the beginning of the execution. Other views can be activated with the numeric prefix M-g v key binding, or you can cycle through the defined views by pressing M-p or M-n.

7.4.3 Creating a more complex TSG file

To create a more complex scenario these basic elements might not always be enough.

Configuration arguments can be specified when declaring a node (called static argument) or assigning outputs to the configuration arguments (called dynamic argument). In the first case the arguments should be listed during the node declaration, in the second case assignment happens similarly as when assigning a node's output to another's input. In this latter case the 1-based index of the configuration argument should be given on the receiving end of the assignment. I.e. node1[0, 1] -> node2[1, -1] would link the 0^{th} output of node1 to the 1^{st} input of node2 and the 1^{st} output of node1 to the first configuration argument of node2. When reading a configuration argument from an

output buffer the last line of that buffer is substituted in the place of the argument. When a configuration argument has both static and dynamically assigned values, the dynamic value takes precedence over the static one. While static configuration arguments can be changed at run-time, dynamic arguments cannot be changed that way.

When facing a complex troubleshooting scenario one might find the need to have nodes that are able to interpret their inputs or compare those to each other. In Epoxide's case we call these nodes Decision nodes.

Decision nodes provide options for selecting among different inputs. An unspecified number of inputs can be connected to the node, and after performing evaluation on these inputs a decision is communicated on one of the node's outputs. Output #0 serves as a positive output (i.e. there was at least one input that satisfied initial criteria) and output #1 is a negative output that displays a timestamp whenever there were no inputs satisfying initial criteria. The number of the node configuration arguments depend on the number of inputs:

- The first argument decides whether the decision should be made only when data is available on every inputs. If some inputs lag behind or need more time to produce data this option can be used to wait for them. A nil value of this argument specifies not to wait, non-nil values make the node wait for every inputs.
- The second argument specifies a timeout. When using it together with a non-nil value of the first argument, a decision is always made at the timeout even if not every input is present. In case of the first argument having a nil value and a timeout is set, when no input has any data at the timeout, a negative output is given. Giving nil as the timeout argument switches the function off whereas giving it a value greater than 0 turns it on. The Decision node itself does not have a timer, so in case a timeout is to be used a Clock node should be connected to the Decision node. It is advised that the Clock node is connected to the last input of the Decision node (this way node configuration arguments should be given corresponding to this input). Connecting more than one Clock nodes to the Decision node results in the Decision node selecting the first Clock node for the timeout calculation. The second configuration argument of the Decision node should be set to the tick count when we would like to have the timeout to occur (since the Clock node only supplies tick counts). E.g. if we want to have a timeout of 5 seconds, Clock(1) (that is a clock giving out a tick in every second) can be specified in conjunction with a Decision node having its second argument set to 5 (or a Clock(0.1) and a timeout value of 50 would have almost the same effect).
- The third argument is used for selecting the proper input to be dispatched to the output. E.g. using the or function when there is at least one input satisfying criteria, the one having the lowest index is going to be displayed on the positive output. In case no input satisfies its corresponding criteria, the Decision node's negative output will be triggered.
- The following arguments should be defined on a per input basis. Let's assume we are dealing with the 1st input:
  - Argument #4 defines whether input 1 should be processed line-by-line. When giving a nil value, everything read in the current iteration is processed in one bulk, whereas giving a non-nil value results in splitting the read data to lines and making an individual decision on each line.
  - Argument #5 specifies whether the result displayed on the Decision node's positive output should be the same as on the input (a non-nil value activates this option). Passing nil to this argument would mean that the result displayed on the positive output should be taken from the output of the input's decision function. Using this case, processing can be made on the data within the Decision node.

- Argument #6 is the decision function assigned to input 1. A decision function can be any elisp function that takes at least one argument, and its output is either nil or non-nil. The decision function serves as the way to examine whether input 1 satisfies some criteria or not.
- Argument #7 defines on which argument of the decision function should the data of input 1 be passed.
- Argument #8 defines the number of arguments of the decision function additional to the data of input 1.
- Arguments #9+ should be the additional arguments of the decision function. When specifying the arguments, they should be listed in the order in which the elisp function requires them but the argument requiring the data from input 1 should be left out. Here other inputs can be passed to the decision function using the form 'input-x where x denotes the index of the input.

To display results collected from different Decision nodes, Decision-summary nodes can be used that are able to display their inputs in a table format listing which Decision node produced successful and unsuccessful outputs.

Epoxide provides the option to use any Emacs buffer as an input. To achieve this, the Emacs-buffers node has to be applied that has no inputs and takes an indefinite number of configuration arguments and provides the same number of outputs as the number of its configuration arguments. The configuration arguments specify which Emacs buffer the node has to look for changes and the changes in these buffers are copied to the node's outputs in the same order as they were given by the configuration arguments (i.e. the first specified buffer (in the configuration list) is connected the to first output, the second to the second and so on).

To do processing on a node's output one can choose between two options. The first is to use the simpler Format node that is able to format strings coming in on its inputs. The node works like the format elisp function but it only accepts string control symbols (since everything arriving on the inputs is string). The other method is to use the more diverse Function node that is able to call elisp functions or lambda expressions.

To have a grasp on what Epoxide can do now, a short description of currently implemented nodes that have not been mentioned before is provided here:

- Arp (Address Resolution Protocol): the node is able to query the ARP cache by evoking the arp command.
- Clock: a simple node that updates its output in given time intervals.
- Command: a general node for wrapping any shell commands that are run as Emacs subprocesses.
- Doubledecker: this node is able to connect to a DoubleDecker broker, receive and send messages to a partner or to a topic.
- Dpids-<OpenFlow controller type>: the node collects DPID (Datapath ID) and switch name information from an OpenFlow controller.
- Escape: the node is able to query NFFG topology information from Escape.
- Flow-space-filter: this node provides support to select only that flow space that is wished to be seen.
- Flow-stat-<OpenFlow controller type>: this node provides functionality to query an OpenFlow controller for flow statistics of a specific switch given with its DPID.
- Gdb (GNU DeBugger): the node is able to attach GDB to a running process.

- Graph: this node provides graph visualization support.
- Host: it is a node to perform DNS lookup using the host shell-call.
- Ifconfig: the node wraps the ifconfig shell command.
- Iperf: wraps around the iperf command.
- Json-filter: this node has one input that supplies the JSON expressions to be processed.
- Rest-api: the node is able to make a REST API call and display its results on its output.
- Table-view: is a node for displaying data received on its inputs in a table form.
- Topology-<OpenFlow controller type>: this node is used for querying topology information from an OpenFlow controller.
- Traceroute: the node is a wrapper for the traceroute process.

7.4.4 Implementing new nodes

Currently, Epoxide can only provide a limited number of predefined nodes, therefore, cannot cover every unique use-case. On such occasions, the functionality of Epoxide can be easily extended by implementing new nodes. Nodes are written in Emacs Lisp using some Epoxide-specific data structures which are well-documented in the source code.

To implement a new node the following three Epoxide framework-specific functions have to be included in the node describing file:

- epoxide-<nodename>-start is called when the node initializes,
- epoxide-<nodename>-exec contains the function of the node, this is the function called by the scheduler during TSG execution,
- epoxide-<nodename>-stop is called when the node is being cleared up or stopped.

In these function names <nodename> should match the node definition file name in all lower case letters. Additional documentation-aiding functions can be added to a node:

- epoxide-<nodename>-input-info provides documentation, value hints and validation for input fields,
- epoxide-<nodename>-config-info provides documentation, value hints and validation for configuration fields,
- epoxide-<nodename>-output-info provides documentation, value hints and validation for output fields.

Epoxide provides common functions that aid node development. A node can use the functions provided by Epoxide or the developer can choose to use a proprietary implementation. The standard functions cover the following areas:

- epoxide-node-read-inputs: reads a node's inputs and marks the position in each input buffer until which it read the input. One can choose to read inputs only when the node's every input has valuable data. The function returns a list containing the buffers, the positions and the read content. The developer can use the function epoxide-node-get-inputs-as-string to concatenate each read input string to a single string or epoxide-node-get-inputs-as-list to convert the read data into a list that contains only the read text.
- epoxide-node-get-config: reads node configuration arguments (static or dynamic),
- epoxide-write-node-output: writes node outputs.

Existing node definition files are collected in <path to project>/src/nodes/ but this location can be customized by changing the value of the variable epoxide-nodes-files-directory. Node definition files should be Emacs Lisp files and their names should start with a capital letter (e.g. Nodename.el).

7.4.5 Keyboard shortcuts

We present the full extent of the specific Epoxide key bindings here.

Troubleshooting Graph (TSG) execution:

- C-x C-e clears any previous information and restarts execution of the TSG,
- C-c C-e restarts the execution of the TSG but keeps previous information (this can be used for adding nodes and links to the TSG).

TSG modification:

- C-c C-a adds a new node to the TSG,
- C-c C-r restarts a node.

Navigation, views, etc:

- v shows the Epoxide variable list,
- c customizes Epoxide variables,
- C-x C-b shows the Epoxide specific Ibuffer list,
- C-x g and M-x t show the TSG visualization,
- M-g e shows the TSG file,
- C-x n jumps to the following node,
- C-x p jumps to the preceding node,
- C-c C-o shows every output buffers of the active node,

- M-n jumps to the next buffer in the TSG,
- M-p jumps to the previous buffer in the TSG,
- C-[1-9] M-g v activates View #[1-9],
- M-N activates the next View,
- M-P activates the previous View.

7.5 Future work

There are two ways Epoxide can be extended. By adding more nodes one would be able to design more complex TSGs and adding a node recommendation system would help creating these TSGs. Such a recommendation system could analyze previous TSGs to suggest new nodes to add to the current troubleshooting scenario or use an expert system to diagnose problems and suggest nodes that best narrow down the possible root causes. The first approach would take Epoxide to a more community based direction where users could share their TSGs to make node suggestions better for the network problems of others. The second approach would give way to automatizing the whole troubleshooting process.

References

[Civ+16] Ruth Civil, Al Morton, Lianshu Zheng, Reshad Rahman, Mahesh Jethanandani, and Kostas Pentikousis. Two-Way Active Measurement Protocol (TWAMP) Data Model. Internet-Draft draft-ietf-ippm-twamp-yang-00. Work in Progress. Internet Engineering Task Force, Mar. 21, 2016. 58 pp. url: https://tools.ietf.org/html/draft-ietf-ippm-twamp-yang-00.

[D4.2] Rebecca Steinert and Wolfgang John (editors). Deliverable 4.2: Service Provider DevOps network capabilities and tools. Tech. rep. UNIFY Project, 2015.

[D4.3] Guido Marchetto and Riccardo Sisto (editors). Deliverable 4.3: Updated concept and evaluation results for SP-DevOps. Tech. rep. UNIFY Project, 2016.

[Gro+12] JSON-RPC Working Group et al. JSON-RPC 2.0 specification. 2012. url: http://www.jsonrpc.org/specification.

[Joh+15] W. John, C. Meirosu, B. Pechenot, P. Sköldström, P. Kreuger, and R. Steinert. "Scalable Software Defined Monitoring for Service Provider DevOps". In: Proc. EWSDN 2015, Bilbao, Spain. Oct. 2015.

[Koh+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. "The click modular router". In: ACM Transactions on Computer Systems (TOCS) 18.3 (2000), pp. 263–297.

[KS15a] P. Kreuger and R. Steinert. "Scalable in-network rate monitoring". In: Proc. IFIP/IEEE IM, Ottawa, Canada. May 2015.

[KS15b] Per Kreuger and Rebecca Steinert. "Scalable in-network rate monitoring". In: Integrated Network Management (IM 2015), 2015 IFIP/IEEE International Symposium on. May 2015.

[Lév+15] Tamás Lévai, István Pelle, Felicián Németh, and András Gulyás. "EPOXIDE: A Modular Prototype for SDN Troubleshooting". In: ACM SIGComm (demo) (2015), pp. 359–360.

[McG07] Paul McGuire. Getting started with pyparsing. O'Reilly Media, Inc., 2007.

[Nel+16] Derek Nelson, Jason McSweeney, Jeff Ferguson, Usman Masood, and Josh Berkus. Homepage of PipelineDB. 2016. url: https://www.pipelinedb.com/.

[Ném+15] F. Németh, R. Steinert, P. Kreuger, and P. Sköldström. Roles of DevOps tools in an automated, dynamic service creation architecture. IFIP/IEEE IM, Demo Session, Ottawa, Canada. May 2015.

[Pan+14] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. "Verifying Isolation Properties in the Presence of Middleboxes". In: CoRR abs/1409.7687 (2014).

[Pel+15] István Pelle, Tamás Lévai, Felicián Németh, and András Gulyás. "One tool to rule them all: a modular troubleshooting framework for SDN (and other) networks". In: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research: SOSR 2015 (2015), Article No. 24, 7 p.

[PMM15] Bertrand Pechenot, Jonas Mårtensson, and Farnaz Moradi. "Monitoring Transport and Cloud for Network Functions Virtualization". In: Proc. EWSDN 2015, Bilbao, Spain. Oct. 2015.

[RFC5357] Jozef Babiarz, Roman M. Krzanowski, Kaynam Hedayat, Kiho Yum, and Al Morton. A Two-Way Active Measurement Protocol (TWAMP). IETF RFC 5357. Oct. 14, 2015. doi: 10.17487/rfc5357. url: https://rfc-editor.org/rfc/rfc5357.txt.

[TK16] Brian Trammell and Mirja Kühlewind. mPlane Protocol Specification. Internet-Draft draft-trammell-mplane-protocol-01. Work in Progress. Internet Engineering Task Force, Mar. 16, 2016. 47 pp. url: https://tools.ietf.org/html/draft-trammell-mplane-protocol-01.