Network Service Assurance and Telemetry optimisation using Heuristics

[1]MIOLJUB JOVANOVIC, [2]MILAN CABARKAPA, [1,2]DJURADJ BUDIMIR
[1]Wireless Communications Research Group University of Westminster London, UK
[2]Department of Telecommunications School of Electrical Engineering University of Belgrade Belgrade, SERBIA

Abstract: This paper identifies possible solution to reduce amount data retrieved for purpose of network monitoring, while retaining the quality of information. Legacy methods used for network service monitoring data using and data retrieved by use of synthetic tests are usually decoupled – describing different contexts. As a result, there is too many monitored operational parameters on monitored devices and consequently too much measurement data exported from network service probes. Novel approach is suggested to perform heuristic analysis of the network service configuration and decomposition of collected data at the network edge. Objective is that amount of collected data can be reduced while at the same time quality of the presented information can be improved by providing clear correlation between network service and telemetry data.

Keywords: intent based network; intent-aware monitoring agent; model-driven telemetry; service assurance formatting.

Received: October 8, 2021. Revised: June 21, 2022. Accepted: August 12, 2022. Published: September 2, 2022.

1. Introduction

Network and performance monitoring is an important task and responsibility for every Network Service Provider (NSP) operator. There are different methodologies onto how network traffic and key parameters for devices and network services are being monitored and processed, such as Netflow/SFlow, SNMP (RMON), Model driven telemetry (MDT)/streaming telemetry, command line interface (CLI) outputs, synthetic network tests etc . One point that is in-common is that all monitoring methods are aimed at establishing whether service level agreement (SLA) of the desired network service has been fulfilled. For example, SLA compliance data generated by synthetic network tests give us info about state of the network service as "End to End" result, however synthetic test results give no insights onto what could be causing service degradation, since synthetic tests don't have awareness of the data path through the network. With legacy methods such as Netflow, SNMP, CLI it's usually possible to retrieve operational data about network elements, but there is very little or no information about the state of the network services that are being deployed on those elements. In recent years, with advancements of streaming telemetry we're receiving even more data on the network elements, devices, yet very little information in the context of network service, which could help us to identify cause of the service level degradation. Overall, it is very difficult for network operator to identify what is the cause of network service degradation, since each monitoring method operates in its own context: legacy device operational data is focused on devices and don't contain info on services, synthetic tests have high level service info and don't have insights on the data path, telemetry is streaming operational data with no service context information. Hence, there is a need to create correlation between all the data points which will provide insights in the service context both on high level, as well as low-level – to help identify the cause of the service degradation.

2. Decomposition of the Network Service Configuration

To understand which data is available for monitoring and how to monitor the network service in question it's necessary to perform service decomposition and establish dependencies between different service components. Performing Network Service Decomposition on the configuration deployed by a service orchestrator, such as NSO[2], ONAP[3] or similar involves analysing the configuration using the heuristics to perform mapping between Service Configuration all the way to the monitored data – such as YANG path, SNMP Object Ids or CLI outputs. The analysis is performed in the following order: Network service configuration, service components, expressions, metric (YANG path).

Example of the Assurance Expression Tree:

Assurance hierarchy between following network Service components

1. Service components: tunnel, VPN, ...
2. Service components: (sub)interface, device, interior routing protocol
3. Service components can depend on other services
4. Service components are assured on the agents

The above-mentioned decomposition is illustrated on Fig. 1, where GRE Tunnel network service has been decomposed to its service components. GRE Tunnel network service

obviously depends on health of Tunnel Interface, which in-turn depends on the underlying physical/logical interface. Ultimately, underlying interface health will also depend on the device where the specific interface resides. Apart from the GRE Tunnel Interface, GRE Tunnel as a service also depends on the Layer3 connectivity, which is also dependent on the health of interior routing protocol on the device itself. Finally, routing protocol's health depends on the health of egress interface, which is forward the network traffic, along with the routing protocol information as well.

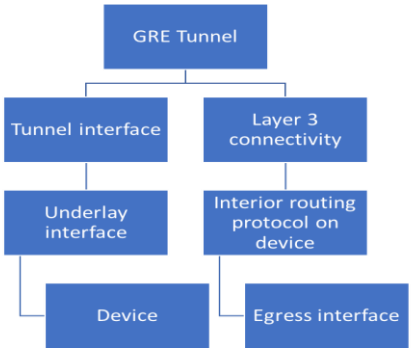

Fig. 1. gre tunnel network service assurance decomposition

3. Analysis Engine

Main hypothesis of the research is to confirm whether amount of the telemetry data could be reduced by leveraging service awareness by means of heuristics. In the Fig. 7 experimental network topology along with the deployed assurance agents have been shown.

Analysis Engine: maps abstract metrics to device-specific metrics implementation

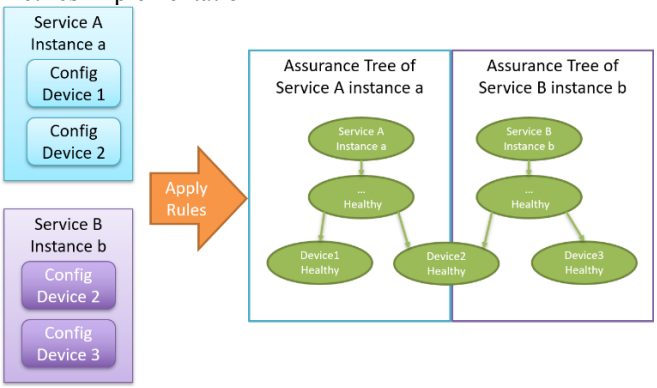

Fig. 2. applying rules on the configuration of a service instance results in an assurance graph that connects service components according to their dependencies.

Transformation of an assurance graph with one service instance depending on two service components into an expression graph. Following step is to apply techniques described in the above work to the ontology-based system, which could be used as foundation to the reasoning engine – for analysis and automated error detection.

Expression tree for the service component can be quite complex, which makes it inherently difficult to navigate through large number of different computations and dependencies.

By applying advanced data modelling techniques, such as Ontologies, we could establish inferred relationships between raw data sources such a MDT and legacy protocols: SNMP, Syslog, RMON or even CLI to ensure higher level of precision and relevance within calculated assurance value for any specific service component assurance expression tree.

Following the Ontology based model and reasoning engine, we could potentially insert inferred rules determine causal relationships using reasoning engine instead of heuristic packages build by a Subject Metter Experts.

4. Heuristic Packages

Notion of heuristic packages has been created and used to encode human knowledge within the rule files with the objective to:. 1. Automate building and monitoring of assurance cases for network services and to 2. Enabling exchange and reuse of assurance case between experts and operators

By naming *heuristic* objective is to cover a large part of most common issues. However, in some cases:

- A service reported as "Broken" might actually be functional
- A service reported as "Healthy" might actually be non-operational

In such cases, heuristics packages can be extended to improve coverage and accuracy.

Heuristic packages contain 3 hierarchical layers, higher ones depending on lower ones:

1. Metric Engine: Device-Independent abstraction of metcis to fetch and process
2. Service Components: Compute status of a specific part of the networking system, based on Metrics.
3. Rules: Combine Service Components to assure a whole network service, based on service configuration

Metric engine provides an abstraction of metrics that can be extracted from network devices. Selects device-specific implementations for each metric.

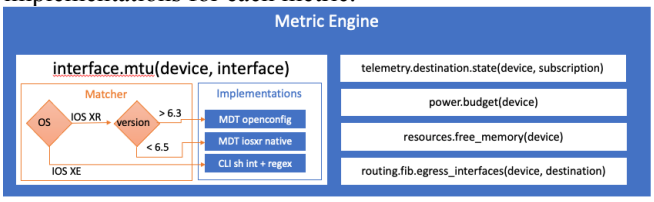

Fig. 3 Metric Engine

Service component focuses on a very specific and well-scoped part of the networking system. Service component reports status by performing evaluation and returns health score in the interval between 0(broken) to 1(healthy) + symptoms. Service components are reusing metrics from the metrics engine.

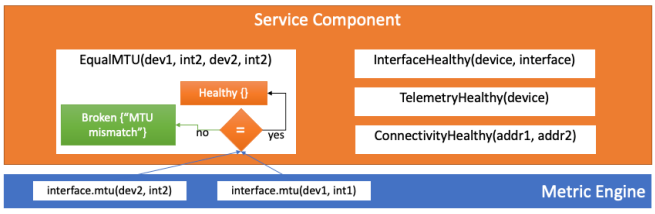

Fig. 4 Service component

Rules are expected to parse configuration pushed by Network Configuration Orchestrator (NSO, ONAP) to enable a service to produce an assurance graph. Assurance graph: service

components with parameters from the configuration and dependencies. Dependencies aggregate symptoms to explain services malfunction.

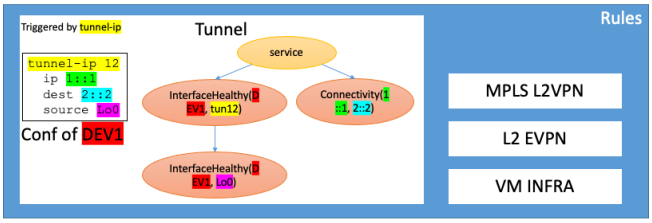

Fig. 5 Rules

Heuristic packages are prepared and created by input from Subject Matter Experts (SMEs) by conducting following steps, aligned with previously defined heuristic package structure:

Metric Engine: SME specifies method for retrieving given metric for a device platform or platforms involved in network service configuration. Currently this step is coded in JSON.

Service components: SME lists symptoms and trigger conditions based on the metrics. This step is coded in Service Component Domain Specific Language (SCL), newly defined language described in more detail below

Rules: SME analyses and decomposes service components needed for a network service feature. Currently logic for this step is implemented in Python.

Service component language (SCL), Domain Specific Language (DSL) has been developed to describe the computation needed to evaluate the status of a service component.

Overview of SCL:

Service components is specified by the following elements:

- a name and a list of arguments
- a list of metrics to collect
- a list of expressions to combine the metrics into a single status value

To further describe and explain SCL we'll demonstrate each of these elements through examples. The complete specification of the syntax is detailed below.

Global structure of a heuristic package file hierarchy:

Each file as the following structure:

- a single level 1 header (The name of the service component)
  A description of the service component
  - a single level 2 header 'Arguments'
    The arguments of the service component
  - a single level 2 header 'Expressions'
    a sequence of level 3 blocks (either 'Measure' or 'compute')
    - 'Measure' blocks contain a list of metrics to monitor
    - 'Compute' blocks contain a list of expressions to compute

Name and arguments:

The name is the level 1 header of the file. Arguments are defined in a list using '*' as an item:

```
# InterfaceHealthy
Checks whether a given interface on a given device
is healthy.
## Arguments
  * device device:  Device supporting the
interface to check.
  * str interface:  Name of the interface to
check.
```

The above example defines the name and arguments of InterfaceHealthy. An instance of this service component is totally parameterized by a device (of type device) and an interface (of type str).

Metrics: Each metric to collect is defined via a name and a set of parameters. An example is

```
* str admin_status =
interface.administrative_status(device=device,
interface=interface)
      _ Whether the interface is currently enabled_
```

Expressions: Each expression is defined via a name, optionally a symptom to raise, and a list of potential expressions. An example reusing the previous metric is

```
* is_up:  Whether the interface is currently
enabled
    broken if false: Interface is down
    + `admin_status == "UP"`
```

Here a single expression is used. If this expression evaluates to false, then a symptom is raised. However, depending on the available metrics, we might want to use different expression to compute a given value. For instance, assuming that two devices: * device 1 provides "total*memory" and "free*memory" metrics * device 2 provides "total*memory" and "used*memory" metrics

In that case, if we want to check that at least 10% of the memory is available, one can write:

```
* memory_healthy: At least 10% of the memory is
available
     + `free_memory/total_memory > 0.1`
     +
`Minus(total_memory,used_memory)/total_memory >
0.1`
```

Convention: A possible way to organize the expressions section is to decompose the service component. For instance, DeviceHealthy can be divided into cpu*healthy*, *memory*healthy, storage_healthy ... For each subexpresssion, include a "Measure" block will all the metrics (i.e. relative to CPU) followed by a "Compute" block with all the expressions needed to assign a value to CPUHealthy.

Finally, the last expression shall combine all expressions of the subparts into a single expression summarizing the value.

Operators: The language in itself does not define any operator since operators need to be externally defined, by convention in the expressions folder.

Metrics**:** In the proposed DSL, name of the metric and is created to reflect definition of a metric and a set of parameters. To actually specify how to retrieve a value for a given device with a given OS, one has to add an entry in the relevant file of the metrics folder.

Spaces tabs and new lines are ignored except otherwise specified.

The syntax of the language is compatible with Markdown: i.e. if spaces and new lines are correctly ordered, the file will correctly render in Markdown. However, it is also possible to write syntactically correct file that do not render well in Markdown.

Syntax : Here is a human readable version of the syntax. Non terminals are in lower case, terminal are in UPPER CASE.

```
service_component := header arguments expressions
```

Header: The file starts with a header that defines the name of the service component and a description of the service component.

```
header := '#' ID SUB_DESCRIPTION
ID : any sequence of letters, numbers or '_' starting by either a letter or an '_'
SUB_DESCRIPTION: anything that does not contain '#'
```

The ID terminal is used for every ID in the syntax. The spaces IN the description are kept, and as long as the character '#' is not met, the sequel is considered part of the description. Thus it is possible to use any markdown except title with "#" in the description.

Arguments: The arguments start with a level-2 Markdown title, followed by a list of a least one argument and optionally some global parameters. ``` arguments := '## Arguments' (argument)+ (display_params)?

```
argument := '*' ID? ID ':' LINE_DESC

LINE_DESC: any string not containing a newline. ```
```

In the argument syntax, the first ID indicates (optionally) the type and the second one is the actual name of the argument. The LINE_DESC contains a description of the argument, terminated by a new line.

The display parameters indicate how to render the service component in a GUI.

```
display_params :=  'display level' '=' ID
'web_label' '=' web_label
web_label := BACKQ3 TEXT ('{' ID '}' TEXT)* BACKQ3
BACKQ3 : 3 backquotes ('`')
TEXT: any string not containing '{'
```

The display level ID should be one of the levels defined in ServiceInstance Class definition.. The web label contains an arbitrary string with some IDs enclosed in braces. The IDs should be arguments declared before. The web label will be formatted using the python 'format' function to replace argument's ID in braces with the value of the argument.

Expressions: The expressions start by a level 2 title 'Expression' and contains a sequence of measurments and computations.

```
expressions := '## Expressions' ( measurements | computations )*
```

Measurements: A set of measurements is introduced by the level 3 title 'Measure' followed by a comment (optional), and then a list of measurement (i.e. metrics) to obtain. As explained above in this document, the metrics have to be defined in the metrics folder.

```
measurements := '### Measure' COMMENT measurement+

measurement := '*' ID? ID '=' METRIC_NAME '('
measurement_parameter (',' measurement_parameter)*
')' '_' M_DESCR '_'

measurement_parameter := '-' ID '=' ID

COMMENT : any string not containing '*'
METRIC_NAME: identifiers separated by dots
M_DESCR: any string not containing '_'
```

For each measurement, we have two IDs, the first one, optional, indicates the type of the metrics, the second one indicates the name of the measurement. After the equal sign, the definition of the metric instance to associate to the measurement name is given. For instance:
`interface.mtu(device=source_device, interface=source_interface)`, where source*device and source*interface are arguments of the service component.
The Metric Name should match an existing metric in the metric engine. The parameters name should be existing parameters of that metric. Finally, the description of the measurement, which should not contain `_` is enclosed between `_`.
Computations: A set of computations is introduced by the level 3 title 'Compute' followed by a comment (optional), and then a list of computations.

```
computations := '### Compute' COMMENT computation+
computation := '*' ID ':' LINE_DESC symptom?
expression_decl+
symptom := LEVEL CONDITION: LINE_DESC
LEVEL: 'broken' | 'degraded'
CONDITION: ('if false'|'if true')
expression_decl := '+' '`' expression '`'
```

A computation is defined by: * a name * a one line description * an (optional) symptom * at least one expression declaration
The name is defined by the first ID in 'computation'. It should not be used by a previous argument, measurement or computation.
The symptom can only be added if the expression evaluates to a boolean value. Level and Condition indicates the level of the symptom (degraded =~ warning and broken =~ error).
There can be several expression definition for the same expression name. If so, the first definition in order for which all subexpressions (i.e. references to other expresions, arguments or metrics) are available is taken.
This mechanism allow to have flexibility in the expression as shown above.
The label of the symptom can contain expressions enclosed between backquotes '`'. In that case, the expression is replaced by its value whenever the symptom is raised (the expressions are always evaluated.) As a corollary, the compiler does not allow the expressions used in the symptom labels to depend on a metric that is not already a dependency of all expression alternatives. For instance:

```
### Measure
 * admin_status: Administrative status of the
interface
   [...]
 * errors_count: Number of errors on the interface
   [...]

### Compute
```

```
 * interface_healthy: Whether the interface is
healthy
   broken if false -> Interface is not up status:
`admin_status` or too much errors `errors_count`
   + `admin_status == "Status UP" and errors_count
< 10`
   + `admin_status`
```

will not compile because the symptom label depends on errors_count but the last alternative doesn't.

Expression The expressions have the following syntax:

```
expression := sum (cmp sum)?
cmp := '==' | '<=' | '<' | '>=' | '>'
sum := factor ('+'  factor)*
factor := atom (( '*' | '/') atom)*
atom := ID | INT | FLOAT | STRING | BOOL | '('
expression ')' | call
call := ID '(' expression (',' expression)* ')'
```

This syntax supports expressions using the arithmetic operators '+', '*' and '/', the comparision operators in cmp, identifiers, litterals of floats, ints, booleans and strings, and function calls.

Function calls are used for operators that do not have a infix version. The first ID is the name of the operator, which is checked again classes defined in the expression folder. In particular the number of arguments is checked. The expression and metrics can be declared in any ordered. However, circular dependencies between expressions are not allowed.

5. Service Component Assurance Tree Example

Now that methodology of the service decomposition has been understood, as depicted on **Error! Reference source not found.** along with heuristic packages concept and SCL we could proceed to explain decomposition of the actual tunnel service, along with all its service components, as depicted at Fig. 6. Each box represents the result of the evaluation for the specific service component where boxes coloured in Green represent service components which have been determined as healthy based on the telemetry data, CLI or SNMP outputs, whatever it may it depend upon. Grey boxes represent service components which state couldn't be conclusively determined, since there is insufficient data to deem service component healthy or unhealthy

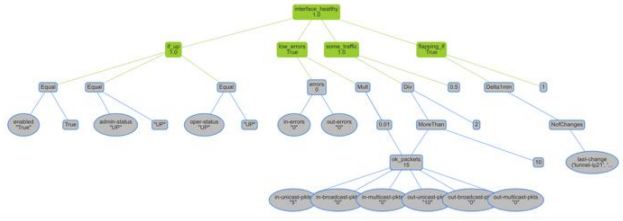

Fig. 6. Graphical representation of the service component assurance tree

Example SCL code which performs service component assurance tree:

```
Is interface flapping.
flapping_if = Delta1min(NofChanges(last-change))
<= 1
Is  interface reported and configured UP.
if_up = (enabled == True) * (admin-status == 'UP')
* (oper-status == 'UP')
Total number of packets correctly received or
sent.
ok_packets = in-unicast-pkts + in-broadcast-pkts +
in-multicast-pkts + out-unicast-pkts + out-
broadcast-pkts + out-multicast-pkts
Total number of errors (input and output).
errors = in-errors + out-errors
Whether the number of errors is low.
low_errors = errors <= 0.01 * ok_packets
Is there some traffic (0.5 -> low traffic, 1.0 ->
normal traffic.)
some_traffic = ok_packets > 10 / 2 + 0.5
Is interface healthy.
interface_healthy = if_up * low_errors *
some_traffic * flapping_if
```

Above mentioned expression and conditions are graphically represented on the Fig. 6. where it can be observed that interface could be considered as healthy (health=1.0, meaning 100% healthy) if all service components are also healthy: interface is up, number of errors is low, there is traffic on the interface and interface is not flapping. Same procedure is then performed on each of the
Branches of the Assurance tree and their logic are discussed below:
Interface is considered "UP" if both administrative enabled state = True and operational state = True
Interface errors are considered low if both in-errors and out-errors are 0. … and so on

6. Experimental Setup

Experimental setup consists simulated customer premises routers (CE) as well as provider core routers (P) and provider edge routers (PE). As shown in Fig. 7, the network with service models is configured using the orchestration network architecture. In the provided example, actual network service intent is to establish communication between Client-1 and Client-3, which in-turn means that communication needs to be established by creating tunnel service between ce-1 and ce-3 network devices. Each of the network devices is streaming telemetry data to the collector, monitoring platform which is receiving and processing all telemetry data.

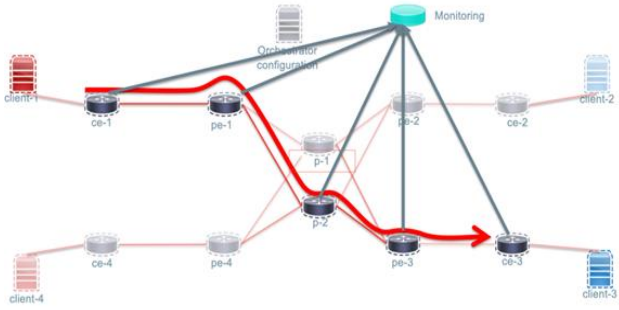

Fig. 7. experimental topology setup with telemetry data streamed from the devices to the monitoring platform

7. Results

Measuring objective was to determine how much data is actually received via MDT under usual telemetry export, with typical data points for router such as environmental, interface stats etc. Result of this work outlines amount of measured data after performing analysis of the incoming telemetry and mapping to service aware MDT. All routers and all incoming data points were taken into account.

As outlined in **Error! Reference source not found.**demonstrated experimental results have reduced the amount of incoming MDT from routers from 5.2 GB to 130 MB, while preserving relevant information which is – is service running and operational per pre-defined KPIs. In short, instead of sending large amount of measured data, model driven telemetry measurements etc, it is possible to send relevant service aware telemetry data which represents computed state of the network service.

8. Conclusion

By leveraging the novel service assurance approach by decomposing service configuration and calculating service health using heuristics, the service definition and construction of this assurance component graph it is possible to reduce amount of telemetry data exported from the network and export only service-intent relevant information instead of raw device data. This in turn means that it's possible to determine service health at the edge and contribute to service assurance in more efficient manner than traditional means of telemetry compression or establishing different channels to send same amount of raw telemetry data. We've presented the designed architecture, which is capable to, at almost real-time, perform analysis of the data streams and perform computations to establish the network service health status.

9. Future Research

Future research would involve applying advanced techniques such as machine learning (ML) or artificial intelligence (AI) on raw data received from monitored devices in an attempt to identify data clusters and dependencies between different data sets. Objective of ML/AI data analysis approach would be to either augment human-defined heuristic packages or to create machine-built heuristics.

References

[1] https://docs.openstack.org/tacker/latest/

[2] https://cloudify.co/

[3] https://www.onap.org/

[4] https://www.cisco.com/c/en/us/solutions/service-provider/solutions-cloud-providers/network-services-orchestrator-solutions.html

[5] Anil Rao, "Reimagining service assurance for NFV, SDN and 5G", White paper, Analysis Mason, 2018.

[6] R. Mijumbi, J. Serrat, J. l. Gorricho, S. Latre, M. Charalambides, and D. Lopez, "Management and Orchestration Challenges in Network Functions Virtualization," IEEE Communications Magazine,vol. 54, no. 1, pp. 98–105, Jan 2016.

[7] A. J. Gonzalez, G. Nencioni, A. Kamisiski, B. E. Helvik, and P. E. Heegaard, "Dependability of the NFV Orchestrator: State of the Art and Research Challenges," IEEE Communications Surveys Tutorials, pp. 1–23, 2018.

[8] M. Pattaranantakul, R. He, Z. Zhang, A. Meddahi and P. Wang, "Leveraging Network Functions Virtualization Orchestrators to Achieve Software-Defined Access Control in the Clouds," in IEEE Transactions on Dependable and Secure Computing, pp. 1-14, Nov. 2018.

[9] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia and P. Casas, "A Survey on Big Data for Network Traffic Monitoring and Analysis," in IEEE Transactions on Network and Service Management, vol. 16, no. 3, pp. 800-813, Sept. 2019.

[10] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo. A Comprehensive Survey on Machine Learning for Networking: Evolution, Applications and Research Opportunities. J. Internet Serv. Appl., 9(16), 2018.

[11] Cisco Systems, Inc, "GitHub Network Telemetry Pipeline," Cisco Systems, Inc, 2017. [Online]. Available: https://github.com/cisco/bigmuddy-network-telemetry-pipeline

[12] M. Jovanović, M. Čabarkapa, B. Clause, N. Nešković, M. Prokin, B. Đurađ, Model driven telemetry using Yang for next generation network applications, 5th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN) 2018, pp. 1186 - 1189, Palić, Serbia, June, 2018.

[13] B. Claise, J. Clarke, and J. Lindblad "Network Programmability with YANG: The Structure of Network Automation with YANG, NETCONF, RESTCONF, and gNMI", Addison-Wesley Book, 1st edition, 2019.

Creative Commons Attribution License 4.0 (Attribution 4.0 International, CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US