OPSAWG
Internet-Draft
Intended status: Informational
Expires: May 6, 2020

B. Claise
J. Quilbeuf
Cisco Systems, Inc.
November 3, 2019

Service Assurance for Intent-based Networking Architecture
draft-claise-opsawg-service-assurance-architecture-00

Abstract

This document describes the architecture for Service Assurance for Intent-based Networking (SAIN). This architecture aims at assuring that service instances are correctly running. As services rely on multiple sub-services by the underlying network devices, getting the assurance of a healthy service is only possible with a holistic view of network devices. This architecture not only helps to correlate the service degradation with the network root cause but also the impacted services impacted when a network component fails or degrades.

Status of This Memo

Copyright Notice

Table of Contents

1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Agent (SAIN Agent): Component that communicates with a device, a set of devices, or another agent to build an expression tree from a received assurance tree and perform the corresponding computation.

Assurance Tree: DAG representing the assurance case for one or several service instances. The nodes are the service instances themselves and the subservices, the edges indicate a dependency relations.

Collector (SAIN collector): Component that fetches the computer-consumable output of the agent(s) and displays it in a user friendly form or process it locally.

DAG: Directed Acyclic Graph.

ECMP: Equal Cost Multiple Paths

Expression Tree: Generic term for a DAG representing a computation in SAIN. More specific terms are:

- o Subservice Expressions: expression tree representing all the computations to execute for a subservice.

- o Service Expressions: expression tree representing all the computations to execute for a service instance, i.e. including the computations for all dependent subservices.

- o Global Computation Forest: expression tree representing all the computations to execute for all services instances in an instance of SAIN (i.e. all computations performed within an instance of SAIN).

Impacting Dependency: Type of dependency in the assurance tree. The status of the dependency is completely taken into account by the dependent service instance or subservice.

Informational Dependency: Type of dependency in the assurance tree. Only the symptoms (i.e. for informational reasons) are taken into account in the dependent service instance or subservice. In particular, the score is not taken into account.

Metric: Information retrieved from a network device.

Metric Engine: Maps metrics to a list of candidate metric implementations depending on the target model.

Metric Implementation: Actual way of retrieving a metric from a device.

Network Service YANG Module: describes the characteristics of service, as agreed upon with consumers of that service [RFC8199].

Service Instance: A specific instance of a service.

Orchestrator (SAIN Orchestrator): Component of SAIN in charge of fetching the configuration specific to each service instance and converting it into an assurance tree.

Health status: Score and symptoms indicating whether a service instance or a subservice is heathy. A non-maximal score MUST always be explained by one or more symptoms.

Subservice: Part of an assurance tree that assures a specific feature or subpart of the network system.

Symptom: Reason explaining why a service instance or a subservice is not completely healthy.

2. Introduction

Network Service YANG Modules [RFC8199] describe the configuration, state data, operations, and notifications of abstract representations of services implemented on one or multiple network elements.

Quoting RFC8199: "Network Service YANG Modules describe the characteristics of a service, as agreed upon with consumers of that service. That is, a service module does not expose the detailed configuration parameters of all participating network elements and features but describes an abstract model that allows instances of the service to be decomposed into instance data according to the Network Element YANG Modules of the participating network elements. The service-to-element decomposition is a separate process; the details depend on how the network operator chooses to realize the service. For the purpose of this document, the term "orchestrator" is used to describe a system implementing such a process.""

In other words, orchestrators deploy Network Service YANG Modules through the configuration of Network Element YANG Modules. Network configuration is based on those YANG data models, with protocol/ encoding such as NETCONF/XML [RFC6241] , RESTCONF/JSON [RFC8040], gNMI/gRPC/protobuf, etc. Knowing that a configuration is applied doesn't imply that the service is running correctly (for example the service might be degraded because of a failure in the network), the network operator must monitor the service operational data at the same time as the configuration. The industry has been standardizing on telemetry to push network element performance information.

A network administrator needs to monitor her network and services as a whole, independently of the use cases or the management protocols. With different protocols come different data models, and different ways to model the same type of information. When network administrators deal with multiple protocols, the network management must perform the difficult and time-consuming job of mapping data models: the model used for configuration with the model used for monitoring. This problem is compounded by a large, disparate set of data sources (MIB modules, YANG models [RFC7950], IPFIX information elements [RFC7011], syslog plain text [RFC3164], TACACS+ [I-D.ietf-opsawg-tacacs], RADIUS [RFC2138], etc.). In order to avoid this data model mapping, the industry converged on model-driven telemetry to stream the service operational data, reusing the YANG

models used for configuration. Model-driven telemetry greatly facilitates the notion of closed-loop automation whereby events from the network drive remediation changes back into the network.

However, it proves difficult for network operators to correlate the service degradation with the network root cause. For example, why does my L3VPN fail to connect? Why is this specific service slow? The reverse, i.e. which services are impacted when a network component fails or degrades, is even more interesting for the operators. For example, which service(s) is(are) impacted when this specific optic dBM begins to degrade? Which application is impacted by this ECMP imbalance? Is that issue actually impacting any other customers?

Intent-based approaches are often declarative, starting from a statement of the "The service works correctly" and trying to enforce it. Such approaches are mainly suited for greenfield deployments.

Instead of approaching intent from a declarative way, this framework focuses on already defined services and tries to infer the meaning of "The service works correctly". To do so, the framework works from an assurance tree, deduced from the service definition and from the network configuration. This assurance tree is decomposed into components, which are then assured independently. The root of the assurance tree represents the service to assure, and its children represent components identified as its direct dependencies; each component can have dependencies as well.

When a service is degraded, the framework will highlight where in the assurance service tree to look, as opposed to going hop by hop to troubleshoot the issue. Not only can can this framework help to correlate service degradation with network root cause/symptoms, but it can deduce from the assurance tree the number and type of services impacted by a component degradation/failure. This added value informs the operational team where to focus its attention for maximum return.

3. Architecture

SAIN aims at assuring that service instances are correctly running. The goal of SAIN is to assure that service instances are operating correctly and if not, to pinpoint what is wrong. More precisely, SAIN computes a score for each service instance and outputs symptoms explaining that score, especially why the score is not maximal. The score augmented with the symptoms is called the health status

As an example of a service, let us consider a point-to-point L2VPN connection (i.e. pseudowire). Such a service would take as

parameters the two ends of the connection (device, interface or subinterface, and address of the other end) and configure both devices (and maybe more) so that a L2VPN connection is established between the two devices. Examples of symptoms might be "Interface has high error rate" or "Interface flapping", or "Device almost out of memory".

The overall architecture of our solution is presented in Figure 1. The assurance tree along some other configuration options is sent to the SAIN agents who are responsible for building the expression tree and computing the statuses in a distributed manner. The collector is in charge of collecting and displaying the current status of the assured service instances.

```
Network            +-----------------+       +-------------------+
Service  --------> | (SAIN)          |       | (SAIN)            |
Instance           | Orchestrator    |       | Collector         |
Configuration      +-----------------+       +-------------------+
                       |                        ^
                       | Configuration          | health Status
                       | (assurance tree)       | (score + symptoms)
                       V                        | streamed
                +-------------------+           | via telemetry
                |+-------------------+          |
                ||+-------------------+         |
                +|| (SAIN)            |---------+
                 +| agent             |
                  +-------------------+
                            ^ ^ ^
                            | | |
                            | | |  Metric Collection
                            V V V
      +-------------------------------------------------------------+
      | Network                                                     |
      |                                                             |
      +-------------------------------------------------------------+
```

Figure 1: SAIN Architecture

In order to produce the score assigned to a service instance, the architecture performs the following tasks:

- o Analyze the configuration pushed to the network device(s) for configuring the service instance and decide: which information is

needed from the device(s), such a piece of information being called a metric, which operations to apply to the metrics for computing the health status.

- o Stream (via telemetry [RFC8641]) operational and config metric values when possible, else continuously fetch.

- o Continuously compute the health status of the service instances, based on the metric values.

As said above, the goal of SAIN is to produce a health status for each service instance to assure, by collecting some metrics and applying operations on them. To meet that goal, the service is decomposed into an assurance tree formed by subservices linked through dependencies. Each subservice is then turned into expressions that are combined according to the dependencies between the subservices in order to obtain the expression tree which details how to fetch the metrics and how to compute the health status for each service instances. The expression tree is then implemented by the SAIN agents. The architecture also exports the health status of each subservice.

3.1. Decomposing a Service Instance Configuration into an Assurance Tree

In order to structure the assurance of a service instance, the service instance is decomposed into so-called subservices. Each subservice focuses on a specific feature or subpart of the network system.

The decomposition into subservices is at the heart of this architecture, for the following reasons.

- o The result of this decomposition is the assurance case of a service instance, that can be represented is as a graph (called assurance tree) to the operator.

- o Subservices provide a scope for particular expertise and thereby enable contribution from external experts. For instance, the subservice dealing with the optics health should be reviewed and extended by an expert in optical interfaces.

- o Subservices that are common to several service instances are reused for reducing the amount of computation needed.

The assurance tree of a service instance is a DAG representing the structure of the assurance case for the service instance. The nodes of this graph are service instances or subservice instances. Each

edge of this graph indicates a dependency between the two nodes at its extremities: the service or subservice at the source of the edge depends on the service or subservice at the destination of the edge.

Figure 2 depicts a simplistic example of the assurance tree for a tunnel service. The node at the top is the service instance, the nodes below are its dependencies. In the example, the tunnel service instance depends on the peer1 and peer2 tunnel interfaces, which in turn depend on the respective physical interfaces, which finally depend on the respective peer1 and peer2 devices. The tunnel service instance also depends on the IP connectivity that depends on the IS-IS routing protocol.

```
                     +------------------+
                     | Tunnel           |
                     | Service Instance |
                     +-----------------+
                               |
           +--------------------+-------------------+
           |                    |                   |
    +-------------+      +-------------+      +--------------+
    | Peer1       |      | Peer2       |      | IP           |
    | Tunnel      |      | Tunnel      |      | Connectivity |
    | Interface   |      | Interface   |      |              |
    +-------------+      +-------------+      +--------------}
           |                    |                   |
    +-------------+      +-------------+      +-------------+
    | Peer1       |      | Peer2       |      | IS-IS       |
    | Physical    |      | Physical    |      | Routing     |
    | Interface   |      | Interface   |      | Protocol    |
    +-------------+      +-------------+      +-------------+
           |                    |
    +-------------+      +-------------+
    |             |      |             |
    | Peer1       |      | Peer2       |
    | Device      |      | Device      |
    +-------------+      +-------------+
```

Figure 2: Assurance Tree Example

Depicting the assurance tree helps the operator to understand (and assert) the decomposition. The assurance tree shall be maintained during normal operation with addition, modification and removal of service instances. A change in the network configuration or topology shall be reflected in the assurance tree. As a first example, a change of routing protocol from IS-IS to OSPF would change the

assurance tree accordingly. As a second example, assuming that ECMP is in place for the source router for that specific tunnel; in that case, multiple interfaces must now be monitored, on top of the monitoring the ECMP health itself.

3.2. Intent and Assurance Tree

The SAIN orchestrator analyzes the configuration of a service instance to:

- o Try to capture the intent of the service instance, i.e. what is the service instance trying to achieve,
- o Decompose the service instance into subservices representing the network features on which the service instance relies.

The SAIN orchestrator must be able to analyze configuration from various devices and produce the assurance tree.

To schematize what a SAIN orchestrator does, assume that the configuration for a service instance touches 2 devices and configure on each device a virtual tunnel interface. Then:

- o Capturing the intent would start by detecting that the service instance is actually a tunnel between the two devices, and stating that this tunnel must be functional. This is the current state of SAIN, however it does not completely capture the intent which might additionally include, for instance, on the latency and bandwidth requirements of this tunnel.
- o Decompose the service instance into subservices is what the assurance tree depicted in Figure 2 does.

In order for SAIN to be applied, the configuration necessary for each service instance should be identifiable and thus should come from a "service-aware" source. While the figure 1 makes a distinction between the SAIN orchestrator and a different component providing the service instance configuration, in practice those two components are mostly likely combined. The internals of the orchestrator are currently out of scope of this standardization.

3.3. Subservices

A subservice corresponds to subpart or a feature of the network system that is needed for a service instance to function properly. In the context of SAIN, subservice is actually a shortcut for subservice assurance, that is the method for assuring that a subservice behaves correctly.

A subservice is characterized by a list of metrics to fetch and a list of computations to apply to these metrics in order to produce a health status. Subservices, as services, have high-level parameters which defines which object should be assured.

3.4. Building the Expression Tree from the Assurance Tree

From the assurance tree is derived a so-called expression tree, which is actually a DAG whose sources are constants or metrics and other nodes are operators. The expression tree encodes all the operations needed to produce heath statuses from the collected metrics.

Subservices shall be device independent. To justify this, let's consider the interface operational status. Dependending on the device capabilities, this status can be collected by an industry-accepted YANG module (IETF, Openconfig), by a vendor-specific YANG module, or even by a MIB module. If the subservice was dependent on the mechanism to collect the operational status, then we would need multiple subservice definitions in order to support all different mechanisms.

In order to keep subservices independent from metric collection method, or, expressed differently, to support multiple combinations of platforms, OSes, and even vendors, the framework introduces the concept of "metric engine". The metric engine maps each device-independent metric used in the subservices to a list of device-specific metric implementations that precisely define how to fetch values for that metric. The mapping is parameterized by the characteristics (model, OS version, etc.) of the device from which the metrics are fetched.

3.5. Building the Expression from a Subservice

Additionally, to the list of metrics, each subservice defines a list of expressions to apply on the metrics in order to compute the health status of the subservice. The definition or the standardization of those expressions (also known as heuristic) is currently out of scope of this standardization.

3.6. Open Interfaces with YANG Modules

The interfaces between the architecture components are open thanks to YANG module(I-D.claise-opsawg-service-assurance-yang) defines objects for assuring network services based on their decomposition into so-called subservices, according to the SAIN architecture.

This module is intended for the following use cases:

o Assurance tree configuration:

  * Subservices: configure a set of subservices to assure, by specifying their types and parameters.

  * Dependencies: configure the dependencies between the subservices, along with their type.

o Assurance telemetry: export the health status of the subservices, along with the observed symptoms.

4. Security Considerations

TO BE COMPLETED

5. IANA Considerations

This document includes no request to IANA.

6. Open Issues

-Security Considerations to be completed

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

7.2. Informative References

[I-D.ietf-opsawg-tacacs] Dahm, T., Ota, A., dcmgash@cisco.com, d., Carrel, D., and L. Grant, "The TACACS+ Protocol", draft-ietf-opsawg-tacacs-15 (work in progress), September 2019.

[RFC2138] Rigney, C., Rubens, A., Simpson, W., and S. Willens, "Remote Authentication Dial In User Service (RADIUS)", RFC 2138, DOI 10.17487/RFC2138, April 1997, <https://www.rfc-editor.org/info/rfc2138>.

[RFC3164] Lonvick, C., "The BSD Syslog Protocol", RFC 3164, DOI 10.17487/RFC3164, August 2001, <https://www.rfc-editor.org/info/rfc3164>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC7011] Claise, B., Ed., Trammell, B., Ed., and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information", STD 77, RFC 7011, DOI 10.17487/RFC7011, September 2013, <https://www.rfc-editor.org/info/rfc7011>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <https://www.rfc-editor.org/info/rfc8040>.

[RFC8199] Bogdanovic, D., Claise, B., and C. Moberg, "YANG Module Classification", RFC 8199, DOI 10.17487/RFC8199, July 2017, <https://www.rfc-editor.org/info/rfc8199>.

[RFC8641] Clemm, A. and E. Voit, "Subscription to YANG Notifications for Datastore Updates", RFC 8641, DOI 10.17487/RFC8641, September 2019, <https://www.rfc-editor.org/info/rfc8641>.

Appendix A. Changes between revisions

v00 - v01

- o Placeholder for next version.

Acknowledgements

The authors would like to thank ...

Authors' Addresses

Benoit Claise
Cisco Systems, Inc.
De Kleetlaan 6a b1
1831 Diegem
Belgium

Email: bclaise@cisco.com

Jean Quilbeuf
Cisco Systems, Inc.
1, rue Camille Desmoulins
92782 Issy Les Moulineaux
France

Email: jquilbeu@cisco.com