Internet Engineering Task Force (IETF)
Request for Comments: 9417
Category: Informational
ISSN: 2070-1721

B. Claise
J. Quilbeuf
Huawei
D. Lopez
Telefonica I+D
D. Voyer
Bell Canada
T. Arumugam
Consultant
July 2023

# Service Assurance for Intent-Based Networking Architecture

## Abstract

This document describes an architecture that provides some assurance that service instances are running as expected. As services rely upon multiple subservices provided by a variety of elements, including the underlying network devices and functions, getting the assurance of a healthy service is only possible with a holistic view of all involved elements. This architecture not only helps to correlate the service degradation with symptoms of a specific network component but, it also lists the services impacted by the failure or degradation of a specific network component.

## Status of This Memo



## Copyright Notice



## Table of Contents

# 1. 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.

Service orchestrators use Network Service YANG Modules that will infer network-wide configuration and, therefore, the invocation of the appropriate device modules (Section 3 of [RFC8969]). Knowing that a configuration is applied doesn't imply that the provisioned service instance is up and running as expected. For instance, the service might be degraded because of a failure in the network, the service quality may be degraded, or a service function may be reachable at the IP level but does not provide its intended function. Thus, the network operator must monitor the service's operational data at the same time as the configuration (Section 3.3 of [RFC8969]). To fuel that task, the industry has been standardizing on telemetry to push network element performance information (e.g., [RFC9375]).

A network administrator needs to monitor its network and services as a whole, independently of 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 management protocols, the network management entities have to perform the difficult and time-consuming job of mapping data models, e.g., the model used for configuration with the model used for monitoring when separate models or protocols are used. This problem is compounded by a large, disparate set of data sources (e.g., MIB modules, YANG data models [RFC7950], IP Flow Information Export (IPFIX) information elements [RFC7011], syslog plain text [RFC5424], Terminal Access Controller Access-Control System Plus (TACACS+) [RFC8907], RADIUS [RFC2865], 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 data models used for configuration. Model-driven telemetry greatly facilitates the notion of closed-loop automation, whereby events and updated operational states streamed from the network drive remediation change 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 layer 3 virtual private network (L3VPN) fail to connect?" or "Why is this specific service not highly responsive?" The reverse, i.e., which services are impacted when a network component fails or degrades, is also important for operators, for example, "Which services are impacted when this specific optic decibel milliwatt (dBm) begins to degrade?", "Which applications are impacted by an imbalance in this Equal-Cost Multipath (ECMP) bundle?", or "Is that issue actually impacting any other customers?" This task usually falls under the so-called "Service Impact Analysis" functional block.

This document defines an architecture implementing Service Assurance for Intent-based Networking (SAIN). Intent-based approaches are often declarative, starting from a statement of "The service works as expected" and trying to enforce it. However, some already-defined services might have been designed using a different approach. Aligned with Section 3.3 of [RFC7149], and instead of requiring a declarative intent as a starting point, this architecture focuses on already-defined services and tries to infer the meaning of "The service works as expected". To do so, the architecture works from an assurance graph, deduced from the configuration pushed to the device for enabling the service instance. If the SAIN orchestrator supports it, the service model (Section 2 of [RFC8309]) or the network model (Section 2.1 of [RFC8969]) can also be used to build the assurance graph. In that case and if the service model includes the declarative intent as well, the SAIN orchestrator can rely on the declared intent instead of inferring it. The assurance graph may also be explicitly completed to add an intent not exposed in the service model itself.

The assurance graph of a service instance is decomposed into components, which are then assured independently. The top of the assurance graph represents the service instance to assure, and its children represent components identified as its direct dependencies; each component can have dependencies as well. Components involved in the assurance graph of a service are called subservices. The SAIN orchestrator updates the assurance graph automatically when the service instance is modified.

When a service is degraded, the SAIN architecture will highlight where in the assurance service graph to look, as opposed to going hop by hop to troubleshoot the issue. More precisely, the SAIN architecture will associate to each service instance a list of symptoms originating from specific subservices, corresponding to components of the network. These components are good candidates for explaining the source of a service degradation. Not only can this architecture help to correlate service degradation with network root cause/symptoms, but it can deduce from the assurance graph the list of service instances impacted by a component degradation/failure. This added value informs the operational team where to focus its attention for maximum return. Indeed, the operational team is likely to focus their priority on the degrading/failing components impacting the highest number of their customers, especially the ones with the Service-Level Agreement (SLA) contracts involving penalties in case of failure.

This architecture provides the building blocks to assure both physical and virtual entities and is flexible with respect to services and subservices of (distributed) graphs and components (Section 3.7).

The architecture presented in this document is implemented by a set of YANG modules defined in a companion document [RFC9418]. These YANG modules properly define the interfaces between the various components of the architecture to foster interoperability.

## 2. Terminology

SAIN agent: A functional component that communicates with a device, a set of devices, or another agent to build an expression graph from a received assurance graph and perform the corresponding computation of the health status and symptoms. A SAIN agent might be running directly on the device it monitors.

Assurance case: "An assurance case is a structured argument, supported by evidence, intended to justify that a system is acceptably assured relative to a concern (such as safety or security) in the intended operating environment" [Piovesan2017].

Service instance: A specific instance of a service.

Intent: "A set of operational goals (that a network should meet) and outcomes (that a network is supposed to deliver) defined in a declarative manner without specifying how to achieve or implement them" [RFC9315].

Subservice: A part or functionality of the network system that can be independently assured as a single entity in an assurance graph.

Assurance graph: A Directed Acyclic Graph (DAG) representing the assurance case for one or several service instances. The nodes (also known as vertices in the context of DAG) are the service instances themselves and the subservices; the edges indicate a dependency relation.

SAIN collector: A functional component that fetches or receives the computer-consumable output of the SAIN agent(s) and processes it locally (including displaying it in a user-friendly form).

DAG: Directed Acyclic Graph.

ECMP: Equal-Cost Multipath.

Expression graph: A generic term for a DAG representing a computation in SAIN. More specific terms are listed below:

Subservice expressions:
An expression graph representing all the computations to execute for a subservice.

Service expressions:
An expression graph representing all the computations to execute for a service instance, i.e., including the computations for all dependent subservices.

Global computation graph:
An expression graph representing all the computations to execute for all services instances (i.e., all computations performed).

Dependency: The directed relationship between subservice instances in the assurance graph.

Metric: A piece of information retrieved from the network running the assured service.

Metric engine: A functional component, part of the SAIN agent, that maps metrics to a list of candidate metric implementations, depending on the network element.

Metric implementation: The actual way of retrieving a metric from a network element.

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

Service orchestrator: "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" [RFC8199].

SAIN orchestrator: A functional component that is in charge of fetching the configuration specific to each service instance and converting it into an assurance graph.

Health status: The score and symptoms indicating whether a service instance or a subservice is "healthy". A non-maximal score must always be explained by one or more symptoms.

Health score: An integer ranging from 0 to 100 that indicates the health of a subservice. A score of 0 means that the subservice is broken, a score of 100 means that the subservice in question is operating as expected, and the special value -1 can be used to specify that no value could be computed for that health score, for instance, if some metric needed for that computation could not be collected.

Strongly connected component: A subset of a directed graph such that there is a (directed) path from any node of the subset to any other node. A DAG does not contain any strongly connected component.

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

## 3. A Functional Architecture

The goal of SAIN is to assure that service instances are operating as expected (i.e., the observed service is matching the expected service) and, if not, to pinpoint what is wrong. More precisely, SAIN computes a score for each service instance and outputs symptoms explaining that score. The only valid situation where no symptoms are returned is when the score is maximal, indicating that no issues were detected for that service instance. The score augmented with the symptoms is called the health status. The exact meaning of the health score value is out of scope of this document. However, the following constraints should be followed: the higher the score, the better the service health is and the two extrema are 0 meaning the service is completely broken, and 100 meaning the service is completely operational.

The SAIN architecture is a generic architecture, which generates an assurance graph from service instance(s), as specified in Section 3.1. This architecture is applicable to not only multiple environments (e.g., wireline and wireless) but also different domains (e.g., 5G network function virtualization (NFV) domain with a virtual infrastructure manager (VIM), etc.) and, as already noted, for physical or virtual devices, as well as virtual functions. Thanks to the distributed graph design principle, graphs from different environments and orchestrators can be combined to obtain the graph of a service instance that spans over multiple domains.

As an example of a service, let us consider a point-to-point layer 2 virtual private network (L2VPN). [RFC8466] specifies the parameters for such a service. Examples of symptoms might be symptoms reported by specific subservices, including "Interface has high error rate", "Interface flapping", or "Device almost out of memory", as well as symptoms more specific to the service (such as "Site disconnected from VPN").

To compute the health status of an instance of such a service, the service definition is decomposed into an assurance graph formed by subservices linked through dependencies.  Each subservice is then turned into an expression graph that details how to fetch metrics from the devices and compute the health status of the subservice. The subservice expressions are combined according to the dependencies between the subservices in order to obtain the expression graph that computes the health status of the service instance.

The overall SAIN architecture is presented in Figure 1.  Based on the service configuration provided by the service orchestrator, the SAIN orchestrator decomposes the assurance graph.  It then sends to the SAIN agents the assurance graph along with some other configuration options.  The SAIN agents are responsible for building the expression graph and computing the health statuses in a distributed manner.  The collector is in charge of collecting and displaying the current inferred health status of the service instances and subservices.  The collector also detects changes in the assurance graph structures (e.g., an occurrence of a switchover from primary to backup path) and forwards the information to the orchestrator, which reconfigures the agents.  Finally, the automation loop is closed by having the SAIN collector provide feedback to the network/service orchestrator.

In order to make agents, orchestrators, and collectors from different vendors interoperable, their interface is defined as a YANG module in a companion document [RFC9418].  In Figure 1, the communications that are normalized by this YANG module are tagged with a "Y".  The use of this YANG module is further explained in Section 3.5.

```
     +-----------------+
     | Service         |
     | Orchestrator    |<----------------------+
     |                 |                       |
     +-----------------+                       |
        |            ^                         |
        |            | Network                 |
        |            | Service                 | Feedback
        |            | Instance                | Loop
        |            | Configuration           |
        |            |                         |
        |            V                         |
        |        +-----------------+  Graph  +-------------------+
        |        | SAIN            | Updates | SAIN              |
        |        | Orchestrator    |<--------| Collector         |
        |        +-----------------+         +-------------------+
        |            |                          ^
        |           Y| Configuration            | Health Status
        |            | (Assurance Graph)       Y| (Score + Symptoms)
        |            V                          | Streamed
        |     +-------------------+             | via Telemetry
        |     |+-------------------+            |
        |     ||+-------------------+           |
        |     +|| SAIN              |-----------+
        |      +| Agent             |
        |       +-------------------+
        |               ^ ^ ^
        |               | | |
        |               | | |  Metric Collection
        V               V V V
    +-------------------------------------------------------------+
    |           (Network) System                                  |
```

```
    |                                                              |
    +--------------------------------------------------------------+
```

Figure 1: SAIN Architecture

In order to produce the score assigned to a service instance, the various involved components perform the following tasks:

* Analyze the configuration pushed to the network device(s) for configuring the service instance. From there, determine which information (called a metric) must be collected from the device(s) and which operations to apply to the metrics to compute the health status.

* Stream (via telemetry, such as YANG-Push [RFC8641]) operational and config metric values when possible, else continuously poll.

* Continuously compute the health status of the service instances based on the metric values.

The SAIN architecture requires time synchronization, with the Network Time Protocol (NTP) [RFC5905] as a candidate, between all elements: monitored entities, SAIN agents, service orchestrator, the SAIN collector, as well as the SAIN orchestrator. This guarantees the correlations of all symptoms in the system, correlated with the right assurance graph version.

### 3.1. Translating a Service Instance Configuration into an Assurance Graph

In order to structure the assurance of a service instance, the SAIN orchestrator decomposes the service instance into so-called subservice instances. Each subservice instance focuses on a specific feature or subpart of the service.

The decomposition into subservices is an important function of the architecture for the following reasons:

* The result of this decomposition provides a relational picture of a service instance, which can be represented as a graph (called an assurance graph) to the operator.

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

* Subservices that are common to several service instances are reused for reducing the amount of computation needed. For instance, the subservice assuring a given interface is reused by any service instance relying on that interface.

The assurance graph 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, i.e., 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 graph 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 (the tunnel interfaces created on the peer1 and peer2 devices, respectively), 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 |
                            +------------------+
                                      |
                 +--------------------+-------------------+
                 |                    |                   |
                 v                    v                   v
            +-------------+    +--------------+    +-------------+
            | Peer1       |    | IP           |    | Peer2       |
            | Tunnel      |    | Connectivity |    | Tunnel      |
            | Interface   |    |              |    | Interface   |
            +-------------+    +--------------+    +-------------+
                   |                  |                  |
                   |    +-------------+--------------+   |
                   |    |             |              |   |
                   v    v             v              v   v
            +-------------+    +-------------+     +-------------+
            | Peer1       |    | IS-IS       |     | Peer2       |
            | Physical    |    | Routing     |     | Physical    |
            | Interface   |    | Protocol    |     | Interface   |
            +-------------+    +-------------+     +-------------+
                   |                                     |
                   v                                     v
            +-------------+                        +-------------+
            |             |                        |             |
            | Peer1       |                        | Peer2       |
            | Device      |                        | Device      |
            +-------------+                        +-------------+
```

Figure 2: Assurance Graph Example

Depicting the assurance graph helps the operator to understand (and assert) the decomposition. The assurance graph shall be maintained during normal operation with addition, modification, and removal of service instances. A change in the network configuration or topology shall automatically be reflected in the assurance graph. As a first example, a change of the routing protocol from IS-IS to OSPF would change the assurance graph accordingly. As a second example, assume that the ECMP is in place for the source router for that specific tunnel; in that case, multiple interfaces must now be monitored, in addition to monitoring the ECMP health itself.

### 3.1.1. Circular Dependencies

The edges of the assurance graph represent dependencies. An assurance graph is a DAG if and only if there are no circular dependencies among the subservices, and every assurance graph should avoid circular dependencies. However, in some cases, circular dependencies might appear in the assurance graph.

First, the assurance graph of a whole system is obtained by combining the assurance graph of every service running on that system. Here, combining means that two subservices having the same type and the same parameters are in fact the same subservice and thus a single node in the graph. For instance, the subservice of type "device" with the only parameter (the device ID) set to "PE1" will appear only once in the whole assurance graph, even if several service instances rely on that device. Now, if two engineers design assurance graphs for two different services, and Engineer A decides that an interface depends on the link it is connected to, but Engineer B decides that the link depends on the interface it is connected to, then when combining the two assurance graphs, we will have a circular dependency interface -> link -> interface.

Another case possibly resulting in circular dependencies is when subservices are not properly identified. Assume that we want to assure a cloud-based computing cluster that runs containers. We could represent the cluster by a subservice and the network service connecting containers on the cluster by another subservice. We would likely model that as the network service depending on the cluster, because the network service runs in a container supported by the cluster. Conversely, the cluster depends on the network service for connectivity between containers, which creates a circular dependency. A finer decomposition might distinguish between the resources for executing containers (a part of our cluster subservice) and the communication between the containers (which could be modeled in the same way as communication between routers).

In any case, it is likely that circular dependencies will show up in the assurance graph. A first step would be to detect circular dependencies as soon as possible in the SAIN architecture. Such a detection could be carried out by the SAIN orchestrator. Whenever a circular dependency is detected, the newly added service would not be monitored until more careful modeling or alignment between the different teams (Engineers A and B) remove the circular dependency.

As a more elaborate solution, we could consider a graph transformation:

* Decompose the graph into strongly connected components.

* For each strongly connected component:

  - remove all edges between nodes of the strongly connected component;

  - add a new "synthetic" node for the strongly connected component;

  - for each edge pointing to a node in the strongly connected component, change the destination to the "synthetic" node; and

  - add a dependency from the "synthetic" node to every node in the strongly connected component.

Such an algorithm would include all symptoms detected by any subservice in one of the strongly connected components and make it available to any subservice that depends on it. Figure 3 shows an example of such a transformation. On the left-hand side, the nodes c, d, e, and f form a strongly connected component. The status of node a should depend on the status of nodes c, d, e, f, g, and h, but this is hard to compute because of the circular dependency. On the right-hand side, node a depends on all these nodes as well, but the circular dependency has been removed.

```
      +---+    +---+          |                +---+    +---+
      | a |    | b |          |                | a |    | b |
      +---+    +---+          |                +---+    +---+
        |        |            |                  |        |
        v        v            |                  v        v
      +---+    +---+          |                +------------+
      | c |--->| d |          |                |  synthetic |
      +---+    +---+          |                +------------+
        ^        |            |               /   |      |   \
        |        |            |              /    |      |    \
        |        v            |             v     v      v     v
      +---+    +---+          |          +---+  +---+  +---+  +---+
```

```
      | f |<---| e |          |         | f |  | c |  | d |  | e |
      +---+    +---+          |         +---+  +---+  +---+  +---+
        |        |            |           |                    |
        v        v            |           v                    v
      +---+    +---+          |         +---+                +---+
      | g |    | h |          |         | g |                | h |
      +---+    +---+          |         +---+                +---+

         Before                                   After
      Transformation                         Transformation
```

Figure 3: Graph Transformation

We consider a concrete example to illustrate this transformation. Let’s assume that Engineer A is building an assurance graph dealing with IS-IS and Engineer B is building an assurance graph dealing with OSPF. The graph from Engineer A could contain the following:

```
                +------------+
                | IS-IS Link |
                +------------+
                      |
                      v
                +------------+
                | Phys. Link |
                +------------+
                  |       |
                  v       v
       +-------------+  +-------------+
       | Interface 1 |  | Interface 2 |
       +-------------+  +-------------+
```

Figure 4: Fragment of the Assurance Graph from Engineer A

The graph from Engineer B could contain the following:

```
                +------------+
                | OSPF Link  |
                +------------+
                  |   |   |
                  v   |   v
     +-------------+  |  +-------------+
     | Interface 1 |  |  | Interface 2 |
     +-------------+  |  +-------------+
                   |  |   |
                   v  v   v
                +------------+
                | Phys. Link |
                +------------+
```

Figure 5: Fragment of the Assurance Graph from Engineer B

The Interface subservices and the Physical Link subservice are common to both fragments above. Each of these subservices appear only once in the graph merging the two fragments. Dependencies from both fragments are included in the merged graph, resulting in a circular dependency:

```
      +------------+      +------------+
      | IS-IS Link |      | OSPF Link  |---+
      +------------+      +------------+   |
            |               |     |        |
            |     +-------- +     |        |
            v     v               |        |
      +------------+              |        |
      | Phys. Link |<-------+     |        |
```

```
         +------------+        |     |        |
           |  ^     |          |     |        |
           |  |     +-------+  |     |        |
           v  |             v  |     v        |
         +-------------+  +-------------+     |
         | Interface 1 |  | Interface 2 |     |
         +-------------+  +-------------+     |
               ^                              |
               |                              |
               +------------------------------+
```

Figure 6: Merging Graphs from Engineers A and B

The solution presented above would result in a graph looking as follows, where a new "synthetic" node is included. Using that transformation, all dependencies are indirectly satisfied for the nodes outside the circular dependency, in the sense that both IS-IS and OSPF links have indirect dependencies to the two interfaces and the link. However, the dependencies between the link and the interfaces are lost since they were causing the circular dependency.

```
               +------------+      +------------+
               | IS-IS Link |      | OSPF Link  |
               +------------+      +------------+
                          |          |
                          v          v
                         +------------+
                         |  synthetic |
                         +------------+
                               |
                   +-----------+-------------+
                   |           |             |
                   v           v             v
         +-------------+ +------------+ +-------------+
         | Interface 1 | | Phys. Link | | Interface 2 |
         +-------------+ +------------+ +-------------+
```

Figure 7: Removing Circular Dependencies after Merging Graphs from Engineers A and B

## 3.2. Intent and Assurance Graph

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

* Try to capture the intent of the service instance, i.e., What is the service instance trying to achieve? At a minimum, this requires the SAIN orchestrator to know the YANG modules that are being configured on the devices to enable the service. Note that, if the service model or the network model is known to the SAIN orchestrator, the latter can exploit it. In that case, the intent could be directly extracted and include more details, such as the notion of sites for a VPN, which is out of scope of the device configuration.

* Decompose the service instance into subservices representing the network features on which the service instance relies.

The SAIN orchestrator must be able to analyze the configuration pushed to various devices of a service instance and produce the assurance graph for that service instance.

To schematize what a SAIN orchestrator does, assume that a service instance touches two devices and configures a virtual tunnel interface on each device. Then:

* 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 operational. This solution is minimally invasive, as it does not require modifying nor knowing the service model. If the service model or network model is known by the SAIN orchestrator, it can be used to further capture the intent and include more information, such as Service-Level Objectives (e.g., the latency and bandwidth requirements for the tunnel) if present in the service model.

* Decomposing the service instance into subservices would result in the assurance graph depicted in Figure 2, for instance.

The assurance graph, or more precisely the subservices and dependencies that a SAIN orchestrator can instantiate, should be curated. The organization of such a process (i.e., ensure that existing subservices are reused as much as possible and avoid circular dependencies) is out-of-scope for this document.

To be applied, SAIN requires a mechanism mapping a service instance to the configuration actually required on the devices for that service instance to run. While Figure 1 makes a distinction between the SAIN orchestrator and a different component providing the service instance configuration, in practice those two components are most likely combined. The internals of the orchestrator are out of scope of this document.

### 3.3. Subservices

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

Subservices, just as with services, have high-level parameters that specify the instance to be assured. The needed parameters depend on the subservice type. For example, assuring a device requires a specific deviceId as a parameter and assuring an interface requires a specific combination of deviceId and interfaceId.

When designing a new type of subservice, one should carefully define what is the assured object or functionality. Then, the parameters must be chosen as a minimal set that completely identifies the object (see examples from the previous paragraph). Parameters cannot change during the life cycle of a subservice. For instance, an IP address is a good parameter when assuring a connectivity towards that address (i.e., a given device can reach a given IP address); however, it’s not a good parameter to identify an interface, as the IP address assigned to that interface can be changed.

A subservice is also characterized by a list of metrics to fetch and a list of operations to apply to these metrics in order to infer a health status.

### 3.4. Building the Expression Graph from the Assurance Graph

From the assurance graph, a so-called global computation graph is derived. First, each subservice instance is transformed into a set of subservice expressions that take metrics and constants as input (i.e., sources of the DAG) and produce the status of the subservice based on some heuristics. For instance, the health of an interface is 0 (minimal score) with the symptom "interface admin-down" if the interface is disabled in the configuration. Then, for each service instance, the service expressions are constructed by combining the subservice expressions of its dependencies. The way service expressions are combined depends on the dependency types (impacting or informational). Finally, the global computation graph is built by combining the service expressions to get a global view of all subservices. In other words, the global computation graph encodes all the operations needed to produce health statuses from the collected metrics.

The two types of dependencies for combining subservices are:

Informational Dependency:
: The type of dependency whose health score does not impact the health score of its parent subservice or service instance(s) in the assurance graph. However, the symptoms should be taken into account in the parent service instance or subservice instance(s) for informational reasons.

Impacting Dependency:
: The type of dependency whose health score impacts the health score of its parent subservice or service instance(s) in the assurance graph. The symptoms are taken into account in the parent service instance or subservice instance(s) as the impacting reasons.

The set of dependency types presented here is not exhaustive. More specific dependency types can be defined by extending the YANG module. For instance, a connectivity subservice depending on several path subservices is partially impacted if only one of these paths fails. Adding these new dependency types requires defining the corresponding operation for combining statuses of subservices.

Subservices shall not be dependent on the protocol used to retrieve the metrics. To justify this, let’s consider the interface operational status. Depending on the device capabilities, this status can be collected by an industry-accepted YANG module (e.g., IETF or Openconfig [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. This also implies that, while waiting for all the metrics to be available via standard YANG modules, SAIN agents might have to retrieve metric values via nonstandard YANG data models, MIB modules, the Command-Line Interface (CLI), etc., effectively implementing a normalization layer between data models and information models.

In order to keep subservices independent of metric collection method (or, expressed differently, to support multiple combinations of platforms, OSes, and even vendors), the architecture 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 (i.e., model, OS version, etc.) of the device from which the metrics are fetched. This metric engine is included in the SAIN agent.

### 3.5. Open Interfaces with YANG Modules

The interfaces between the architecture components are open thanks to the YANG modules specified in [RFC9418]; they specify objects for assuring network services based on their decomposition into so-called subservices, according to the SAIN architecture.

These modules are intended for the following use cases:

* Assurance graph 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 types.

* Assurance telemetry: Export the health status of the subservices, along with the observed symptoms.

Some examples of YANG instances can be found in Appendix A of [RFC9418].

## 3.6. Handling Maintenance Windows

Whenever network components are under maintenance, the operator wants to inhibit the emission of symptoms from those components. A typical use case is device maintenance, during which the device is not supposed to be operational. As such, symptoms related to the device health should be ignored. Symptoms related to the device-specific subservices, such as the interfaces, might also be ignored because their state changes are probably the consequence of the maintenance.

The ietf-service-assurance model described in [RFC9418] enables flagging subservices as under maintenance and, in that case, requires a string that identifies the person or process that requested the maintenance. When a service or subservice is flagged as under maintenance, it must report a generic "Under Maintenance" symptom for propagation towards subservices that depend on this specific subservice. Any other symptom from this service or by one of its impacting dependencies must not be reported.

We illustrate this mechanism on three independent examples based on the assurance graph depicted in Figure 2:

* Device maintenance, for instance, upgrading the device OS. The operator flags the subservice "Peer1" device as under maintenance. This inhibits the emission of symptoms, except "Under Maintenance" from "Peer1 Physical Interface", "Peer1 Tunnel Interface", and "Tunnel Service Instance". All other subservices are unaffected.

* Interface maintenance, for instance, replacing a broken optic. The operator flags the subservice "Peer1 Physical Interface" as under maintenance. This inhibits the emission of symptoms, except "Under Maintenance" from "Peer 1 Tunnel Interface" and "Tunnel Service Instance". All other subservices are unaffected.

* Routing protocol maintenance, for instance, modifying parameters or redistribution. The operator marks the subservice "IS-IS Routing Protocol" as under maintenance. This inhibits the emission of symptoms, except "Under Maintenance" from "IP connectivity" and "Tunnel Service Instance". All other subservices are unaffected.

In each example above, the subservice under maintenance is completely impacting the service instance, putting it under maintenance as well. There are use cases where the subservice under maintenance only partially impacts the service instance. For instance, consider a service instance supported by both a primary and backup path. If a subservice impacting the primary path is under maintenance, the service instance might still be functional but degraded. In that case, the status of the service instance might include "Primary path Under Maintenance", "No redundancy", as well as other symptoms from the backup path to explain the lower health score. In general, the computation of the service instance status from the subservices is done in the SAIN collector whose implementation is out of scope for this document.

The maintenance of a subservice might modify or hide modifications of the structure of the assurance graph. Therefore, unflagging a subservice as under maintenance should trigger an update of the assurance graph.

## 3.7. Flexible Functional Architecture

The SAIN architecture is flexible in terms of components. While the SAIN architecture in Figure 1 makes a distinction between two components, the service orchestrator and the SAIN orchestrator, in practice the two components are most likely combined. Similarly, the SAIN agents are displayed in Figure 1 as being separate components. In practice, the SAIN agents could be either independent components or directly integrated in monitored entities. A practical example is an agent in a router.

The SAIN architecture is also flexible in terms of services and subservices. In the defined architecture, the SAIN orchestrator is coupled to a service orchestrator, which defines the kinds of services that the architecture handles. Most examples in this document deal with the notion of Network Service YANG Modules with well-known services, such as L2VPN or tunnels. However, the concept of services is general enough to cross into different domains. One of them is the domain of service management on network elements, which also require their own assurance. Examples include a DHCP server on a Linux server, a data plane, an IPFIX export, etc. The notion of "service" is generic in this architecture and depends on the service orchestrator and underlying network system, as illustrated by the following examples:

* If a main service orchestrator coordinates several lower-level controllers, a service for the controller can be a subservice from the point of view of the orchestrator.

* A DHCP server / data plane / IPFIX export can be considered subservices for a device.

* A routing instance can be considered a subservice for an L3VPN.

* A tunnel can be considered a subservice for an application in the cloud.

* A service function can be considered a subservice for a service function chain [RFC7665].

The assurance graph is created to be flexible and open, regardless of the subservice types, locations, or domains.

The SAIN architecture is also flexible in terms of distributed graphs. As shown in Figure 1, the architecture comprises several agents. Each agent is responsible for handling a subgraph of the assurance graph. The collector is responsible for fetching the subgraphs from the different agents and gluing them together. As an example, in the graph from Figure 2, the subservices relative to Peer 1 might be handled by a different agent than the subservices relative to Peer 2, and the Connectivity and IS-IS subservices might be handled by yet another agent. The agents will export their partial graph, and the collector will stitch them together as dependencies of the service instance.

And finally, the SAIN architecture is flexible in terms of what it monitors. Most, if not all, examples in this document refer to physical components, but this is not a constraint. Indeed, the assurance of virtual components would follow the same principles, and an assurance graph composed of virtualized components (or a mix of virtualized and physical ones) is supported by this architecture.

### 3.8. Time Window for Symptoms' History

The health status reported via the YANG modules contains, for each subservice, the list of symptoms. Symptoms have a start and end date, making it is possible to report symptoms that are no longer occurring.

The SAIN agent might have to remove some symptoms for specific subservice symptoms because they are outdated and no longer relevant or simply because the SAIN agent needs to free up some space. Regardless of the reason, it's important for a SAIN collector connecting/reconnecting to a SAIN agent to understand the effect of this garbage collection.

Therefore, the SAIN agent contains a YANG object specifying the date and time at which the symptoms' history starts for the subservice instances. The subservice reports only symptoms that are occurring or that have been occurring after the history start date.

### 3.9. New Assurance Graph Generation

The assurance graph will change over time, because services and subservices come and go (changing the dependencies between subservices) or as a result of resolving maintenance issues. Therefore, an assurance graph version must be maintained, along with the date and time of its last generation. The date and time of a particular subservice instance (again dependencies or under maintenance) might be kept. From a client point of view, an assurance graph change is triggered by the value of the assurance-graph-version and assurance-graph-last-change YANG leaves. At that point in time, the client (collector) follows the following process:

* Keep the previous assurance-graph-last-change value (let's call it time T).

* Run through all the subservice instances and process the subservice instances for which the last-change is newer than the time T.

* Keep the new assurance-graph-last-change as the new referenced date and time.

## 4. IANA Considerations

This document has no IANA actions.

## 5. Security Considerations

The SAIN architecture helps operators to reduce the mean time to detect and the mean time to repair. However, the SAIN agents must be secured; a compromised SAIN agent may be sending incorrect root causes or symptoms to the management systems. Securing the agents falls back to ensuring the integrity and confidentiality of the assurance graph. This can be partially achieved by correctly setting permissions of each node in the YANG data model, as described in Section 6 of [RFC9418].

Except for the configuration of telemetry, the agents do not need "write access" to the devices they monitor. This configuration is applied with a YANG module, whose protection is covered by Secure Shell (SSH) [RFC6242] for the Network Configuration Protocol (NETCONF) or TLS [RFC8446] for RESTCONF. Devices should be configured so that agents have their own credentials with write access only for the YANG nodes configuring the telemetry.

The data collected by SAIN could potentially be compromising to the network or provide more insight into how the network is designed. Considering the data that SAIN requires (including CLI access in some cases), one should weigh data access concerns with the impact that reduced visibility will have on being able to rapidly identify root causes.

For building the assurance graph, the SAIN orchestrator needs to obtain the configuration from the service orchestrator. The latter should restrict access of the SAIN orchestrator to information needed to build the assurance graph.

If a closed loop system relies on this architecture, then the well-known issue of those systems also applies, i.e., a lying device or compromised agent could trigger partial reconfiguration of the service or network. The SAIN architecture neither augments nor reduces this risk. An extension of SAIN, which is out of scope for this document, could detect discrepancies between symptoms reported by different agents, and thus detect anomalies if an agent or a device is lying.

If NTP service goes down, the devices clocks might lose their synchronization. In that case, correlating information from different devices, such as detecting symptoms about a link or correlating symptoms from different devices, will give inaccurate results.

# 6. References

## 6.1. Normative References


[RFC8309] Wu, Q., Liu, W., and A. Farrel, "Service Models Explained", RFC 8309, DOI 10.17487/RFC8309, January 2018, <https://www.rfc-editor.org/info/rfc8309>.

[RFC8969] Wu, Q., Ed., Boucadair, M., Ed., Lopez, D., Xie, C., and L. Geng, "A Framework for Automating Service and Network Management with YANG", RFC 8969, DOI 10.17487/RFC8969, January 2021, <https://www.rfc-editor.org/info/rfc8969>.

[RFC9418] Claise, B., Quilbeuf, J., Lucente, P., Fasano, P., and T. Arumugam, "A YANG Data Model for Service Assurance", RFC 9418, DOI 10.17487/RFC9418, July 2023, <https://www.rfc-editor.org/info/rfc9418>.


## 6.2. Informative References


[OpenConfig] "OpenConfig", <https://openconfig.net>.

[Piovesan2017] Piovesan, A. and E. Griffor, "7 - Reasoning About Safety and Security: The Logic of Assurance", DOI 10.1016/B978-0-12-803773-7.00007-3, 2017, <https://doi.org/10.1016/B978-0-12-803773-7.00007-3>.

[RFC2865] Rigney, C., Willens, S., Rubens, A., and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)", RFC 2865, DOI 10.17487/RFC2865, June 2000, <https://www.rfc-editor.org/info/rfc2865>.

[RFC5424] Gerhards, R., "The Syslog Protocol", RFC 5424, DOI 10.17487/RFC5424, March 2009, <https://www.rfc-editor.org/info/rfc5424>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC6242] Wasserman, M., "Using the NETCONF Protocol over Secure Shell (SSH)", RFC 6242, DOI 10.17487/RFC6242, June 2011, <https://www.rfc-editor.org/info/rfc6242>.

[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>.

[RFC7149] Boucadair, M. and C. Jacquenet, "Software-Defined Networking: A Perspective from within a Service Provider Environment", RFC 7149, DOI 10.17487/RFC7149, March 2014, <https://www.rfc-editor.org/info/rfc7149>.

[RFC7665] Halpern, J., Ed. and C. Pignataro, Ed., "Service Function Chaining (SFC) Architecture", RFC 7665, DOI 10.17487/RFC7665, October 2015, <https://www.rfc-editor.org/info/rfc7665>.

[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>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8466] Wen, B., Fioccola, G., Ed., Xie, C., and L. Jalil, "A YANG Data Model for Layer 2 Virtual Private Network (L2VPN) Service Delivery", RFC 8466, DOI 10.17487/RFC8466, October 2018, <https://www.rfc-editor.org/info/rfc8466>.

[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>.

[RFC8907] Dahm, T., Ota, A., Medway Gash, D.C., Carrel, D., and L. Grant, "The Terminal Access Controller Access-Control System Plus (TACACS+) Protocol", RFC 8907, DOI 10.17487/RFC8907, September 2020, <https://www.rfc-editor.org/info/rfc8907>.

[RFC9315] Clemm, A., Ciavaglia, L., Granville, L. Z., and J. Tantsura, "Intent-Based Networking - Concepts and Definitions", RFC 9315, DOI 10.17487/RFC9315, October 2022, <https://www.rfc-editor.org/info/rfc9315>.

[RFC9375] Wu, B., Ed., Wu, Q., Ed., Boucadair, M., Ed., Gonzalez de Dios, O., and B. Wen, "A YANG Data Model for Network and VPN Service Performance Monitoring", RFC 9375, DOI 10.17487/RFC9375, April 2023, <https://www.rfc-editor.org/info/rfc9375>.

## Acknowledgements

The authors would like to thank Stephane Litkowski, Charles Eckel, Rob Wilton, Vladimir Vassiliev, Gustavo Alburquerque, Stefan Vallin, ric Vyncke, Mohamed Boucadair, Dhruv Dhody, Michael Richardson, and Rob Wilton for their reviews and feedback.

Contributors

* Youssef El Fathi

* ric Vyncke

Authors’ Addresses

Benoit Claise
Huawei
Email: benoit.claise@huawei.com

Jean Quilbeuf
Huawei
Email: jean.quilbeuf@huawei.com

Diego R. Lopez
Telefonica I+D
Don Ramon de la Cruz, 82
28006 Madrid
Spain
Email: diego.r.lopez@telefonica.com

Dan Voyer
Bell Canada
Canada
Email: daniel.voyer@bell.ca

Thangavelu Arumugam
Consultant
Milpitas, California
United States of America
Email: thangavelu@yahoo.com