

NETMOD Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2020

M. Boucadair
Orange
Q. Wu
Z. Wang
Huawei
D. King
Lancaster University
C. Xie
China Telecom
November 3, 2019

Framework for Use of ECA (Event Condition Action) in Network Self
Management
draft-bwd-netmod-eca-framework-00

Abstract

Event-driven management is meant to provide a useful method to monitor state change of managed objects and resources, and facilitate automatic triggering of a response to events, based on an established set of rules. This would provide rapid autonomic responses to specific conditions, enabling self-management behaviors, including: self-configuration, self-healing, self-optimization, and self-protection.

This document provides a framework that describes the architecture for supporting event-driven management of managed object state across devices. It does not describe specific protocols or protocol extensions needed to realize the objectives and capabilities discussed in the document.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 6, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	4
2. Problem Statement	4
2.1. Defining Network Event and Network Control Logic	4
2.2. Delegating Network Control Logic to Network Device	4
2.3. Executing ECA Script in the Network Device	5
2.4. Event-Driven Notification Handling	6
2.5. Requisite State Information	6
3. Architectural Concepts	7
3.1. What is Defined in ECA Policy?	7
3.2. Where is ECA Script and State Held?	8
3.3. What State is Held?	9
4. Architecture Overview	9
4.1. Telemetry Automation in the Network Device	10
4.2. Detecting and Resolving Policy Conflict	12
4.3. Chain Reaction of Coordinated Events	12
5. Security Considerations	12
6. Acknowledgements	13
7. References	13
7.1. Normative References	13
7.2. Informative References	14
Authors' Addresses	15

1. Introduction

Network management data objects can often take two different values: the value configured by the administrator or an application (configuration) and the value that the device is actually using (operational state). Particularly, these network management data objects can be fetched from various different YANG datastore

[RFC8342] by subscribing to continuous datastore updates [RFC8641] without needing to poll for data periodically.

YANG-Push mechanisms are used to select which data objects are of interest using filters and provide frequent or prompt updates of remote object state, thus allowing (client) applications to maintain a continuous view of operational data and state and enabling a network operator to optimize the system behavior across the whole network to meet objectives and provide some performance guarantees for network services.

Network management may rely upon one or multiple policies to influence management behavior within the system and make sure policies are enforced or executed correctly so that there will no conflict in policies and that the observed behavior is the expected one. Event-driven policy (i.e., ECA Policy [RFC8328]) enables actions being automatically triggered based on when certain events in the network occur while certain conditions hold. YANG Push subscription provides a source for such events.

It is often the case that where Event Condition Action (ECA) is defined is decoupled from where ECA is executed. ECA Engine in the management system or the network device defines one or multiple events corresponding to the workflow management, correlate these events with action triggers and create ECA policy. ECA policy can be enforced either at the management system or pushed to and executed by the network device. Alternative, some of these predefined events can be translated into filter in the YANG push subscription which is in turn used to select data objects that are of interest. When these data objects are streamed out to the destination, both the management system and network device check for the condition when the event is observed. If the condition is satisfied, the ECA script is executed.

Event-driven management (of states of managed objects) across a wide range of devices can be used to monitor state changes of managed objects or resource and automatic trigger of rules in response to events so as to better service assurance for customers and to provide rapid autonomic response that can exhibit self-management properties including self-configuration, self-healing, self-optimization, and self-protection.

This document provides a framework that describes the architecture for supporting such event-driven management.

This document does not describe specific protocols or protocol extensions needed to realize the objectives and capabilities discussed in the document.

1.1. 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.

2. Problem Statement

2.1. Defining Network Event and Network Control Logic

Datastores are used by network management protocols such as NETCONF [RFC6241] and RESTCONF [RFC8040]. Operational state data objects, in the operational state datastore, provide network visibility to the actual state of the network, and ensure the network is running efficiently.

The network event are used to keep track of state of changes associated with one of multiple operational state data objects in the network device. Typical examples of network event include a fault, an alarm, a change in network state, network security threat, hardware malfunction, buffer utilization crossing a threshold, network connection setup, and an external input to the system.

To control which state a network device should be in or is allowed to be in at any given time, a set of conditions and actions are defined and correlated with network events, which constitute an event-driven policy or network control logic.

YANG Push subscription allows client applications to select which datastore nodes are of interest and provides source of network events. The NETCONF client can define event-based policy based on YANG Push subscription data source or some other data source.

2.2. Delegating Network Control Logic to Network Device

Usually the NETCONF clients subscribe to continuous datastore updates and rely on event notifications sent to the NETCONF client to check for the condition so that reaction to many network events may be very slow in the face of communication glitches between the client and the sever. Such solution doesn't scale well.

It is more desirable in many circumstances to delegate all event response behaviors (e.g., recover from network failure, instruct the network to control congestion) to the NETCONF server so that the network can react to network change as quickly as the event is detected.

The event response behaviors delegation can be done using YANG push subscription filter enhancements, e.g., define a new filter to allow the NETCONF client send updates only when the value falls within a certain range. Another example is to define a filter to allow the NETCONF client send updates only when the value exceeds a certain threshold for the first time but not again until the threshold is cleared. In the latter case, additional state is required.

In addition, the event response behavior delegation can be done by pushing ECA policy to the network device. Similar to YANG Push subscription filter, the ECA approach also includes filter and defines it as Event and Condition separately in the ECA policy model. Different from using YANG Push subscription filter, ECA allow a group of events to be observed, allow multiple actions to be triggered, e.g., sending log report notification, add or remove multiple YANG Push subscriptions.

2.3. Executing ECA Script in the Network Device

When the YANG Push subscription filter or ECA policy is pushed to the server, the server is expected to register the event conveyed in the YANG push subscription filter or Event-driven policy, generate server specific script. With a server specific script, the server can manipulate various network resources autonomously.

After the event registration, the server subscribes to its own publications encapsulated in the event notification with respect to all events that are associated with ECA Policy so that the publication is intercepted and all events specified in the ECA policy model are continuously monitored by the server before the publication is encapsulated in the event notification and sent to the YANG Push subscription's client. At the moment of event detection, the server loads the operational state data object filtered by the YANG Push subscription's filters or ECA policy into the auto-configured ECA's event and execute the ECA's associated condition-action chain.

The condition is associated with an ECA event and evaluated only within event threads triggered by the event detection, and the action corresponds to a set of statements that may trigger state changes in the device or publication content changes in the Event subscription and could be various different operations to be carried out by the server:

- o Configuration data object reconfiguration;
- o ECA Log report Notification;
- o Add or remove one or multiple YANG Push Subscriptions;

- o Invoke another Event in the same network device or different network device.

2.4. Event-Driven Notification Handling

ECA notifications are the only ECA actions that directly interact and hence need to be unambiguously understood by the client.

ECA notification can be sent when the client may find any interesting about the associated event with all the logic to compute said data (e.g., datastore content changes history, median values), and delegate computation task to the server via an ECA script.

When a "Send ECA notification" action is configured as an ECA Action, the client may receive different ECA notification associated with the same event or different events, YANG Push Publication will also be sent through Event notification. Therefore it is important for client to correlate of events and ECA notifications received from the server.

When ECA notification and YANG Push Publication are both pushed to the client, the client can execute client specific script generated in the same way as the server does and manipulate various network resources autonomously remotely. However the network resource can not be manipulated twice in both client and the server. Therefore policy conflict should be avoided or resolved.

2.5. Requisite State Information

A ECA policy rule is read as: When event occurs in a situation where condition is true, then action is executed. The ECA associated state is used to indicate when Events are triggered and what actions must be performed on the occurrence of an event.

A simple information model for one piece of the ECA associated state is as follows:

```
{
  event name;
  start time;
  end time;
  threshold value;
  event occurrence times
}
```

The event that is observed could be a fault, an alarm, a change in network state, network security threats, hardware malfunctions,

buffer utilization exceeding a threshold. For any of the aforementioned events, multiple actions may be triggered.

3. Architectural Concepts

3.1. What is Defined in ECA Policy?

ECA Event is a change of datastore operational state. Each policy rule consists of a set of conditions and a set of actions. Policy rules may be aggregated into policy groups. These groups may be nested, to represent a hierarchy of policies.

ECA Condition is evaluated to TRUE or FALSE logical expression. ECA condition is specified as a hierarchy of comparisons and logical combinations of thereof, which allows for configuring logical hierarchies. One of ECA condition example is logical hierarchies specified in a form of:

<target><relation><arg>

where target represent managed data object while arg represent either constant/enum/identity, Policy variable or pointed by XPath data store node or sub-tree,

relation is one of the comparison operations from the set: ==, !=, >, <, >=, <=

Logical calculation between multiple trigger conditions:

The YANG language cannot clearly describe complex logical operations between different condition lists under the same event, for example, (condition A & condition B) or condition C.

By default, the ECA model performs logic "AND" operation between different conditions in the same Event. That is, event is triggered when different conditions are met at the same time. For example,

Event A consists of two conditions:
o Condition A;
o Condition B;
If Condition A AND Condition B is met;
Event A is triggered;
Action A is executed.

For the logic "OR" operation between different conditions, the conditions can be defined in different events. If the corresponding event is triggered, the same action is executed. For example,

```
Event A is triggered on Condition A.  
Event B is triggered on Condition B.  
If Condition A is met;  
Event A is triggered;  
Action A is executed.  
If Condition B is met;  
Event B is triggered;  
Action A is executed.
```

ECA Action is one of the following operations to be carried out by a server:

- o Configuration data object reconfiguration
- o ECA Log report Notification
- o Add or remove one or multiple YANG Push Subscriptions
- o Invoke another Event in the same network device or different network device

In case of one event triggering another event, a set of events can be grouped together and executed, in a coordinated manner. The action associated with the event can be executed in the same network device or in different network devices. In the latter case, events executed by different network devices need to coordinate as a group to fulfil a task, previously set.

3.2. Where is ECA Script and State Held?

The ECA state information described in Section 2.5 and associated ECA script has to be held somewhere. There are two locations where this applies:

- o in a central controller where decisions about resource adjustment are made;
- o in the network nodes where the resources exist.

The first of these locations have a good visibility of the whole network or information of the flow packets are going to take through the entire network, but requires a centralized, searchable repository of all network information that can be used for diagnostics, service assurance, maintenance or audit purposes. The response to network event can be slow since all monitored data objects from large amount of network devices need to be sent and correlated at the point where decisions about resource adjustment are made, less alone multiple network event triggering a single action.

Conversely, if the ECA state and associated ECA script is held in the network nodes, it makes policing of resource adjustment easier. It means many points in the network can have immediate response to network event. The limitation is the configurations and state of a particular device does not have the visibility of the whole network or information of the flow packets are going to take through the entire network, so they provide little insight into network level policy-related behavior.

3.3. What State is Held?

As already described, the network control logic associated with ECA script needs access to ECA state table. It stores network events pushed from YANG push subscription or ECA policy model, threshold value it set for observed network management data object.

In addition, when the event needs to be continuously monitored, the Event scheduling information such as start time, end time can be included.

In case of sending updates only when the value exceeds a certain threshold for the first time but not again until the threshold is cleared, a threshold clear flag is also needed.

In case of monitoring the data change or data change rate, for example, YANG Push On-Change mode [RFC8641] or ECA Threshold Test [I.D-wwx-netmod-event-yang], the ECA state table need to store history state to check the condition to be satisfied and determine the current state.

4. Architecture Overview

The architectural considerations and conclusions described in the previous section lead to the architecture described in this section and illustrated in Figure 1. The interfaces and interactions shown in the figure and labeled (a) through (f) are described in Section 4.1.

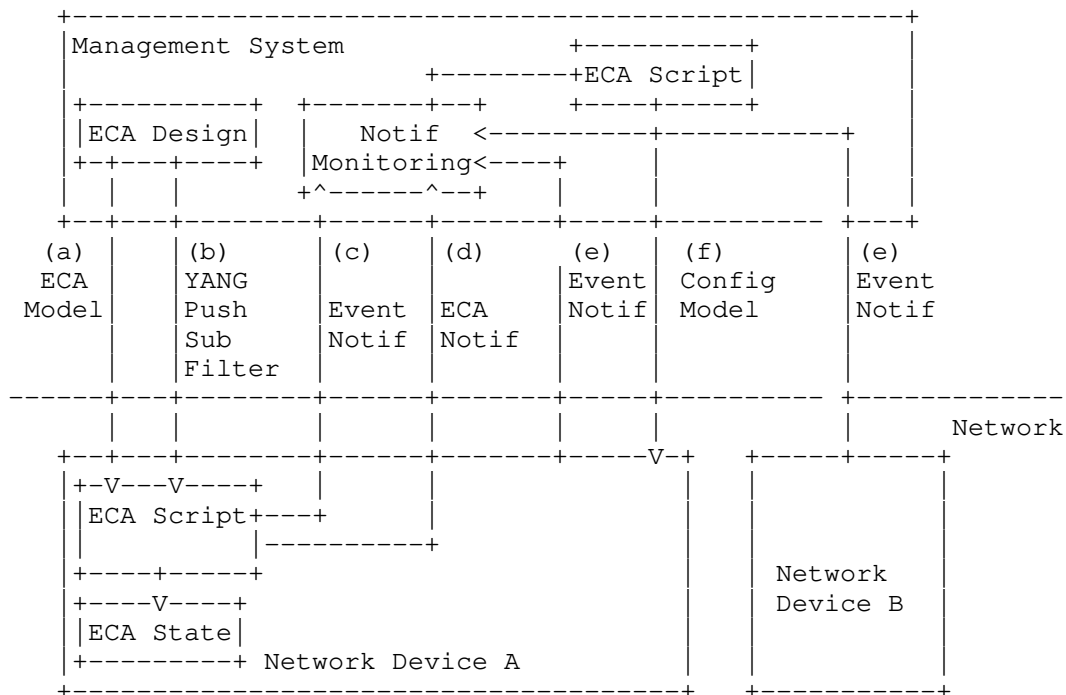


Figure 1. Reference Architecture for Use of ECA and Network Self Management

4.1. Telemetry Automation in the Network Device

As shown in Figure 1, some component in the management system defines and designs ECA Policy rule. This may be invoked by a Service assurance application or device fault self-management application. We show this component on the figure as the "ECA Design", and it extracts Event and Condition in the ECA model and fill into YANG Push Subscription as filter. When YANG Push subscription filter is pushed down to the network device, ECA script can be generated automatically from it (ECA script can also be generated in the management system and downloaded to the network device). The YANG Push subscription request, indicated on Figure 1 by the arrow (b), includes all of the parameters of network management data objects that the requester wishes to be supplied, such as filter node, threshold value, start time, and end time. Note that the requester in this case may be the management system shown in the figure or a distinct system such as data collector.

The network device registers network event that is corresponding to the filter carried in the YANG Push Subscription and enters the

network event in its ECA state and then the server subscribes to its own continuous datastore updates in the operational state datastore that is encapsulated in the event notification as publication to the YANG Push subscriber.

Upon the network event is detected, the server intercepts the publication of subscribed data and loads the operational state data object in the operational state datastore into the auto-configured ECA's event and execute the ECA's associated condition. When ECA Condition is evaluated to TRUE, the operational state data objects will be filtered and the remaining data objects will be entered back into the publication of subscribed data and encapsulated in the event notification (c) and sent to notification monitoring component in the management system.

The notification monitoring component may further derive some new ECA policy rule and fed into ECA Design component. The remaining procedure is same as the procedure starting from (b).

Alternatively, the ECA design component can push ECA model directly with additional actions included (a) to the network device, ECA script is generated automatically from ECA model. The ECA model request, indicated on Figure 1 by the arrow (a), includes additional action parameters besides one included in the YANG Push subscription request.

The network device register network event that is corresponding to the ECA carried in the ECA request and enter them in its ECA state and then the server subscribes to its own continuous datastore updates in the operational state datastore that is encapsulated in the event notification as publication to the YANG Push subscriber.

Upon the network event is detected, the server loads the operational state data object in the operational state datastore into the auto-configured ECA's event and execute the ECA's associated condition. Different from YANG Push subscription filter, the server will not intercept the publication of subscribed data. Instead, it allows the server to trigger a set of actions associated with the network event, e.g., send ECA log report notification, add/remove YANG push subscription, reconfigure the network management data object within the control of the server. After all actions are executed, one or multiple separate ECA notifications (d) can be sent to the notification monitoring component in the management system, the remaining procedure is same as YANG Push subscription case.

Conversely, when, network level policy-related behavior became necessary, once a subscription has been set up, event notification message associated with the subscription from different network

device will be sent to the notification monitoring component in the management system(e), which in turn trigger network behavior change on the network device via configuration model (f).

4.2. Detecting and Resolving Policy Conflict

There are two possible places where policy conflict can take places:

1. An event triggers multiple actions in the network device that cannot occur together as specified by the system administrator.
2. Multiple ECA notifications or multiple combination of ECA notification and Event notification lead to generate ECA policy that cannot occur together.

In both case, policy conflict can be addressed by policy conflict detection mechanism and Policy validation mechanism.

4.3. Chain Reaction of Coordinated Events

In some cases events executed by the same or different network devices can be executed in a coordinated manner. To make sure these network devices coordinate on some task or a group of events coordinate in the same network device, these events on the same or different network devices need to be pre-configured to work together. During capability negotiation phase, the management system should know what each network device supports, which event may take action, and what condition on which event. So ECA model with multiple events can be configured on the network device to allow one event be triggered by another event configured on the same network device.

5. Security Considerations

The framework described in this document for supporting autonomic event-driven self-management will require consideration of potential security and operational requirements, and ensure best security practices and methods are applied.

Key security considerations that will be discussed in future versions of this document, include:

- o Authentication of ECA programming requests;
- o Application of suitable authorization methods when enabling ECA functions;
- o Securing ECA communication channels;

- o Locking ECA device config and state databases;
- o Mitigation, and negation, of ECA functional component attacks;
- o Logging and auditing of ECA transactions;
- o Maintaining ECA device confidentially.

6. Acknowledgements

This work has benefited from the discussions of ECA Policy over the years. In particular, the SUPA project [<https://datatracker.ietf.org/wg/supa/about/>] provided approaches to express high-level, possibly network-wide policies to a network management function (within a controller, an orchestrator, or a network element).

Igor Bryskin, Xufeng Liu, Alexander Clemm, Henk Birkholz, Tianran Zhou contributed to an earlier version of [GNCA]. We would like to thank the authors of that document on event response behaviors delegation for material that assisted in thinking that helped develop this document.

Finally, the authors would like to thank David Hutchison and Mehdi Bezahaf at Lancaster University, Phil Eardley and Andy Reid at British Telecom, for their input and applicability of ECA device self management to the Next Generation Converged Digital Infrastructure (NG-CDI) project.

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>>.
- [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>>.
- [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>>.

- [RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <<https://www.rfc-editor.org/info/rfc8341>>.
- [RFC8342] Bjorklund, M., Schoenwaelder, J., Shafer, P., Watsen, K., and R. Wilton, "Network Management Datastore Architecture (NMDA)", RFC 8342, DOI 10.17487/RFC8342, March 2018, <<https://www.rfc-editor.org/info/rfc8342>>.

7.2. Informative References

- [I-D.bryskin-netconf-automation-yang]
Bryskin, I., Liu, X., Clemm, A., Birkholz, H., and T. Zhou, "Generalized Network Control Automation YANG Model", draft-bryskin-netconf-automation-yang-03 (work in progress), July 2019.
- [I-D.clemm-netmod-push-smart-filters]
Clemm, A., Voit, E., Liu, X., Bryskin, I., Zhou, T., Zheng, G., and H. Birkholz, "Smart Filters for Push Updates", draft-clemm-netmod-push-smart-filters-01 (work in progress), October 2018.
- [I-D.clemm-nmrg-dist-intent]
Clemm, A., Ciavaglia, L., Granville, L., and J. Tantsura, "Intent-Based Networking - Concepts and Overview", draft-clemm-nmrg-dist-intent-02 (work in progress), July 2019.
- [I-D.wwx-netmod-event-yang]
Wang, Z., WU, Q., Xie, C., Bryskin, I., Liu, X., Clemm, A., Birkholz, H., and T. Zhou, "A YANG Data model for ECA Policy Management", draft-wwx-netmod-event-yang-04 (work in progress), November 2019.
- [RFC8328] Liu, W., Xie, C., Strassner, J., Karagiannis, G., Klyus, M., Bi, J., Cheng, Y., and D. Zhang, "Policy-Based Management Framework for the Simplified Use of Policy Abstractions (SUPA)", RFC 8328, DOI 10.17487/RFC8328, March 2018, <<https://www.rfc-editor.org/info/rfc8328>>.
- [RFC8572] Watsen, K., Farrer, I., and M. Abrahamsson, "Secure Zero Touch Provisioning (SZTP)", RFC 8572, DOI 10.17487/RFC8572, April 2019, <<https://www.rfc-editor.org/info/rfc8572>>.

Authors' Addresses

Mohamed Boucadair
Orange
Rennes 35000
France

Email: mohamed.boucadair@orange.com

Qin Wu
Huawei
101 Software Avenue, Yuhua District
Nanjing, Jiangsu 210012
China

Email: bill.wu@huawei.com

Michael Wang
Huawei
101 Software Avenue, Yuhua District
Nanjing, Jiangsu 210012
China

Email: wangzitao@huawei.com

Daniel King
Lancaster University
Bailrigg, Lancaster LA1 4YW
UK

Email: d.king@lancaster.ac.uk

Chongfeng Xie
China Telecom
Beijing
China

Email: xiechf.bri@chinatelecom.cn

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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 6, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Terminology	2
2. Introduction	4
3. Architecture	5
3.1. Decomposing a Service Instance Configuration into an Assurance Tree	7
3.2. Intent and Assurance Tree	9
3.3. Subservices	9
3.4. Building the Expression Tree from the Assurance Tree	10
3.5. Building the Expression from a Subservice	10
3.6. Open Interfaces with YANG Modules	10
4. Security Considerations	11
5. IANA Considerations	11
6. Open Issues	11
7. References	11
7.1. Normative References	11
7.2. Informative References	11
Appendix A. Changes between revisions	12
Acknowledgements	12
Authors' Addresses	12

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 healthy. 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 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.

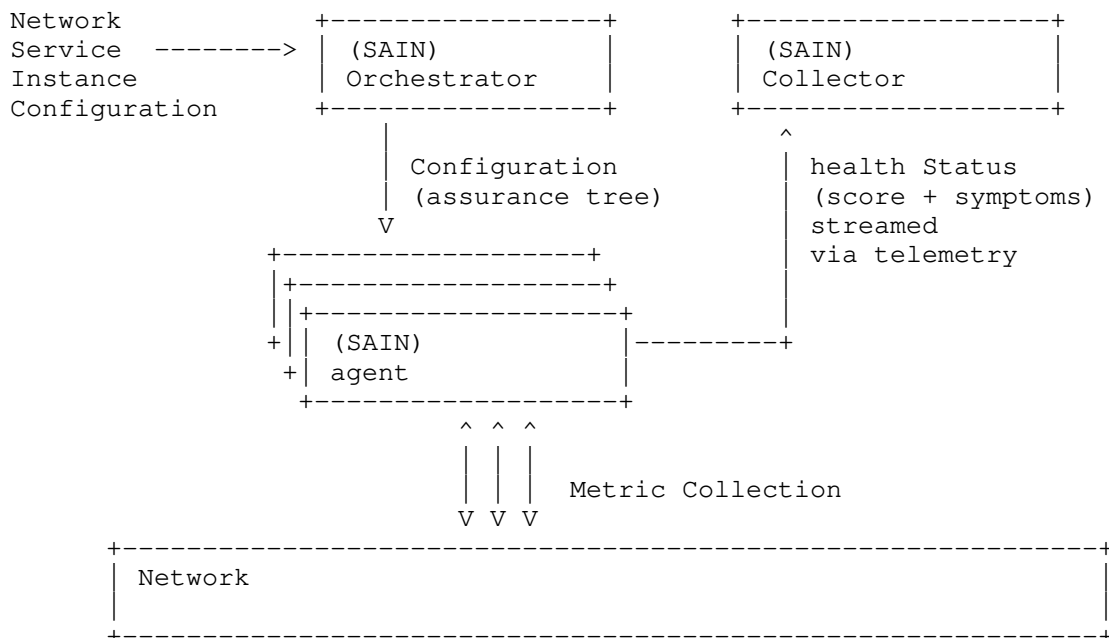


Figure 1: SAIN Architecture

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 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.

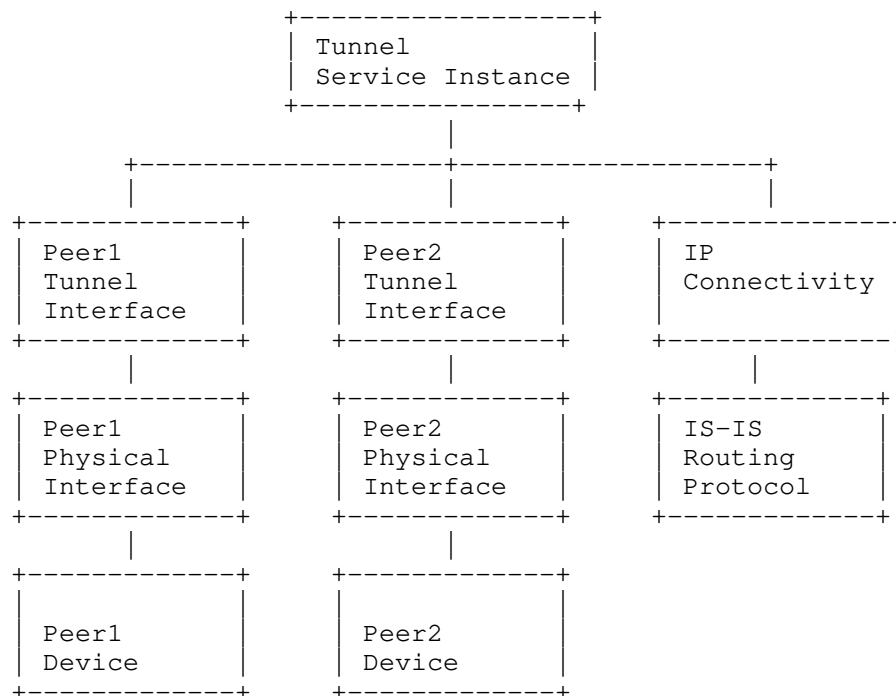


Figure 2: Assurance Tree Example

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 health statuses from the collected metrics.

Subservices shall be device independent. 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 (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

OPSAWG
Internet-Draft
Intended status: Informational
Expires: October 25, 2021

B. Claise
Huawei
J. Quilbeuf
Independent
D. Lopez
Telefonica I+D
D. Voyer
Bell Canada
T. Arumugam
Cisco Systems, Inc.
April 23, 2021

Service Assurance for Intent-based Networking Architecture
draft-claise-opsawg-service-assurance-architecture-05

Abstract

This document describes an 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 when a network component fails or degrades.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 25, 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Terminology	2
2. Introduction	5
3. Architecture	6
3.1. Decomposing a Service Instance Configuration into an Assurance Graph	9
3.2. Intent and Assurance Graph	10
3.3. Subservices	11
3.4. Building the Expression Graph from the Assurance Graph	11
3.5. Building the Expression from a Subservice	12
3.6. Open Interfaces with YANG Modules	12
3.7. Handling Maintenance Windows	13
3.8. Flexible Architecture	14
3.9. Timing	15
3.10. New Assurance Graph Generation	15
4. Security Considerations	16
5. IANA Considerations	16
6. Contributors	16
7. Open Issues	16
8. References	16
8.1. Normative References	16
8.2. Informative References	17
Appendix A. Changes between revisions	18
Acknowledgements	18
Authors' Addresses	19

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.

SAIN Agent: 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.

Assurance 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 relations.

SAIN collector: Component that fetches or receives 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 Graph: Generic term for a DAG representing a computation in SAIN. More specific terms are:

- o **Subservice Expressions:** expression graph representing all the computations to execute for a subservice.
- o **Service Expressions:** expression graph representing all the computations to execute for a service instance, i.e. including the computations for all dependent subservices.
- o **Global Computation Graph:** 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.

Informational Dependency: Type of dependency whose score does not impact the 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: Type of dependency whose score impacts the 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.

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.

Service configuration orchestrator: 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."

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

Health status: 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: Integer ranging from 0 to 100 indicating the health of a subservice. A score of 0 means that the subservice is broken, a score of 100 means that the subservice is perfectly operational.

Subservice: Part of an assurance graph 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, service configuration 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+ [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 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 graph, deduced from the service definition and from the network configuration. This assurance graph is decomposed into components, which are then assured independently. The root of the assurance graph represents the service to assure, and its children represent components identified as its direct dependencies; each component can have dependencies as well. The SAIN architecture maintains the correct assurance graph when services are modified or when the network conditions change.

When a service is degraded, the framework will highlight where in the assurance service graph to look, as opposed to going hop by hop to troubleshoot the issue. Not only can this framework help to correlate service degradation with network root cause/symptoms, but it can deduce from the assurance graph 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.

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 of components (Section 3.8).

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.

The SAIN architecture is a generic architecture, applicable to multiple environments. Obviously wireline but also wireless, including 5G, virtual infrastructure manager (VIM), and even virtual functions. Thanks to the distributed graph design principle, graphs from different environments/orchestrator can be combined together.

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".

To compute the health status of such as service, the service 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 which computes the health status of the service.

The overall architecture of our solution is presented in Figure 1. Based on the service configuration, the SAIN orchestrator deduces the assurance graph. It then sends to the SAIN agents the assurance graph along 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. Finally, the automation loop is closed by having the SAIN Collector providing feedback to the network orchestrator.

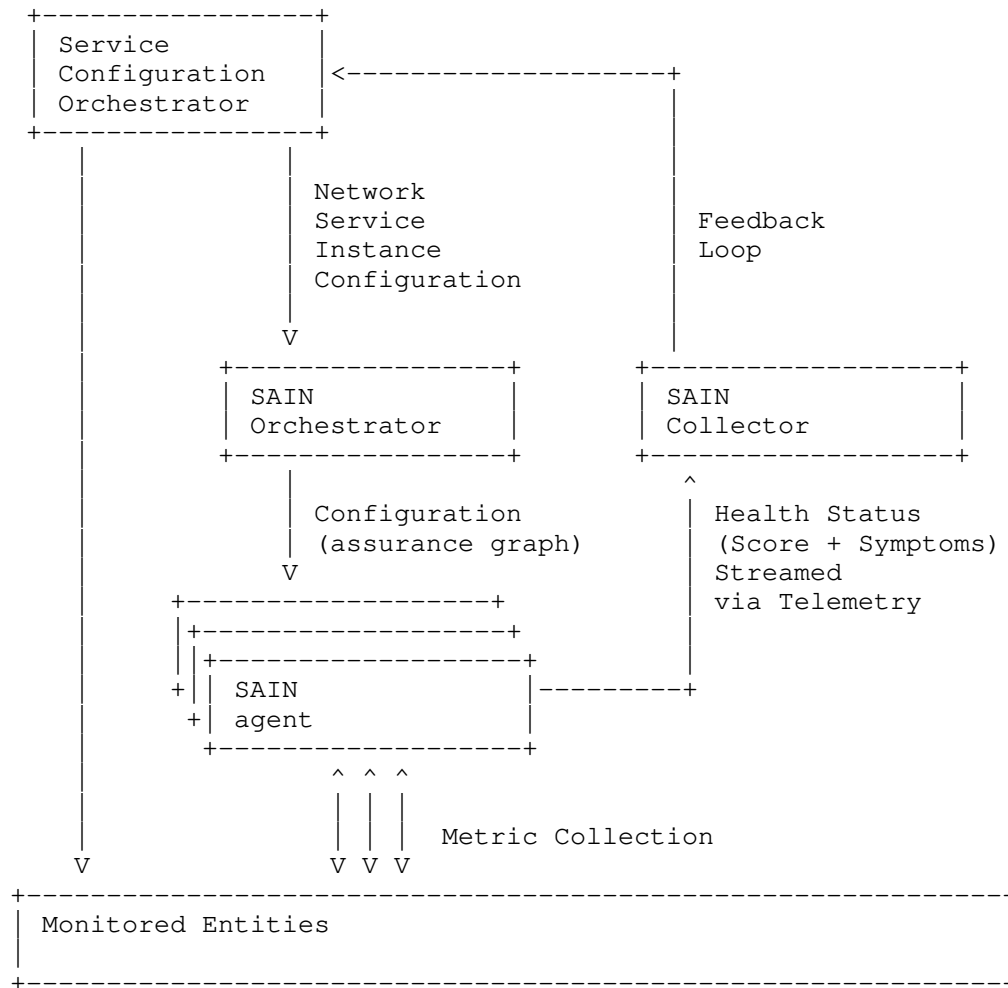


Figure 1: SAIN Architecture

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

3.1. Decomposing a Service Instance Configuration into an Assurance Graph

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

The decomposition into subservices is an important function of this architecture, for the following reasons.

- o TThe result of this decomposition provides a relational picture of a service instance, that can be represented as a graph (called assurance graph) 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 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: 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, 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.

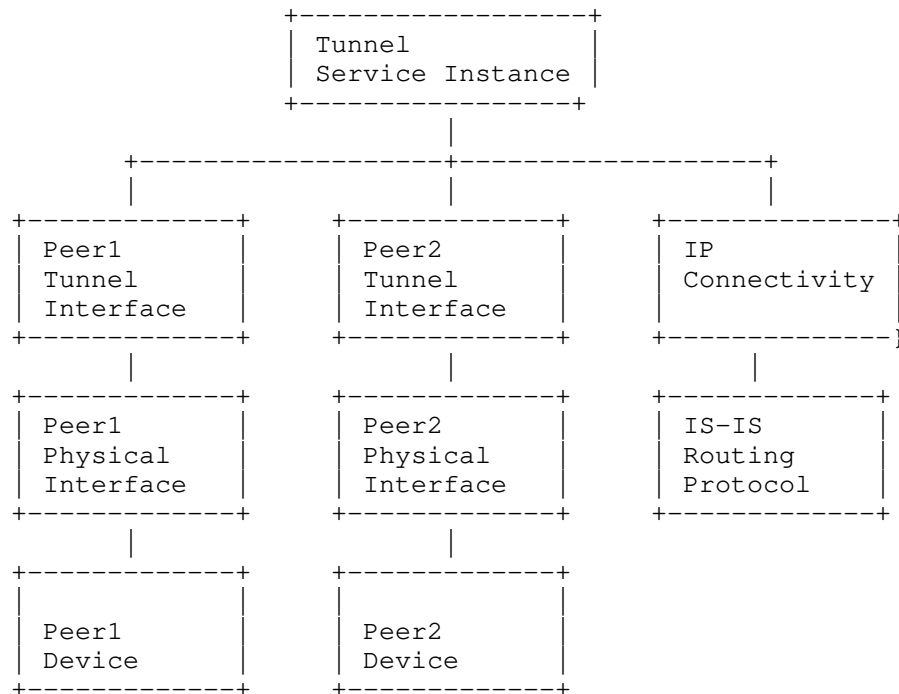


Figure 2: Assurance Graph Example

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 be reflected in the assurance graph. As a first example, a change of routing protocol from IS-IS to OSPF would change the assurance graph 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 Graph

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 graph.

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

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 document.

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.

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

A subservice is also characterized by a list of metrics to fetch and a list of computations 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 is derived a so-called global computation graph. 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. 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. In other words, the global computation graph encodes all the operations needed to produce health statuses from the collected metrics.

Subservices shall be device independent. 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 (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. 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 non-standard YANG models, via MIB modules, Command Line Interface (CLI), etc., effectively implementing a normalization layer between data models and information models.

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 the YANG modules specified in YANG Modules for Service Assurance [I-D.claise-opsawg-service-assurance-yang]; they specify 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 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.
- o Assurance telemetry: export the health status of the subservices, along with the observed symptoms.

3.7. Handling Maintenance Windows

Whenever network components are under maintenance, the operator want 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, as well as symptoms related to the device-specific subservices, such as the interfaces, as their state changes is probably the consequence of the maintenance.

To configure network components as "under maintenance" in the SAIN architecture, the ietf-service-assurance model proposed in [I-D.claise-opsawg-service-assurance-yang] specifies an "under-maintenance" flag per service or subservice instance. When this flag is set and only when this flag is set, the companion field "maintenance-contact" must be set to a string that identifies the person or process who requested the maintenance. Any symptom produced by a service or subservice under maintenance, or by one of its dependencies MUST NOT be reported. A service or subservice under maintenance MAY propagate a symptom "Under Maintenance" towards services or subservices that depend on it.

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

- o Device maintenance, for instance upgrading the device OS. The operator sets the "under-maintenance" flag for the subservice "Peer1" device. This inhibits the emission of symptoms from "Peer1 Physical Interface", "Peer1 Tunnel Interface" and "Tunnel Service Instance". All other subservices are unaffected.
- o Interface maintenance, for instance replacing a broken optic. The operator sets the "under-maintenance" flag for the subservice "Peer1 Physical Interface". This inhibits the emission of

symptoms from "Peer 1 Tunnel Interface" and "Tunnel Service Instance". All other subservices are unaffected.

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

3.8. Flexible Architecture

The SAIN architecture is flexible in terms of components. While the SAIN architecture in Figure 1 makes a distinction between two components, the SAIN configuration orchestrator and the SAIN orchestrator, in practice those two components are mostly likely combined. Similarly, the SAIN agents are displayed in Figure 1 as being separate components. Practically, 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. Most examples in this document deal with the notion of Network Service YANG modules, with well known service such as L2VPN or tunnels. However, the concepts of services is general enough to cross into different domains. One of them is the domain of service management on network elements, with also requires its own assurance. Examples includes a DHCP server on a linux server, a data plane, an IPFIX export, etc. The notion of "service" is generic in this architecture. Indeed, a configured service can itself be a service for someone else. Exactly like an DHCP server/ data plane/IPFIX export can be considered as services for a device, exactly like an routing instance can be considered as a service for a L3VPN, exactly like a tunnel can considered as a service for an application in the cloud. 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, our 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 constrain. 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 well possible within this architecture.

3.9. Timing

The SAIN architecture requires the Network Time Protocol (NTP) [RFC5905] between all elements: monitored entities, SAIN agents, Service Configuration 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.

The SAIN agent might have to remove some symptoms for specific subservice symptoms, because there are outdated and not relevant any longer, or simply because the SAIN agent needs to free up some space. Regardless of the reason, it's important for a SAIN collector (re-)connecting 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.

3.10. New Assurance Graph Generation

The assurance graph will change along the time, because services and subservices come and go (changing the dependencies between subservices), or simply because a subservice is now under maintenance. 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:

- o Keep the previous assurance-graph-last-change value (let's call it time T)
- o Run through all subservice instance and process the subservice instances for which the last-change is newer than the time T
- o Keep the new assurance-graph-last-change as the new referenced date and time

4. Security Considerations

The SAIN architecture helps operators to reduce the mean time to detect and mean time to repair. As such, it should not cause any security threats. However, the SAIN agents must be secure: a compromised SAIN agents could be sending wrong root causes or symptoms to the management systems.

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 NETCONF or TLS [RFC8446] for RESTCONF.

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.

If a closed loop system relies on this architecture then the well known issue of those system also applies, i.e., a lying device or compromised agent could trigger partial reconfiguration of the service or network. The SAIN architecture neither augments or reduces this risk.

5. IANA Considerations

This document includes no request to IANA.

6. Contributors

- o Youssef El Fathi
- o Eric Vyncke

7. Open Issues

Refer to the Intent-based Networking NMRG documents

8. References

8.1. 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>>.

- [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>>.
- [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>>.

8.2. Informative References

- [I-D.claise-opsawg-service-assurance-yang] Claise, B. and J. Quilbeuf, "Service Assurance for Intent-based Networking Architecture", February 2020.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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., 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>>.

Appendix A. Changes between revisions

v02 - v03

- o Timing Concepts
- o New Assurance Graph Generation

v01 - v02

- o Handling maintenance windows
- o Flexible architecture better explained
- o Improved the terminology
- o Notion of mapping information model to data model, while waiting for YANG to be everywhere
- o Started a security considerations section

v00 - v01

- o Terminology clarifications
- o Figure 1 improved

Acknowledgements

The authors would like to thank Stephane Litkowski, Charles Eckel, Rob Wilton, Vladimir Vassiliev, Gustavo Albuquerque, Stefan Vallin, and Eric Vyncke for their reviews and feedback.

Authors' Addresses

Benoit Claise
Huawei

Email: benoit.claise@huawei.com

Jean Quilbeuf
Independent

Email: jean@quilbeuf.net

Diego R. Lopez
Telefonica I+D
Don Ramon de la Cruz, 82
Madrid 28006
Spain

Email: diego.r.lopez@telefonica.com

Dan Voyer
Bell Canada
Canada

Email: daniel.voyer@bell.ca

Thangam Arumugam
Cisco Systems, Inc.
Milpitas (California)
United States of America

Email: tarumuga@cisco.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 7, 2020

A. Clemm
Futurewei
L. Ciavaglia
Nokia
L. Granville
Federal University of Rio Grande do Sul (UFRGS)
J. Tantsura
Apstra, Inc.
November 4, 2019

Intent-Based Networking - Concepts and Overview
draft-clemm-nmrg-dist-intent-03

Abstract

Intent and Intent-Based Networking are taking the industry by storm. At the same time, those terms are used loosely and often inconsistently, in many cases overlapping and confused with other concepts such as "policy". This document is intended to clarify the concept of "Intent" and provide an overview of functionality that associated with it. The goal is to contribute towards a common and shared understanding of terms, concepts, and functionality which can be used as foundation to guide further definition of associated research and engineering problems and their solutions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 7, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Key Words	4
3. Definitions and Acronyms	4
4. Introduction of Concepts	5
4.1. Intent and Intent-Based Management	5
4.2. Related Concepts	6
4.2.1. Service Models	6
4.2.2. Policy and Policy-Based Management	8
4.2.3. Distinguishing between Intent, Policy, and Service Models	10
5. Principles	11
6. Lifecycle	14
7. Intent-Based Networking - Functionality	18
7.1. Intent Fulfillment	18
7.2. Intent Assurance	18
8. Items for Discussion	19
9. IANA Considerations	19
10. Security Considerations	19
11. References	19
11.1. Normative References	19
11.2. Informative References	19
Authors' Addresses	21

1. Introduction

Traditionally in the IETF, interest with regard to management and operations has focused on individual network and device features. Standardization emphasis has generally been put on management instrumentation that needed to be provided to a networking device. A prime example for this is SNMP-based management and the 200+ MIBs that have been defined by the IETF over the years. More recent examples include YANG data model definitions for aspects such as interface configuration, ACL configuration, or Syslog configuration.

There is a sense and reality that in modern network environments managing networks by configuring myriads of "nerd knobs" on a device-

by-device basis is no longer sustainable. Big challenges arise with keeping device configurations not only consistent across a network, but consistent with the needs of services and service features they are supposed to enable. Adoptability to changes at scale is a fundamental property of a well designed IBN system, that requires ability to consume and process analytics that are context/intent aware at near real time speeds. At the same time, operations need to be streamlined and automated wherever possible to not only lower operational expenses, but allow for rapid reconfiguration of networks at sub-second time scales and to ensure networks are delivering their functionality as expected.

Accordingly, IETF has begun to address end-to-end management aspects that go beyond the realm of individual devices in isolation. Examples include the definition of YANG models for network topology [RFC8345] or the introduction of service models used by service orchestration systems and controllers [RFC8309]. In addition, a lot of interest has been fueled by the discussion about how to manage autonomic networks as discussed in the ANIMA working group. Autonomic networks are driven by the desire to lower operational expenses and make management of the network as a whole exceptionally easy, putting it at odds with the need to manage the network one device and one feature at a time. However, while autonomic networks are intended to exhibit "self-management" properties, they still require input from an operator or outside system to provide operational guidance and information about the goals, purposes, and service instances that the network is to serve.

This vision has since caught on with the industry in a big way, leading to a significant number solutions that offer "intent-based management" that promise network providers to manage networks holistically at a higher level of abstraction and as a system that happens to consist of interconnected components, as opposed to a set of independent devices (that happen to be interconnected). Those offerings include IBN systems (offering full lifecycle of intent), SDN controllers (offering a single point of control and administration for a network) as well as network management and Operations Support Systems (OSS).

However, it has been recognized for a long time that comprehensive management solutions cannot operate only at the level of individual devices and low-level configurations. In this sense, the vision of "intent" is not entirely new. In the past, ITU-T's model of a Telecommunications Management Network, TMN, introduced a set of management layers that defined a management hierarchy, consisting of network element, network, service, and business management. High-level operational objectives would propagate in top-down fashion from upper to lower layers. The associated abstraction hierarchy was key

to decompose management complexity into separate areas of concerns. This abstraction hierarchy was accompanied by an information hierarchy that concerned itself at the lowest level with device-specific information, but that would, at higher layers, include, for example, end-to-end service instances. Similarly, the concept of "policy-based management" has for a long time touted the ability to allow users to manage networks by specifying high-level management policies, with policy systems automatically "rendering" those policies, i.e. breaking them down into low-level configurations and control logic.

What has been missing, however, is putting these concepts into a more current context and updating it to account for current technology trends. This document attempts to clarify the concepts behind intent. It differentiates it from related concepts. It also provides an overview of first-order principles of Intent-Based Networking as well as associated functionality. In addition, a number of research challenges are highlighted. The goal is to contribute to a common and shared understanding that can be used as a foundation to articulate research and engineering problems in the area of Intent-Based Networking.

2. Key Words

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.

3. Definitions and Acronyms

ACL: Access Control List

Intent: An abstracted, declarative and vendor agnostic set of rules used to provide full lifecycle (Design/Build/Deploy/Validate) to a network and services it provides.

Policy: A rule, or set of rules, that governs the choices in behavior of a system.

SSoT: Single Source of Truth - A functional block in an IBN system that normalizes user' intent and serves as the single source of data for the lower layers.

IBA: Intent Based Analytics - Analytics that are defined and derived from user' intent and used to validate the intended state.

IBS: Intent Based System.

PDP: Policy Decision Point

PEP: Policy Enforcement Point

Service Model: A model that represents a service that is provided by a network to a user.

4. Introduction of Concepts

The following section provides an overview of the concept of intent respectively intent-based management. It also provides an overview of the related concepts of service models, and of policies respectively policy-based management, and explains how they relate to intent and intent-based management.

4.1. Intent and Intent-Based Management

In the context of Autonomic Networks, Intent is defined as "an abstract, high-level policy used to operate a network" [RFC7575]. According to this definition, an intent is a specific type of policy. However, to avoid using "intent" simply as a synonym for "policy, a clearer distinction needs to be introduced that distinguishes intent clearly from other types of policies.

For one, while Intent-Based Management clearly aims to lead towards networks that are dramatically simpler to manage and operate requiring only minimal outside intervention, the concept of "intent" is not limited to autonomic networks, but applies to any network. Networks, even when considered "autonomic", are not clairvoyant and have no way of automatically knowing particular operational goals nor what instances of networking services to support. In other words, they do not know what the "intent" of the network provider is that gives the network the purpose of its being. This still needs to be communicated by what informally constitutes "intent".

More specifically, intent is a declaration of operational goals that a network should meet and outcomes that the network is supposed to deliver, without specifying how to achieve them. Those goals and outcomes are defined in a manner that is purely declarative - they specify what to accomplish, not how to achieve it. "Intent" thus applies several important concepts simultaneously:

- o It provides data abstraction: Users and operators do not need to be concerned with low-level device configuration and nerd knobs.

- o It provides functional abstraction from particular management and control logic: Users and operators do not need to be concerned even with how to achieve a given intent. What is specified is a desired outcome, with the intent-based system automatically figuring out a course of action (e.g. a set of rules, an algorithm) for how to achieve the outcome.

In an autonomic network, intent should be rendered by the network itself, i.e. translated into device-specific rules and courses of action. Ideally, it should not even be orchestrated or broken down by a higher-level, centralized system, but by the network devices themselves using a combination of distributed algorithms and local device abstraction. Because intent holds for the network as a whole, not individual devices, it needs to be automatically disseminated across all devices in the network, which can themselves decide whether they need to act on it. This facilitates management even further, since it obviates the need for a higher-layer system to break down and decompose higher-level intent, and because there is no need to even discover and maintain an inventory of the network to be able to manage it.

Tentative definition for intent-based networks Networks configuring and adapting autonomously to the user or operator intentions (i.e., a desired state or behavior) without the need to specify every technical detail of the process and operations to achieve it (i.e., the "machines" will figure out on their own how to realize the user goal).

Other definitions of intent exist such as [TR523] and will be investigated in future revisions of this document. Likewise, some definitions of intent allow for the presence of a centralized function that renders the intent into lower-level policies or instructions and orchestrates them across the network. While to the end user the concept of "intent" appears the same regardless of its method of rendering, this interpretation opens a slippery slope of how to clearly distinguish "intent" from other higher-layer abstractions. Again, these notions will be further investigated in future revisions of this document and in collaboration with NMRG.

4.2. Related Concepts

4.2.1. Service Models

A service model is a model that represents a service that is provided by a network to a user. Per [RFC8309], a service model describes a service and its parameters in a portable/vendor agnostic way that can be used independent of the equipment and operating environment on which the service is realized. Two subcategories are distinguished:

a "Customer Service Model" describes an instance of a service as provided to a customer, possibly associated with a service order. A "Service Delivery Model" describes how a service is instantiated over existing networking infrastructure.

An example of a service could be a Layer 3 VPN service [RFC8299], a Network Slice, or residential Internet access. Service models represent service instances as entities in their own right. Services have their own parameters, actions, and lifecycles. Typically, service instances can be bound to end users, who might be billed for the service.

Instantiating a service typically involves multiple aspects:

- o A user (or northbound system) needs to define and/or request a service to be instantiated.
- o Resources need to be allocated, such as IP addresses, AS numbers, VLAN or VxLAN pools, interfaces, bandwidth, or memory.
- o How to map services to the resources needs to be defined. Multiple mappings are often possible, which to select may depend on context (such as which type of access is available to connect the end user with the service).
- o [I-D.ietf-teas-te-service-mapping-yang] is an example of such mapping - a data model to map customer service models (e.g., the L3VPM Service Model) to Traffic Engineering (TE) models (e.g., the TE Tunnel or the Abstraction and Control of Traffic Engineered Networks Virtual Network model)
- o Bindings need to be maintained between upper and lower-level objects.
- o Once instantiated, the service needs to be validated and assured to ensure that the network indeed delivers the service as requested.

They involve a system, such as a controller, that provides provisioning logic. Orchestration itself is generally conducted using a "push" model, in which the controller/manager initiates the operations as required, pushing down the specific configurations to the device. (In addition to instantiating and creating new instances of a service, updating, modifying, and decommissioning services need to be also supported.) The device itself typically remains agnostic to the service or the fact that its resources or configurations are part of a service/concept at a higher layer.

Instantiated service models map to instantiated lower-layer network and device models. Examples include instances of paths, or instances of specific port configurations. The service model typically also models dependencies and layering of services over lower-layer networking resources that are used to provide services. This facilitates management by allowing to follow dependencies for troubleshooting activities, to perform impact analysis in which events in the network are assessed regarding their impact on services and customers. Services are typically orchestrated and provisioned top-to-bottom, which also facilitates keeping track of the assignment of network resources. Service models might also be associated with other data that does not concern the network but provides business context. This includes things such as customer data (such as billing information), service orders and service catalogues, tariffs, service contracts, and Service Level Agreements (SLAs) including contractual agreements regarding remediation actions.

Like intent, service models provide higher layers of abstraction. Service models are often also complemented with mappings that capture dependencies between service and device or network configurations. Unlike intent, service models do not allow to define a desired "outcome" that would be automatically maintained by the intent system. Instead, management of service models requires development of sophisticated algorithms and control logic by network providers or system integrators.

4.2.2. Policy and Policy-Based Management

Policy-based management (PBM) is a management paradigm that separates the rules that govern the behavior of a system from the functionality of the system. It promises to reduce maintenance costs of information and communication systems while improving flexibility and runtime adaptability. It is present today at the heart of a multitude of management architectures and paradigms including SLA-driven, Business-driven, autonomous, adaptive, and self-* management [Boutaba07]. The interested reader is asked to refer to the rich set of existing literature which includes this and many other references. In the following, we will only provide a much-abridged and distilled overview.

At the heart of policy-based management is the concept of a policy. Multiple definitions of policy exist: "Policies are rules governing the choices in behavior of a system" [Sloman94]. "Policy is a set of rules that are used to manage and control the changing and/or maintaining of the state of one or more managed objects" [Strassner03]. Common to most definitions is the definition of a policy as a "rule". Typically, the definition of a rule consists of an event (whose occurrence triggers a rule), a set of conditions

(that get assessed and that must be true before any actions are actually "fired"), and finally a set of one or more actions that are carried out when the condition holds.

Policy-based management can be considered an imperative management paradigm: Policies specify precisely what needs to be done when and in which circumstance. Using policies, management can in effect be defined as a set of simple control loops. This makes policy-based management a suitable technology to implement autonomic behavior that can exhibit self-* management properties including self-configuration, self-healing, self-optimization, and self-protection. In effect, policies define management as a set of simple control loops.

Policies typically involve a certain degree of abstraction in order to cope with heterogeneity of networking devices. Rather than having a device-specific policy that defines events, conditions, and actions in terms of device-specific commands, parameters, and data models, policy is defined at a higher-level of abstraction involving a canonical model of systems and devices to which the policy is to be applied. A policy agent on a controller or the device subsequently "renders" the policy, i.e., translates the canonical model into a device-specific representation. This concept allows to apply the same policy across a wide range of devices without needing to define multiple variants. In other words - policy definition is de-coupled from policy instantiation and policy enforcement. This enables operational scale and allows network operators and authors of policies to think in higher terms of abstraction than device specifics and be able to reuse the same, high level definition definition across different networking domains, WAN, DC or public cloud.

Policy-based management is typically "push-based": Policies are pushed onto devices where they are rendered and enforced. The push operations are conducted by a manager or controller, which is responsible for deploying policies across the network and monitor their proper operation. That said, other policy architectures are possible. For example, policy-based management can also include a pull-component in which the decision regarding which action to take is delegated to a so-called Policy Decision Point (PDP). This PDP can reside outside the managed device itself and has typically global visibility and context with which to make policy decisions. Whenever a network device observes an event that is associated with a policy, but lacks the full definition of the policy or the ability to reach a conclusion regarding the expected action, it reaches out to the PDP for a decision (reached, for example, by deciding on an action based on various conditions). Subsequently, the device carries out the decision as returned by the PDP - the device "enforces" the policy

and hence acts as a PEP (Policy Enforcement Point). Either way, PBM architectures typically involve a central component from which policies are deployed across the network, and/or policy decisions served.

Like Intent, policies provide a higher layer of abstraction. Policy systems are also able to capture dynamic aspects of the system under management through specification of rules that allow to define various triggers for certain courses of actions. Unlike intent, the definition of those rules (and courses of actions) still needs to be articulated by users. Since the intent is unknown, conflict resolution within or between policies requires interactions with a user or some kind of logic that resides outside of PBM.

4.2.3. Distinguishing between Intent, Policy, and Service Models

What Intent, Policy, and Service Models all have in common is the fact that they involve a higher-layer of abstraction of a network that does not involve device-specifics, that generally transcends individual devices, and that makes the network easier to manage for applications and human users compared to having to manage the network one device at a time. Beyond that, differences emerge. Service models have less in common with policy and intent than policy and intent do with each other.

Summarized differences:

- o A service model is a data model that is used to describe instances of services that are provided to customers. A service model has dependencies on lower level models (device and network models) when describing how the service is mapped onto underlying network and IT infrastructure. Instantiating a service model requires orchestration by a system; the logic for how to orchestrate/manage/provide the service model, and how to map it onto underlying resources, is not included as part of the model itself.
- o Policy is a set of rules, typically modeled around a variation of events/conditions/actions, used to express simple control loops that can be rendered by devices themselves, without requiring intervention by outside system. Policy lets users define what to do under what circumstances, but it does not specify a desired outcome.
- o Intent is a higher-level declarative policy that operates at the level of a network and services it provides, not individual devices. It is used to define outcomes and high-level operational goals, without the need to enumerate specific events, conditions,

and actions. Which algorithm or rules to apply can be automatically "learned/derived from intent" by the intent system. In the context of autonomic networking, ideally, intent is rendered by the network itself; also the dissemination of intent across the network and any required coordination between nodes is resolved by the network itself without the need for outside systems.

One analogy to capture the difference between policy and intent systems is that of Expert Systems and Learning Systems in the field of Artificial Intelligence. Expert Systems operate on knowledge bases with rules that are supplied by users. They are able to make automatic inferences based on those rules, but are not able to "learn" on their own. Learning Systems (popularized by deep learning and neural networks), on the other hand, are able to learn without depending on user programming. However, they do require a learning or training phase and explanations of actions that the system actually takes provide a different set of challenges.

5. Principles

The following operating principles allow characterizing the intent-based/-driven/-defined nature of a system.

1. Single Source of Truth (SSoT) and Single Version/View of Truth (SVoT). The SSoT is an essential component of an intent-based system as it enables several important operations. The set of validated intent expressions is the system's SSoT. SSoT and the records of the operational states enable comparing the intended state and actual state of the system and determining drift between them. SSoT and the drift information provide the basis for corrective actions. If the intent-based is equipped with prediction capabilities or means, it can further develop strategies to anticipate, plan and pro-actively act on the diverging trends with the aim to minimize their impact. Beyond providing a means for consistent system operation, SSoT also allows for better traceability to validate if/how the initial intent and associated business goals have been properly met, to evaluate the impacts of changes in the intent parameters and impacts and effects of the events occurring in the system. Single Version (or View) of Truth derives from the SSoT and can be used to perform other operations such as query, poll or filter the measured and correlated information to create so-called "views". These views can serve the operators and/or the users of the intent-based system. To create intents as single sources of truth, the intent-based system must follow well-specified and well-documented processes and models. In other contexts

[Lenrow15], SSoT is also referred to as the invariance of the intent.

2. One touch but not one shot. In an ideal intent-based system, the user expresses its intents in one form or another and then the system takes over all subsequent operations (one touch). A zero-touch approach could also be imagined in case where the intent-based system has the capabilities or means to recognize intentions in any form of data. However, the zero- or one-touch approach should not be mistaken the fact that reaching the state of a well-formed and valid intent expression is not a one-shot process. On the contrary, the interfacing between the user and the intent-based system could be designed as an interactive and interactive process. Depending on the level of abstraction, the intent expressions will initially contain more or less implicit parts, and unprecise or unknown parameters and constraints. The role of the intent-based system is to parse, understand and refine the intent expression to reach a well-formed and valid intent expression that can be further used by the system for the fulfillment and assurance operations. An intent refinement process could use a combination of iterative steps involving the user to validate the proposed refined intent and to ask the user for clarifications in case some parameters or variables could not be deduced or learned by the means of the system itself. In addition, the Intent-Based System will need to moderate between conflicting intent, helping users to properly choose between intent alternatives that may have different ramifications.
3. Autonomy and Oversight. A desirable goal for an intent-based system is to offer a high degree of flexibility and freedom on both the user side and system side, e.g. by giving the user the ability to express intents using its own terms, by supporting different forms of expression of intents and being capable of refining the intent expressions to well-formed and exploitable expressions. The dual principle of autonomy and oversight allows to operate a system that will have the necessary levels of autonomy to conduct its tasks and operations without requiring intervention of the user and taking its own decisions (within its areas of concern and span of control) as how to perform and meet the user expiations in terms of performance and quality, while at the same time providing the proper level of oversight to satisfy the user requirements for reporting and escalation of relevant information. to be added: description for feedback, reporting, guarantee scope (check points, guard rails, dynamically provisioned, context rich, regular operation vs. exception/ abnormal, information zoom in-out, and link to SVoT. Accountable for decisions and efficiency, late binding (leave it to the

system where to place functionality, how to accomplish certain goals).

4. Learning. An intent-based system is a learning system. By contrast to imperative type of system, such as Event-Condition-Action policy rules, where the user define beforehand the expected behavior of the system to various event and conditions, in an intent-based system, the user only declare what the system should achieve and not how to achieve these goals. There is thus a transfer of reasoning/rationality from the human (domain knowledge) to the system. This transfer of cognitive capability implies also the availability in the intent-based system of capabilities or means for learning, reasoning and knowledge representation and management. The learning abilities of an intent-based systems can apply to different tasks such as optimization of the intent rendering or intent refinement processes. The fact that an intent-based system is a continuously evolving system creates the condition for continuous learning and optimization. Other cognitive capabilities such as planning can also be leveraged in an intent-based system to anticipate or forecast future system state and response to changes in intents or network conditions and thus elaboration of plans to accommodate the changes while preserving system stability and efficiency in a trade-off with cost and robustness of operations. Cope with unawareness of users (smart recommendations).
5. Explainability. Need expressive network capabilities, requirements and constraints to be able to compose/decompose intents, map user's expectation to system capabilities. capability exposure. not just automation of steps that need to be taken, but of bridging the semantic gap between "intent" and actionable levels of instructions Context: multi providers, need discovery and semantic descriptions Explainability: why is a network doing what it is doing
6. Abstraction - users do not need to be concerned with how intent is achieved

Additional principles will be described in future revision of this document addressing aspects such as: Target groups not individual devices, agnostic to implementation details, user-friendly, user vocabulary vs. language of the device/network, explainability, validation and troubleshooting, how to resolve and point out conflicts (between intents), reconcile the reality of what is possible with the fiction of what the user would want, "moderate", awareness of operating within system boundaries, outcome-driven

((what not how, for the user); (what and how/where, for the operator).not imperative/instruction based.)).

The above principles will be further used to understand implications on the design of intent-based systems and their supporting architecture, and derive functional and operational requirements.

6. Lifecycle

Intent is subject to a lifecycle: it comes into being, may undergo changes over the course of time, and may at some point be retracted. This lifecycle is closely tied to various interconnection functions that are associated with the intent concept.

Figure 1 depicts an intent lifecycle and its main functions. The functions are divided into two functional (horizontal) planes and into three (vertical) spaces.

The functional planes provide structure for the main functional concerns that are associated with intent: how to fulfill intent, and how to assure it.

- o Fulfillment is concerned with the functions that take intent from its origination by a user (generally, an administrator of the responsible organization) to its realization in the network. This includes:
 - * Functions that recognize intent from interaction with the user and functions that allow users to refine their intent and articulate it in such ways so that it becomes actionable by an Intent-Based System. Those functions can involve unconventional human-machine interactions, in which a human will not simply give simple commands, but which may involve a human-machine dialog to provide clarifications, to explain ramifications and tradeoffs, and to facilitate refinements.
 - * Functions that translate user intent into courses of actions and requests to take against the network, which will be meaningful to network configuration and provisioning systems. Possibly, this includes learning functions and algorithms that optimize the courses of actions to take in order to result in the best outcomes, specifically in cases where multiple ways of achieving those outcomes are conceivable.
 - * Functions that perform and orchestrate the configuration and provisioning steps that were determined by the previous intent translation step.

- o Assurance is concerned with the functions that are necessary ensure that the network indeed complies with the desired intent once it has been fulfilled. This includes:
 - * Functions that monitor and observe the network and its exhibited behavior.
 - * Functions that assess and validate whether the observation indicate compliance with intent. This can include functions that perform analysis and aggregation of raw observation data.
 - * Functions that trigger corrective action as needed.
 - * Functions that abstract the observations and analysis results in a way that makes it possible for users to relate them to intent. In many cases, lower-level concepts such as detailed performance statistics and observations related to low-level settings need to be "up-leveled" to concepts the user can relate to and take action on.
 - * Functions that report intent compliance status and that provide adequate summarization and visualization to the user.

The spaces indicate the different perspectives and interactions with different roles that are involved to address the functions:

- o The user space involves the functions that interface the network and intent-based system with the human user. It involves the functions that allow users to articulate and the intent-based system to recognize that intent. It also involves the functions that report back the status of the network relative to the intent and that allow users to assess whether their intent is having the desired effect.
- o The translation or Intent-Based System (IBS) space involves the functions that bridge the gap between intent users and network operations. This includes the functions used to translate an intent into a course of action, the algorithms used to plan and optimize those courses of actions also in consideration of feedback, the functions to analyze and abstract observations to validate compliance with intent and take corrective actions as necessary.
- o The Network Operations space, finally, involves the traditional orchestration, configuration, monitoring, and measurement functions which are used to effectuate the rendered intent and observe its effects on the network.

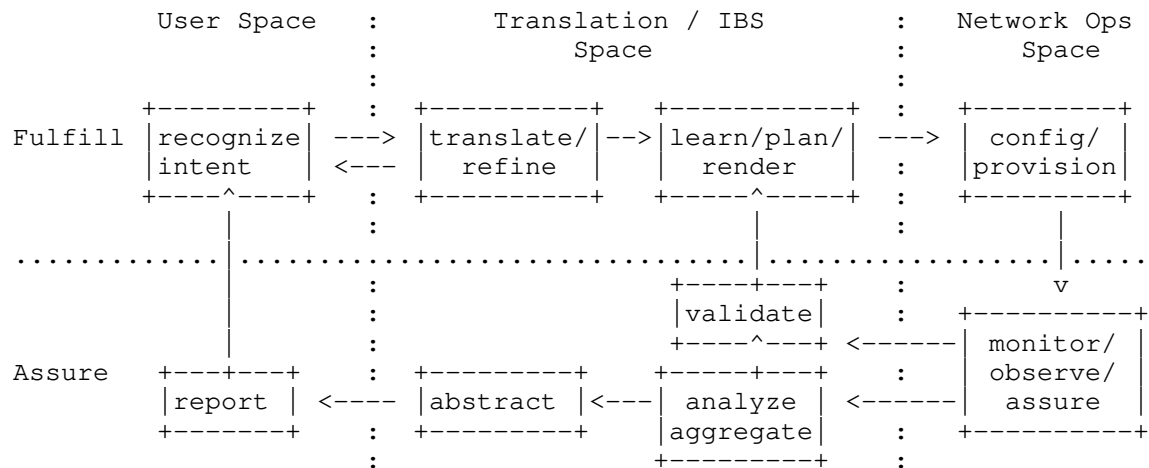


Figure 1: Intent Lifecycle

When inspecting the diagram carefully, it become apparent that the intent lifecycle in fact involves two cycles, or loops:

- o The "inner" intent control loop between IBS and Network Operations space is completely automated and does not involve any human in the loop. It involves automatic analysis and validation of intent based on observations from the network operations space, and feeding those observations into the function that plans the rendering of networking intent in order to make adjustments as needed in the configuration of the network.
- o The "outer" intent control loop involves also the user space and includes the user taking action and adjusting their intent based on feedback from the IBS.

Slight alternatives in intent lifecycles and the functions involved are conceivable. Figure 2 depicts one such alternative with an emphasis in intent fulfilment. (Todo: Intent attributes, intent states.) Distinguish flow from users to network, and from network to user.)

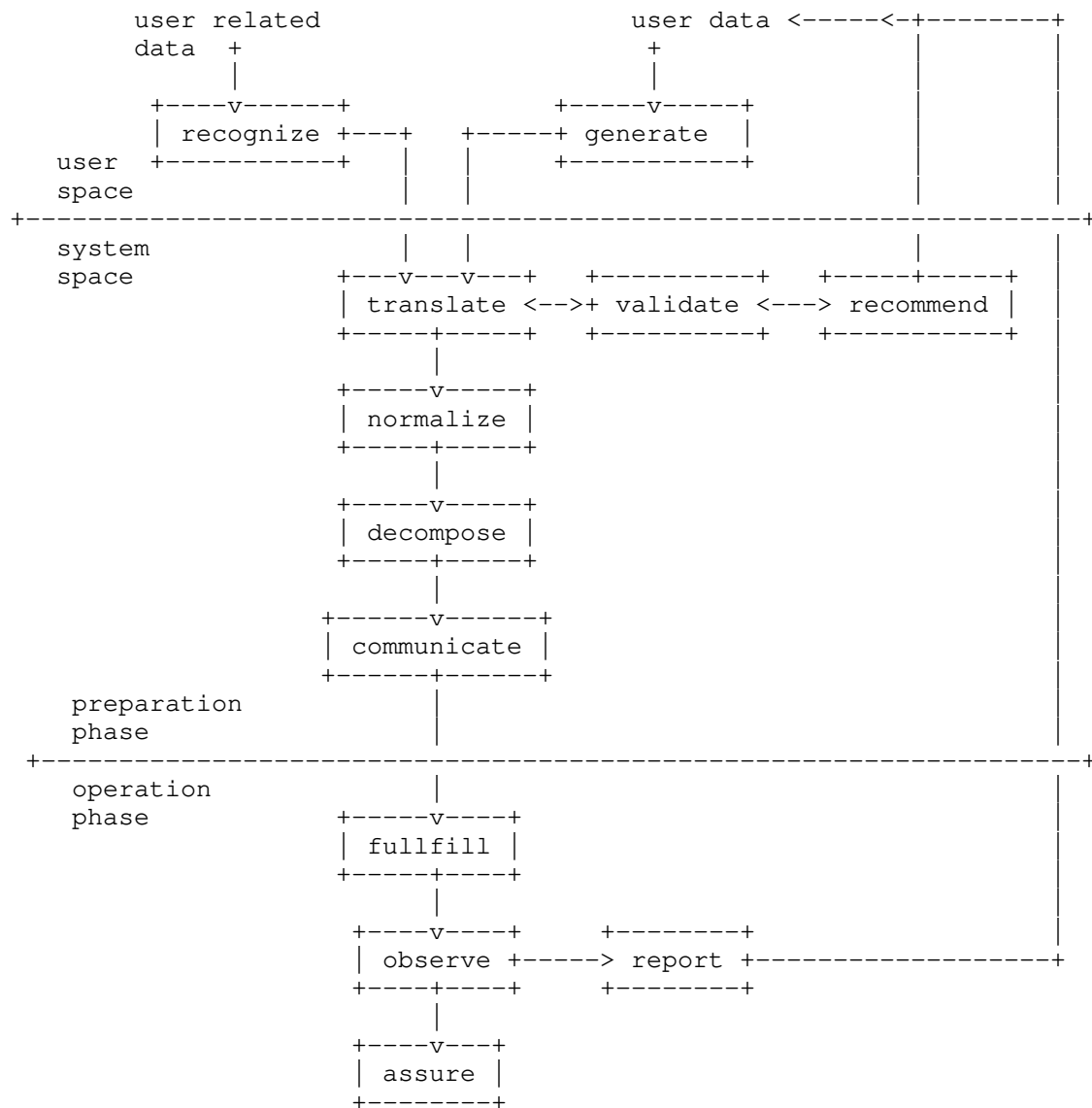


Figure 2: Intent Lifecycle (alt.)

7. Intent-Based Networking - Functionality

Intent-Based Networking involves a wide variety of functions which can be roughly divided into two categories:

- o Intent Fulfillment provides functions and interfaces that allow users to communicate intent to the network, and that orchestrates the intent, i.e. that breaks down intent abstractions into lower-level network and device abstractions and performs or coordinates the configuration operations across the network.
- o Intent Assurance provides functions and interfaces that allow users to validate and monitor that the network is indeed adhering to and complying with intent. Control plane or lower-level management operations can cause behavior that inadvertently conflicts with intent which was orchestrated earlier. Accordingly, "intent drift" may occur. Network operators need to be able to detect when such drift occurs, or is about to occur, and be provided with the necessary functions to resolve such conflicts. This can occur by either bringing the network back into compliance, or by articulating modifications to the original intent to moderate between conflicting interests.

The following sections provide a more comprehensive overview of those functions.

7.1. Intent Fulfillment

RBD

7.2. Intent Assurance

Ability to reason about system' state by employing closed-loop validation in the presence of an inevitable change is a fundamental property of an Intent Assurance part of an IBN system. Since service expectations are created during intent consumption and modeling phase, closed-loop intent validation should start immediately, with the service instantiation. Telemetry consumed could then be enriched with an additional context and must always be processed in context of the Intent it has been instantiated. Direct relationship between the Intent and telemetry gathered enables correlation between changes in states and the Intent and provides contextual base for reasoning about the changes.

8. Items for Discussion

Arguably, given the popularity of the term intent, its use could be broadened to encompass also known concepts ("intent-washing"). For example, it is conceivable to introduce intent-based terms for various concepts that, although already known, are related to the context of intent. Each of those terms could then designate an intent subcategory, for example:

- o Operational Intent: defines intent related to operational goals of an operator; corresponds to the original "intent" term.
- o Rule Intent: a synonym for policy rules regarding what to do when certain events occur.
- o Service intent: a synonym for customer service model [RFC8309].
- o Flow Intent: A synonym for a Service Level Objective for a given flow.

Whether to do so is an item for discussion by the Research Group.

9. IANA Considerations

Not applicable

10. Security Considerations

Not applicable

11. References

11.1. 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>>.

11.2. Informative References

- [Boutaba07] Boutaba, R. and I. Aib, "Policy-Based Management: A Historical perspective. Journal of Network and Systems Management (JNSM), Springer, Vol. 15 (4).", December 2007.
- [eTOM] TMForum, "GB 921 Business Process Framework, Release 17.0.1.", February 2018.
- [I-D.ietf-teas-te-service-mapping-yang] Lee, Y., Dhody, D., Fioccola, G., WU, Q., Ceccarelli, D., and J. Tantsura, "Traffic Engineering (TE) and Service Mapping Yang Model", draft-ietf-teas-te-service-mapping-yang-02 (work in progress), September 2019.
- [Lenrow15] Lenrow, D., "Intent As The Common Interface to Network Resources, Intent Based Network Summit 2015 ONF Boulder: IntentNBI", February 2015.
- [RFC7575] Behringer, M., Pritikin, M., Bjarnason, S., Clemm, A., Carpenter, B., Jiang, S., and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals", RFC 7575, DOI 10.17487/RFC7575, June 2015, <<https://www.rfc-editor.org/info/rfc7575>>.
- [RFC8299] Wu, Q., Ed., Litkowski, S., Tomotaki, L., and K. Ogaki, "YANG Data Model for L3VPN Service Delivery", RFC 8299, DOI 10.17487/RFC8299, January 2018, <<https://www.rfc-editor.org/info/rfc8299>>.
- [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>>.
- [RFC8345] Clemm, A., Medved, J., Varga, R., Bahadur, N., Ananthakrishnan, H., and X. Liu, "A YANG Data Model for Network Topologies", RFC 8345, DOI 10.17487/RFC8345, March 2018, <<https://www.rfc-editor.org/info/rfc8345>>.
- [Sloman94] Sloman, M., "Policy Driven Management for Distributed Systems. Journal of Network and Systems Management (JNSM), Springer, Vol. 2 (4).", December 1994.
- [Strassner03] Strassner, J., "Policy-Based Network Management. Elsevier.", 2003.

[TR523] Foundation, O. N., "Intent NBI - Definition and Principles. ONF TR-523.", October 2016.

Authors' Addresses

Alexander Clemm
Futurewei
2330 Central Expressway
Santa Clara, CA 95050
USA

Email: ludwig@clemm.org

Laurent Ciavaglia
Nokia
Route de Villejust
Nozay 91460
FR

Email: laurent.ciavaglia@nokia.com

Lisandro Zambenedetti Granville
Federal University of Rio Grande do Sul (UFRGS)
Av. Bento Goncalves
Porto Alegre 9500
BR

Email: granville@inf.ufrgs.br

Jeff Tantsura
Apstra, Inc.

Email: jefftant.ietf@gmail.com