Network Working Group X. Cai
Internet-Draft C. Meirosu
Intended status: Informational G. Mirsky
Expires: April 17, 2016 Ericsson
October 15, 2015

Recursive Monitoring Language in Network Function Virtualization (NFV) Infrastructures
draft-cai-nfvrg-recursive-monitor-00

Abstract

Network Function Virtualization (NFV) poses a number of monitoring challenges; one potential solution to these challenges is a recursive monitoring language. This document presents a set of requirements for such a recursive monitoring language.

Table of Contents

1. Introduction

This document discusses the recursive monitoring query language to support NFV infrastructures (e.g., defined in ETSI [ETSI-ARC] and UNIFY [I-D.unify-nfvrg-recursive-programming]). A network service can be constructed of Virtual Network Functions (VNFs) or Physical Network Functions (PNFs) interconnected through a Network Function Forwarding Graph (NF-FG). A single VNF, in turn, can consist of interconnected elements; in other words, NF-FGs can be nested.

Service operators and developers are interested in monitoring the performance of a service contained within an NF-FG (as above) or any part of it. For example, an operator may want to measure the CPU or memory usage of an entire network service and the network delay cross a VNF which consists of multiple VMs, instead of only individual virtual or physical entities.

In existing systems, this is usually done by mapping the performance metrics of VNFs to primitive network functions or elements, statically and manually when the virtualized service is deployed. However, in the architecture defined in ETSI [ETSI-ARC] and UNIFY [I-D.unify-nfvrg-recursive-programming] a multi-layer hierarchical architecture is adopted, and the VNF and associated resources,

expressed NF-FGs, may be composed recursively in different layers of the architecture. This will pose greater challenges for performance queries for a specific service, as the mapping of performance metrics from the service layer (highest layer) to the infrastructure (lowest layer) is more complex than an infrastructure with a single layer of orchestration. We argue that it is important to have an automatic and dynamic way to decompose performance queries in this environment in a recursive way, following the different abstraction levels expressed in the NF-NFs at hierarchical architecture layers. Hence, we propose using a declarative language such as Datalog [Green-2013] to perform recursive queries based on input in form of the resource graph depicted as NF-FG. By reusing the NF-FG models and monitoring database already deployed in NFV infrastructure, the language can hide the complexity of the multilayer network architecture with limited extra effort and resources. Even for single layer NFV architectures, using such language can simplify performance queries and enable a more dynamic performance decomposition and aggregation for the service layer.

Recursive query languages can support many DevOps [I-D.unify-nfvrg-devops] processes, most notably observability and troubleshooting tasks relevant for both operators and developer roles, e.g. for high-level troubleshooting where various information from different sources need to be retrieved. Additionally, the query language might be used by specific modules located in the control and orchestration layers, e.g. a module realizing infrastructure embedding of NF-FGs might query monitoring data for an up-to-date picture of current resource usage. Also scaling modules of specific network functions might take advantage of the flexible query engine pulling of monitoring information on demand (e.g. resource usage, traffic trends, etc.), as complement to relying on devices and/or elements to push this information based on pre-defined thresholds.

1.1. Conventions used in this document

1.1.1. Terminology

ETSI - European Telecommunication Standards Institute

NF-FG - Network Function Forwarding Graph

NFV - Network Function Virtualization

PNF - Physical Network Function

SG - Service Graph

VNF - Virtual Network Function

1.1.2. Requirements Language

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 [RFC2119].

2. Requirements towards NFV Monitoring Language

Following are the requirements for a language to express constructs and actions of monitoring NFV infrastructures:

- The network service MAY consist of VNFs which contain inter-connected elements and be described by nested NF-FGs. The language MUST support recursive query.

- The language is used by the service operators or developers to monitor the high-level performance of the network service. Declarative language could provide better description on the monitoring task rather the procedure than imperative language. The language MUST be declarative.

3. Sample Use Cases

In Figure 1, the Service Graph (SG) and corresponding NF-FGs of a network service is illustrated. The service consists of two Network Functions NF1 and NF2, which consists of (VNF1-1, VNF1-2) and (VNF2-1, VNF2-2) respectively. In VNF1-1 and VNF2-2 there are recursively nested VNFs VNF1-3 and VNF2-3.

```
                           +---+        +---+
           +-------------- |NF1|--------|NF2| ---------------+
           |               +-+-+        +-+-+                |
           |                 |            |                  |
           |                 |            |                  |
   +-------+-------+ +------------+ +--------------+ +------------------+
   |     VNF1-1    | |   VNF1-2   | |    VNF2-1    | |      VNF2-2      |
   |+------+  +----+ +----+ +----+ | +---+  +---+ | | +---+  +-------+ |
   ||VNF1-3|  |vm1|| ||vm2| |vm3|| | |vm4|  |vm5| | | |vm6|  |VNF2-3| |
   |-------+  |----| |----+ |----| | |---+  |---+ | | |---+  +-------+ |
   +---------------+ +------------+ +--------------+ +------------------+
      |   |                                                  |     |
   +--++  +---+                                           +--++  +-+--+
   |vm7|  |vm8|                                           |vm9|  |vm10|
   +---+  +---+                                           +---+  +-----+
```

Figure 1: The sample NF-FG of network service

Two use cases of the recursive monitoring query are described below.

First, consider the use case where an operator of the network service wants to query the end to end delay from network function NF1 to Network Function NF2 in the service graph. Here the end to end delay of two network functions are defined as the delay between ingress node of source network function and egress node of destination network function. After running a querying script, the delay between NF1 and NF2 in service layer should be mapped recursively to the delay between two specific virtual machines (vm7 and vm10) in the NFV infrastructure.

Second, consider the use case where an operator wants to measure the CPU usage of network function NF1 in order to dynamically scale in/out this function. Several types of CPU usage of a network function can be defined. For example, average CPU usage is the average value of measured CPU usage of all nodes belongs to the network function. Maximum CPU usage is the measured usage of the node that has the highest CPU load. To get either the average or maximum CPU usage, the query language to recursively identify all nodes (i.e., vm1, vm2, vm3, vm7 and vm8) of NF1, then retrieve the measured CPU usage of these nodes from somewhere and return the mean or maximum value to the operator.

4. Overview of the Recursive Language

In this section we describe the recursive monitoring language. The query language proposed here is based on Datalog, which is a declarative logic programming language that provides recursive query

capability. Datalog has been used in cloud computing in recent years, e.g., the OpenStack [OpenStack] policy engine Congress [OpenStack-Congress].

As like other Datalog based language, the recursive monitoring query program consists of a set of declarative Datalog rules and a query. A rule has the form:

```
h <= p1, p2, ..., pn
```

which can be defined as "p1 and p2 and ... and pn implies h". "h" is the head of the rule, and "p1, p2, ..., pn" is a list of literals that constitutes the body of the rule. Literals "p(x1, ..., xi, ..., xn)" are either predicates applied to arguments "xi" (variables and constants), or function symbols applied to arguments. The program is said to be recursive if a cycle exists through the predicates, i.e., predicate appearing both in the head and body of the same rule. The order in which the rules are presented in a program is semantically irrelevant. The commas separating the predicates in a rule are logical conjuncts (AND); the order in which predicates appear in a rule body has no semantic significance, i.e. no matter in what order rules been processed, the result is atomic, i.e. the same. The names of predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an upper-case letter. A variable appearing in the head is called distinguished variable while a variable appearing in the body is called non-distinguished variable. The head is true of the distinguished variables if there exist values of the non-distinguished variables that make all sub goals of the body true. In every rule, each variable stands for the same value. Thus, variables can be considered as placeholders for values. Possible values are those that occur as constants in some rule/fact of the program itself. In the program, a query is of the form "query(m, y1, ..., yn)", in which "query" is a predicate contains arguments "m" and "yi". "m" represents the monitoring function to be queried, e.g., end to end delay, average CPU usage, and etc. "yi" is the arguments for the query function. The meaning of a query given a set of Datalog rules and facts is the set of all facts of query() that are given or can be inferred using the rules in the program. The predicates can be divided into two categories: extensional database predicates (EDB predicates), which contains ground facts, meaning it only has constant arguments; and intentional database predicates (IDB predicates), which correspond to derived facts computed by Datalog rules.

In order to perform a recursive monitoring query, the resource graph described in NF-FG needs be transformed so it is represented as a set of Datalog ground facts which are used by the rules in the program.

The following keywords can be defined to represent the NF-FG graph into Datalog facts, which are then used in the query scripts:

sub(x, y) which represents 'y' is an element of the directly descend sub-layer of 'x';

link(x, y) which represents that there is a direct link between elements 'x' and 'y';

node(z) which represents an node in NF-FG.

It should be noted that more keywords can be defined in order to describe other properties of NF-FG.

In addition, a set of functions call can be defined in order to support the monitoring query. The function call will start with "fn_" in the syntax and may include 'boolean' predicates, arithmetic computations and some other simple operation. The function calls can be provided by the query engine or developers.

If the sub NF-FGs of a network service are provided by different NFV infrastructure provider and not available to the provider who like to measure some aspect of the NF-FG due to some reason, e.g., security, additional extensions to the language and query engine would be required (this is called a distributed query). This scenario is not considered in this draft; left for further study.

5. Formal Syntax

The following syntax specification describes the Datalog based reursive monitoring language and uses the augmented Backus-Naur Form (BNF) as described in [RFC2234].

```
<program>         ::= <statement>*
<statement>       ::= <rule> | < fact>
<rule>            ::= [<rule-identifier>] <head> <= <body>
<fact>            ::= [<fact-identifier>]<clause> |
                       <fact_predicate>(<terms>)
<head>            ::= <clause>
<body>            ::= <clause>
<clause>          ::= <atom> | <atom>, <clause>
<atom>            ::= <predicate> ( <terms> )
<predicate>       ::= <lowercase-letter><string>
<fact_predicate>  ::= ("sub"; | "node" |
                       "link")( <terms> )
<terms>           ::= <term> | <term>, <terms>
<term>            ::= <VARIABLE> | <constant>
<constant>        ::= <lowercase-letter><string>
<VARIABLE>        ::= <Uppercase-letter><string>
<fact-identifier> ::= "F"<integer>
<rule-identifier> ::= "R"<integer>
```

6. Requirements for Using the Language

To utilize the recursive monitoring language a query engine has to be deployed into NFV infrastructure. Some basic functions are required for the query engine.

The query engine MUST provide the capability to parse and interpret the query scripts which are written with the language.

The query engine MUST be able to retrieve the NF-FG created by NFV infrastructure and translate them into Datalog based ground facts.

The query engine MUST be able to query the database in which the monitoring results of primitive metric are stored.

An interface between query engine and the users of the language (e.g., developer or network service operator) MUST be defined to exchange the query scripts and query results.

7. Sample Query Scripts

According to the defined language, the sample query scripts for the above mentioned use cases are illustrated in this section. Some example query scripts are illustrated in this section.

7.1. Query End to End Delay Between Network Functions

Two kinds of delay between network functions are discussed here: end-to-end delay and hop-by-hop delay. Here end to end delay is defined as the delay between the ingress node in the lowest layer of the source network function and the egress node in the lowest layer of the destination network function. And the hop by hop delay is defined as the aggregation of the delay of each segment which consists of the path from the source to the destination network function.

The scripts to query the end to end delay from NF1 to NF2 as illustrated in Figure 1 contains both the ground facts and IDB predicates:

```
F1: sub(NF1, VNF1-3, vm1, vm2, vm3), sub(NF2, vm4, vm5, vm6, VNF1-3),
sub(VNF1-3, vm7, vm8), sub(VNF1-3, vm9, vm10)
F2: node(NF1, NF2, VNF1-3, vm1, vm2, vm3, vm4, vm5, vm6, VNF1-3,
vm7, vm8, vm9, vm10)
F3: link(NF1, NF2), link (VNF1-3, vm1), link(vm2, vm3), link(vm3, vm4),
link(vm4,vm5), link(vm5,vm6), link(vm6, VNF1-3), link(vm7, vm8),
link(vm9, vm10)
R1: child(X,Y) <= sub(X,Z), child(Z,Y)
R2: child(X,Y) <= sub(X,Y)
R3: leaf(X,Y) <= child(X,Y), ~sub(Y,Z)
R4: in_leaf(X, Y) <= leaf(X, Y) & ~link(M, Y)
R5: out_leaf(X, Y) <= leaf(X, Y) & ~link(Y, M)
R6: e2e_delay(S,D,P) <= link(S,D), P == f_e2e_delay(in_leaf(S,Y),
out_leaf(D,Z))
query(e2e_delay, NF1, NF2)
```

F1-F3 are used to translate the NF-FG in Figure x into ground facts. R1-R5 are used to traversal the NF-FG recursively to get the ingress node of VNF1 and egress node of VNF2. R1-R2 can recursively traversal the graphs and figure out all child nodes (i.e., VNF1-1, VNF1-3, VNF1-2, vm1, vm2, vm3, vm7, vm8, VNF2-1, VNF2-2, VNF2-3, vm4, vm5, vm6, vm9, vm10 in Figure 1). R3 is used to figure out all leaf nodes (i.e., virtual machines). In the example, they include all virtual machines. R4 and R5 are used to get the ingress and egress nodes of NF1 and NF2 respectively, i.e., vm7 and vm10. In R6 the delay for given source and destination network functions is measured by function f_e2e_delay. R1-R6 can be stored into a library of the query engine as a template e2e_delay, so that the receivers only need send a simple query request, e.g. e2e_delay NF1 NF2, to the query engine to measure the end to end delay between NF1 and NF2.

7.2. Query the CPU Usage of Network Functions

Below are scripts to query the CPU usage (maximum and average usage) of a given network function:

```
F1: sub(NF1, VNF1-3, vm1, vm2, vm3), sub(NF2, vm4, vm5, vm6, VNF1-3),
sub(VNF1-3, vm7, vm8), sub(VNF1-3, vm9, vm10)
F2: node(NF1, NF2, VNF1-3, vm1, vm2, vm3, vm4, vm5, vm6, VNF1-3, vm7,
vm8, vm9, vm10)
R1: child(X,Y) <= sub(X,Z), child(Z,Y)
R2: child(X,Y) <= sub(X,Y)
R3: leaf(X,Y) <= child(X,Y), ~sub(Y,Z)
R4: max_cpu(X,C) <= leaf(X,Y), C == f_max_cpu(leaf(X,Y))
R5: mean_cpu (X,C) <= leaf(X, Y), C == f_mean_cpu(leaf(X,Y))
Query(max_cpu, NF1)
```

F1-F2 are used to translate the NF-FG in Figure x into ground facts. R1-R3 are used to traversal the NF-FG recursively to get all child nodes of NF1 in Figure x. R1-R2 recursively traversal the graphs and figure out all child nodes of NF1(i.e., VNF1-1, VNF1-3, VNF1-2, vm1, vm2, vm3, vm7, vm8). R3 is used to figure out all leaf nodes of NF1(i.e., vm1, vm2, vm3, vm7, vm8). In R4, the maximum CPU usage is calculated by function f_max_cpu. In R5, the average CPU usage is calculated by function f_mean_cpu.

Here only the query scripts for network delay and CPU usage are illustrated. But the language can also be applied to other performance metrics like throughput and etc.

8. IANA Considerations

TBD

9. Security Considerations

TBD

10. Acknowledgements

Authors deeply appreciate thorough review and insightful comments by Russ White.

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, <http://www.rfc-editor.org/info/rfc2119>.

11.2. Informative References

[ETSI-ARC]
"Architectural Framework v1.1.1", ETSI , October 2013.

[Green-2013]
"Green, T. J., S. Huang, B.T. Loo, and W. Zhou, Datalog and Recursive Query Processing", Foundations and Trends in Databases Vol. 5, No. 2, November 2013.

[I-D.unify-nfvrg-devops]
Meirosu, C., Manzalini, A., Kim, J., Steinert, R., Sharma, S., Marchetto, G., Papafili, I., Pentikousis, K., and S. Wright, "DevOps for Software-Defined Telecom Infrastructures", draft-unify-nfvrg-devops-02 (work in progress), July 2015.

[I-D.unify-nfvrg-recursive-programming]
Szabo, R., Qiang, Z., and M. Kind, "Towards recursive virtualization and programming for network and cloud resources", draft-unify-nfvrg-recursive-programming-01 (work in progress), July 2015.

[OpenStack]
"OpenStack (online)", http://www.openstack.org/ .

[OpenStack-Congress]
"OpenStack Congress (online)", https://wiki.openstack.org/wiki/Congress .

[RFC2234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, DOI 10.17487/RFC2234, November 1997, <http://www.rfc-editor.org/info/rfc2234>.

Authors' Addresses

Xuejun Cai
Ericsson

Email: xuejun.cai@ericsson.com

Catalin Meirosu
Ericsson

Email: catalin.meirosu@ericsson.com

Greg Mirsky
Ericsson

Email: gregory.mirsky@ericsson.com