Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 8, 2020

B. Carpenter
Univ. of Auckland
L. Ciavaglia
Nokia
S. Jiang
Huawei Technologies Co., Ltd
P. Peloso
Nokia
July 7, 2019

Guidelines for Autonomic Service Agents
draft-carpenter-anima-asa-guidelines-07

Abstract

This document proposes guidelines for the design of Autonomic Service Agents for autonomic networks. It is based on the Autonomic Network Infrastructure outlined in the ANIMA reference model, making use of the Autonomic Control Plane and the Generic Autonomic Signaling Protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document proposes guidelines for the design of Autonomic Service Agents (ASAs) in the context of an Autonomic Network (AN) based on the Autonomic Network Infrastructure (ANI) outlined in the ANIMA reference model [I-D.ietf-anima-reference-model]. This infrastructure makes use of the Autonomic Control Plane (ACP) [I-D.ietf-anima-autonomic-control-plane] and the Generic Autonomic Signaling Protocol (GRASP) [I-D.ietf-anima-grasp].

There is a considerable literature about autonomic agents with a variety of proposals about how they should be characterized. Some examples are [DeMola06], [Huebscher08], [Movahedi12] and [GANA13]. However, for the present document, the basic definitions and goals for autonomic networking given in [RFC7575] apply . According to RFC 7575, an Autonomic Service Agent is "An agent implemented on an autonomic node that implements an autonomic function, either in part (in the case of a distributed function) or whole."

ASAs must be distinguished from other forms of software component. They are components of network or service management; they do not in themselves provide services. For example, the services envisaged for network function virtualisation [RFC8568] or for service function chaining [RFC7665] might be managed by an ASA rather than by traditional configuration tools.

The reference model [I-D.ietf-anima-reference-model] expands this by adding that an ASA is "a process that makes use of the features provided by the ANI to achieve its own goals, usually including interaction with other ASAs via the GRASP protocol [I-D.ietf-anima-grasp] or otherwise. Of course it also interacts with the specific targets of its function, using any suitable mechanism. Unless its function is very simple, the ASA will need to handle overlapping asynchronous operations. This will require either a multi-threaded implementation, or a logically equivalent event loop structure. It may therefore be a quite complex piece of software in its own right, forming part of the application layer above the ANI."

There will certainly be very simple ASAs that manage a single objective in a straightforward way and do not asynchronous operations. In such a case, many aspects of the current document do not apply. However, in general a basic property of an ASA is that it is a relatively complex software component that will in many cases control and monitor simpler entities in the same host or elsewhere. For example, a device controller that manages tens or hundreds of simple devices might contain a single ASA.

The remainder of this document offers guidance on the design of such ASAs.

2. Logical Structure of an Autonomic Service Agent

As mentioned above, all but the simplest ASAs will need to suport asynchronous operations. Not all programming environments explicitly support multi-threading. In that case, an 'event loop' style of implementation should be adopted, in which case each thread would be implemented as an event handler called in turn by the main loop. For this, the GRASP API (Section 3.3) must provide non-blocking calls.

If necessary, the GRASP session identifier will be used to distinguish simultaneous operations.

A typical ASA will have a main thread that performs various initial housekeeping actions such as:

- Obtain authorization credentials.
- Register the ASA with GRASP.
- Acquire relevant policy parameters.
- Define data structures for relevant GRASP objectives.
- Register with GRASP those objectives that it will actively manage.
- Launch a self-monitoring thread.
- Enter its main loop.

The logic of the main loop will depend on the details of the autonomic function concerned. Whenever asynchronous operations are required, extra threads will be launched, or events added to the event loop. Examples include:

- Repeatedly flood an objective to the AN, so that any ASA can receive the objective's latest value.
- Accept incoming synchronization requests for an objective managed by this ASA.
- Accept incoming negotiation requests for an objective managed by this ASA, and then conduct the resulting negotiation with the counterpart ASA.
- Manage subsidiary non-autonomic devices directly.

These threads or events should all either exit after their job is done, or enter a wait state for new work, to avoid blocking others unnecessarily.

According to the degree of parallelism needed by the application, some of these threads or events might be launched in multiple instances. In particular, if negotiation sessions with other ASAs are expected to be long or to involve wait states, the ASA designer might allow for multiple simultaneous negotiating threads, with appropriate use of queues and locks to maintain consistency.

The main loop itself could act as the initiator of synchronization requests or negotiation requests, when the ASA needs data or resources from other ASAs. In particular, the main loop should watch for changes in policy parameters that affect its operation. It should also do whatever is required to avoid unnecessary resource consumption, such as including an arbitrary wait time in each cycle of the main loop.

The self-monitoring thread is of considerable importance. Autonomic service agents must never fail. To a large extent this depends on careful coding and testing, with no unhandled error returns or exceptions, but if there is nevertheless some sort of failure, the self-monitoring thread should detect it, fix it if possible, and in the worst case restart the entire ASA.

Appendix B presents some example logic flows in informal pseudocode.

3. Interaction with the Autonomic Networking Infrastructure

3.1. Interaction with the security mechanisms

An ASA by definition runs in an autonomic node. Before any normal ASAs are started, such nodes must be bootstrapped into the autonomic network's secure key infrastructure in accordance with [I-D.ietf-anima-bootstrapping-keyinfra]. This key infrastructure will be used to secure the ACP (next section) and may be used by ASAs to set up additional secure interactions with their peers, if needed.

Note that the secure bootstrap process itself may include special-purpose ASAs that run in a constrained insecure mode.

3.2. Interaction with the Autonomic Control Plane

In a normal autonomic network, ASAs will run as clients of the ACP. It will provide a fully secured network environment for all communication with other ASAs, in most cases mediated by GRASP (next section).

Note that the ACP formation process itself may include special-purpose ASAs that run in a constrained insecure mode.

3.3. Interaction with GRASP and its API

GRASP [I-D.ietf-anima-grasp] is expected to run as a separate process with its API [I-D.ietf-anima-grasp-api] available in user space. Thus ASAs may operate without special privilege, unless they need it for other reasons. The ASA's view of GRASP is built around GRASP objectives (Section 5), defined as data structures containing

administrative information such as the objective's unique name, and its current value. The format and size of the value is not restricted by the protocol, except that it must be possible to serialise it for transmission in CBOR [RFC7049], which is no restriction at all in practice.

The GRASP API should offer the following features:

- o Registration functions, so that an ASA can register itself and the objectives that it manages.
- o A discovery function, by which an ASA can discover other ASAs supporting a given objective.
- o A negotiation request function, by which an ASA can start negotiation of an objective with a counterpart ASA. With this, there is a corresponding listening function for an ASA that wishes to respond to negotiation requests, and a set of functions to support negotiating steps.
- o A synchronization function, by which an ASA can request the current value of an objective from a counterpart ASA. With this, there is a corresponding listening function for an ASA that wishes to respond to synchronization requests.
- o A flood function, by which an ASA can cause the current value of an objective to be flooded throughout the AN so that any ASA can receive it.

For further details and some additional housekeeping functions, see [I-D.ietf-anima-grasp-api].

This API is intended to support the various interactions expected between most ASAs, such as the interactions outlined in Section 2. However, if ASAs require additional communication between themselves, they can do so using any desired protocol. One option is to use GRASP discovery and synchronization as a rendez-vous mechanism between two ASAs, passing communication parameters such as a TCP port number via GRASP. As noted above, either the ACP or in special cases the autonomic key infrastructure will be used to secure such communications.

3.4. Interaction with policy mechanism

At the time of writing, the policy (or "Intent") mechanism for the ANI is undefined. It is expected to operate by an information distribution mechanism that can reach all autonomic nodes, and therefore every ASA. However, each ASA must be capable of operating

"out of the box" in the absence of locally defined policy, so every ASA implementation must include carefully chosen default values and settings for all policy parameters.

4. Interaction with Non-Autonomic Components

An ASA, to have any external effects, must also interact with non-autonomic components of the node where it is installed. For example, an ASA whose purpose is to manage a resource must interact with that resource. An ASA whose purpose is to manage an entity that is already managed by local software must interact with that software. This is stating the obvious, and the details are specific to each case, but it has an important security implication. The ASA might act as a loophole by which the managed entity could penetrate the security boundary of the ANI. The ASA must be designed to avoid such loopholes, and should if possible operate in an unprivileged mode.

In an environment where systems are virtualized and specialized using techniques such as network function virtualization or network slicing, there will be a design choice whether ASAs are deployed once per physical node or once per virtual context. A related issue is whether the ANI as a whole is deployed once on a physical network, or whether several virtual ANIs are deployed. This aspect needs to be considered by the ASA designer.

5. Design of GRASP Objectives

The general rules for the format of GRASP Objective options, their names, and IANA registration are given in [I-D.ietf-anima-grasp]. Additionally that document discusses various general considerations for the design of objectives, which are not repeated here. However, we emphasize that the GRASP protocol does not provide transactional integrity. In other words, if an ASA is capable of overlapping several negotiations for a given objective, then the ASA itself must use suitable locking techniques to avoid interference between these negotiations. For example, if an ASA is allocating part of a shared resource to other ASAs, it needs to ensure that the same part of the resource is not allocated twice. This might impact the design of the objective as well as the logic flow of the ASA.

In particular, if 'dry run' mode is defined for the objective, its specification, and every implementation, must consider what state needs to be saved following a dry run negotiation, such that a subsequent live negotiation can be expected to succeed. It must be clear how long this state is kept, and what happens if the live negotiation occurs after this state is deleted. An ASA that requests a dry run negotiation must take account of the possibility that a successful dry run is followed by a failed live negotiation. Because

of these complexities, the dry run mechanism should only be supported by objectives and ASAs where there is a significant benefit from it.

The actual value field of an objective is limited by the GRASP protocol definition to any data structure that can be expressed in Concise Binary Object Representation (CBOR) [RFC7049]. For some objectives, a single data item will suffice; for example an integer, a floating point number or a UTF-8 string. For more complex cases, a simple tuple structure such as [item1, item2, item3] could be used. Nothing prevents using other formats such as JSON, but this requires the ASA to be capable of parsing and generating JSON. The formats acceptable by the GRASP API will limit the options in practice. A fallback solution is for the API to accept and deliver the value field in raw CBOR, with the ASA itself encoding and decoding it via a CBOR library.

Note that a mapping from YANG to CBOR is defined by [I-D.ietf-core-yang-cbor]. Subject to the size limit defined for GRASP messages, nothing prevents objectives using YANG in this way.

6. Life Cycle

Autonomic functions could be permanent, in the sense that ASAs are shipped as part of a product and persist throughout the product's life. However, a more likely situation is that ASAs need to be installed or updated dynamically, because of new requirements or bugs. Because continuity of service is fundamental to autonomic networking, the process of seamlessly replacing a running instance of an ASA with a new version needs to be part of the ASA's design.

The implication of service continuity on the design of ASAs can be illustrated along the three main phases of the ASA life-cycle, namely Installation, Instantiation and Operation.

```
                     +--------------+
   Undeployed ------>|              |------> Undeployed
                     |  Installed   |
                 +-->|              |---+
        Mandate  |   +--------------+   | Receives a
      is revoked |   +--------------+   |  Mandate
                 +---|              |<--+
                     | Instantiated |
                 +-->|              |---+
             set |   +--------------+   | set
            down |   +--------------+   | up
                 +---|              |<--+
                     |  Operational |
                     |              |
                     +--------------+
```

Figure 1: Life cycle of an Autonomic Service Agent

6.1. Installation phase

Before being able to instantiate and run ASAs, the operator must first provision the infrastructure with the sets of ASA software corresponding to its needs and objectives. The provisioning of the infrastructure is realized in the installation phase and consists in installing (or checking the availability of) the pieces of software of the different ASA classes in a set of Installation Hosts.

There are 3 properties applicable to the installation of ASAs:

The dynamic installation property allows installing an ASA on
   demand, on any hosts compatible with the ASA.

The decoupling property allows controlling resources of a NE from a
   remote ASA, i.e. an ASA installed on a host machine different from
   the resources' NE.

The multiplicity property allows controlling multiple sets of
   resources from a single ASA.

These three properties are very important in the context of the installation phase as their variations condition how the ASA class could be installed on the infrastructure.

6.1.1. Installation phase inputs and outputs

Inputs are:

[ASA class of type_x] that specifies which classes ASAs to install,

[Installation_target_Infrastructure] that specifies the candidate Installation Hosts,

[ASA class placement function, e.g. under which criteria/constraints as defined by the operator]
that specifies how the installation phase shall meet the operator's needs and objectives for the provision of the infrastructure. In the coupled mode, the placement function is not necessary, whereas in the decoupled mode, the placement function is mandatory, even though it can be as simple as an explicit list of Installation hosts.

The main output of the installation phase is an up-to-date directory of installed ASAs which corresponds to [list of ASA classes] installed on [list of installation Hosts]. This output is also useful for the coordination function and corresponds to the static interaction map (see next section).

The condition to validate in order to pass to next phase is to ensure that [list of ASA classes] are well installed on [list of installation Hosts]. The state of the ASA at the end of the installation phase is: installed. (not instantiated). The following commands or messages are foreseen: install(list of ASA classes, Installation_target_Infrastructure, ASA class placement function), and un-install (list of ASA classes).

6.2. Instantiation phase

Once the ASAs are installed on the appropriate hosts in the network, these ASA may start to operate. From the operator viewpoint, an operating ASA means the ASA manages the network resources as per the objectives given. At the ASA local level, operating means executing their control loop/algorithm.

But right before that, there are two things to take into consideration. First, there is a difference between 1. having a piece of code available to run on a host and 2. having an agent based on this piece of code running inside the host. Second, in a coupled case, determining which resources are controlled by an ASA is straightforward (the determination is embedded), in a decoupled mode determining this is a bit more complex (hence a starting agent will have to either discover or be taught it).

The instantiation phase of an ASA covers both these aspects: starting the agent piece of code (when this does not start automatically) and determining which resources have to be controlled (when this is not obvious).

6.2.1. Operator's goal

Through this phase, the operator wants to control its autonomic network in two things:

1 determine the scope of autonomic functions by instructing which of the network resources have to be managed by which autonomic function (and more precisely which class e.g. 1. version X or version Y or 2. provider A or provider B),

2 determine how the autonomic functions are organized by instructing which ASAs have to interact with which other ASAs (or more precisely which set of network resources have to be handled as an autonomous group by their managing ASAs).

Additionally in this phase, the operator may want to set objectives to autonomic functions, by configuring the ASAs technical objectives.

The operator's goal can be summarized in an instruction to the ANIMA ecosystem matching the following pattern:

```
[ASA of type_x instances] ready to control
[Instantiation_target_Infrastructure] with
[Instantiation_target_parameters]
```

6.2.2. Instantiation phase inputs and outputs

Inputs are:

[ASA of type_x instances] that specifies which are the ASAs to be targeted (and more precisely which class e.g. 1. version X or version Y or 2. provider A or provider B),

[Instantiation_target_Infrastructure] that specifies which are the resources to be managed by the autonomic function, this can be the whole network or a subset of it like a domain a technology segment or even a specific list of resources,

[Instantiation_target_parameters] that specifies which are the technical objectives to be set to ASAs (e.g. an optimization target)

Outputs are:

[Set of ASAs - Resources relations] describing which resources are managed by which ASA instances, this is not a formal message, but a resulting configuration of a set of ASAs,

6.2.3. Instantiation phase requirements

The instructions described in section 4.2 could be either:

sent to a targeted ASA In which case, the receiving Agent will have to manage the specified list of [Instantiation_target_Infrastructure], with the [Instantiation_target_parameters].

broadcast to all ASAs In which case, the ASAs would collectively determine from the list which Agent(s) would handle which [Instantiation_target_Infrastructure], with the [Instantiation_target_parameters].

This set of instructions can be materialized through a message that is named an Instance Mandate (description TBD).

The conclusion of this instantiation phase is a ready to operate ASA (or interacting set of ASAs), then this (or those) ASA(s) can describe themselves by depicting which are the resources they manage and what this means in terms of metrics being monitored and in terms of actions that can be executed (like modifying the parameters values). A message conveying such a self description is named an Instance Manifest (description TBD).

Though the operator may well use such a self-description "per se", the final goal of such a description is to be shared with other ANIMA entities like:

o the coordination entities (see [I-D.ciavaglia-anima-coordination] - Autonomic Functions Coordination)

o collaborative entities in the purpose of establishing knowledge exchanges (some ASAs may produce knowledge or even monitor metrics that other ASAs cannot make by themselves why those would be useful for their execution)

6.3. Operation phase

Note: This section is to be further developed in future revisions of the document, especially the implications on the design of ASAs.

During the Operation phase, the operator can:

Activate/Deactivate ASA: meaning enabling those to execute their autonomic loop or not.

Modify ASAs targets: meaning setting them different objectives.

Modify ASAs managed resources: by updating the instance mandate which would specify different set of resources to manage (only applicable to decouples ASAs).

During the Operation phase, running ASAs can interact the one with the other:

in order to exchange knowledge (e.g. an ASA providing traffic predictions to load balancing ASA)

in order to collaboratively reach an objective (e.g. ASAs pertaining to the same autonomic function targeted to manage a network domain, these ASA will collaborate - in the case of a load balancing one, by modifying the links metrics according to the neighboring resources loads)

During the Operation phase, running ASAs are expected to apply coordination schemes

then execute their control loop under coordination supervision/ instructions

The ASA life-cycle is discussed in more detail in "A Day in the Life of an Autonomic Function" [I-D.peloso-anima-autonomic-function].

7. Coordination between Autonomic Functions

Some autonomic functions will be completely independent of each other. However, others are at risk of interfering with each other - for example, two different optimization functions might both attempt to modify the same underlying parameter in different ways. In a complete system, a method is needed of identifying ASAs that might interfere with each other and coordinating their actions when necessary. This issue is considered in "Autonomic Functions Coordination" [I-D.ciavaglia-anima-coordination].

8. Coordination with Traditional Management Functions

Some ASAs will have functions that overlap with existing configuration tools and network management mechanisms such as command line interfaces, DHCP, DHCPv6, SNMP, NETCONF, RESTCONF and YANG-based solutions. Each ASA designer will need to consider this issue and how to avoid clashes and inconsistencies. Some specific

considerations for interaction with OAM tools are given in [RFC8368]. As another example, [I-D.ietf-anima-prefix-management] describes how autonomic management of IPv6 prefixes can interact with prefix delegation via DHCPv6. The description of a GRASP objective and of an ASA using it should include a discussion of any such interactions.

A related aspect is that management functions often include a data model, quite likely to be expressed in a formal notation such as YANG. This aspect should not be an afterthought in the design of an ASA. To the contrary, the design of the ASA and of its GRASP objectives should match the data model; as noted above, YANG serialized as CBOR may be used directly as the value of a GRASP objective.

9. Robustness

It is of great importance that all components of an autonomic system are highly robust. In principle they must never fail. This section lists various aspects of robustness that ASA designers should consider.

1. If despite all precautions, an ASA does encounter a fatal error, it should in any case restart automatically and try again. To mitigate a hard loop in case of persistent failure, a suitable pause should be inserted before such a restart. The length of the pause depends on the use case.

2. If a newly received or calculated value for a parameter falls out of bounds, the corresponding parameter should be either left unchanged or restored to a safe value.

3. If a GRASP synchronization or negotiation session fails for any reason, it may be repeated after a suitable pause. The length of the pause depends on the use case.

4. If a session fails repeatedly, the ASA should consider that its peer has failed, and cause GRASP to flush its discovery cache and repeat peer discovery.

5. Any received GRASP message should be checked. If it is wrongly formatted, it should be ignored. Within a unicast session, an Invalid message (M_INVALID) may be sent. This function may be provided by the GRASP implementation itself.

6. Any received GRASP objective should be checked. If it is wrongly formatted, it should be ignored. Within a negotiation session, a Negotiation End message (M_END) with a Decline option (O_DECLINE)

should be sent. An ASA may log such events for diagnostic purposes.

7. If an ASA receives either an Invalid message (M_INVALID) or a Negotiation End message (M_END) with a Decline option (O_DECLINE), one possible reason is that the peer ASA does not support a new feature of either GRASP or of the objective in question. In such a case the ASA may choose to repeat the operation concerned without using that new feature.

8. All other possible exceptions should be handled in an orderly way. There should be no such thing as an unhandled exception (but see point 1 above).

10. Security Considerations

ASAs are intended to run in an environment that is protected by the Autonomic Control Plane [I-D.ietf-anima-autonomic-control-plane], admission to which depends on an initial secure bootstrap process [I-D.ietf-anima-bootstrapping-keyinfra]. In some deployments, a secure partition of the link layer might be used instead [I-D.carpenter-anima-l2acp-scenarios]. However, this does not relieve ASAs of responsibility for security. In particular, when ASAs configure or manage network elements outside the ACP, they must use secure techniques and carefully validate any incoming information. As appropriate to their specific functions, ASAs should take account of relevant privacy considerations [RFC6973].

Authorization of ASAs is a subject for future study. At present, ASAs are trusted by virtue of being installed on a node that has successfully joined the ACP.

11. IANA Considerations

This document makes no request of the IANA.

12. Acknowledgements

Useful comments were received from Toerless Eckert, Alex Galis, Bing Liu, and other members of the ANIMA WG.

13. References

13.1. Normative References

[I-D.ietf-anima-autonomic-control-plane]
Eckert, T., Behringer, M., and S. Bjarnason, "An Autonomic Control Plane (ACP)", draft-ietf-anima-autonomic-control-plane-19 (work in progress), March 2019.

[I-D.ietf-anima-bootstrapping-keyinfra]
Pritikin, M., Richardson, M., Behringer, M., Bjarnason, S., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructures (BRSKI)", draft-ietf-anima-bootstrapping-keyinfra-22 (work in progress), June 2019.

[I-D.ietf-anima-grasp]
Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-15 (work in progress), July 2017.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

13.2. Informative References

[DeMola06]
De Mola, F. and R. Quitadamo, "An Agent Model for Future Autonomic Communications", Proceedings of the 7th WOA 2006 Workshop From Objects to Agents 51-59, September 2006.

[GANA13] "Autonomic network engineering for the self-managing Future Internet (AFI): GANA Architectural Reference Model for Autonomic Networking, Cognitive Networking and Self-Management.", April 2013, <http://www.etsi.org/deliver/etsi_gs/AFI/001_099/002/01.01.01_60/gs_afi002v010101p.pdf>.

[Huebscher08]
Huebscher, M. and J. McCann, "A survey of autonomic computing--degrees, models, and applications", ACM Computing Surveys (CSUR) Volume 40 Issue 3 DOI: 10.1145/1380584.1380585, August 2008.

[I-D.carpenter-anima-l2acp-scenarios]
Carpenter, B. and B. Liu, "Scenarios and Requirements for Layer 2 Autonomic Control Planes", draft-carpenter-anima-l2acp-scenarios-00 (work in progress), February 2019.

[I-D.ciavaglia-anima-coordination]
Ciavaglia, L. and P. Peloso, "Autonomic Functions Coordination", draft-ciavaglia-anima-coordination-01 (work in progress), March 2016.

[I-D.ietf-anima-grasp-api]
Carpenter, B., Liu, B., Wang, W., and X. Gong, "Generic Autonomic Signaling Protocol Application Program Interface (GRASP API)", draft-ietf-anima-grasp-api-03 (work in progress), January 2019.

[I-D.ietf-anima-prefix-management]
Jiang, S., Du, Z., Carpenter, B., and Q. Sun, "Autonomic IPv6 Edge Prefix Management in Large-scale Networks", draft-ietf-anima-prefix-management-07 (work in progress), December 2017.

[I-D.ietf-anima-reference-model]
Behringer, M., Carpenter, B., Eckert, T., Ciavaglia, L., and J. Nobre, "A Reference Model for Autonomic Networking", draft-ietf-anima-reference-model-10 (work in progress), November 2018.

[I-D.ietf-core-yang-cbor]
Veillette, M., Petrov, I., and A. Pelov, "CBOR Encoding of Data Modeled with YANG", draft-ietf-core-yang-cbor-10 (work in progress), April 2019.

[I-D.peloso-anima-autonomic-function]
Pierre, P. and L. Ciavaglia, "A Day in the Life of an Autonomic Function", draft-peloso-anima-autonomic-function-01 (work in progress), March 2016.

[Movahedi12]
Movahedi, Z., Ayari, M., Langar, R., and G. Pujolle, "A Survey of Autonomic Network Architectures and Evaluation Criteria", IEEE Communications Surveys & Tutorials Volume: 14 , Issue: 2 DOI: 10.1109/SURV.2011.042711.00078, Page(s): 464 - 490, 2012.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

[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>.

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

[RFC8368] Eckert, T., Ed. and M. Behringer, "Using an Autonomic Control Plane for Stable Connectivity of Network Operations, Administration, and Maintenance (OAM)", RFC 8368, DOI 10.17487/RFC8368, May 2018, <https://www.rfc-editor.org/info/rfc8368>.

[RFC8568] Bernardos, CJ., Rahman, A., Zuniga, JC., Contreras, LM., Aranda, P., and P. Lynch, "Network Virtualization Research Challenges", RFC 8568, DOI 10.17487/RFC8568, April 2019, <https://www.rfc-editor.org/info/rfc8568>.

Appendix A. Change log [RFC Editor: Please remove]

draft-carpenter-anima-asa-guidelines-07, 2019-07-17:

Improved explanation of threading vs event-loop

Other editorial improvements.

draft-carpenter-anima-asa-guidelines-06, 2018-01-07:

Expanded and improved example logic flow.

Editorial corrections.

draft-carpenter-anima-asa-guidelines-05, 2018-06-30:

Added section on relationshp with non-autonomic components.

Editorial corrections.

draft-carpenter-anima-asa-guidelines-04, 2018-03-03:

Added note about simple ASAs.

Added note about NFV/SFC services.

Improved text about threading v event loop model

Added section about coordination with traditional tools.

Added appendix with example logic flow.

draft-carpenter-anima-asa-guidelines-03, 2017-10-25:

Added details on life cycle.

Added details on robustness.

Added co-authors.

draft-carpenter-anima-asa-guidelines-02, 2017-07-01:

Expanded description of event-loop case.

Added note about 'dry run' mode.

draft-carpenter-anima-asa-guidelines-01, 2017-01-06:

More sections filled in

draft-carpenter-anima-asa-guidelines-00, 2016-09-30:

Initial version

Appendix B. Example Logic Flows

This appendix describes generic logic flows for an Autonomic Service Agent (ASA) for resource management. Note that these are illustrative examples, and in no sense requirements. As long as the rules of GRASP are followed, a real implementation could be different. The reader is assumed to be familiar with GRASP [I-D.ietf-anima-grasp] and its conceptual API [I-D.ietf-anima-grasp-api].

A complete autonomic function for a resource would consist of a number of instances of the ASA placed at relevant points in a network. Specific details will of course depend on the resource concerned. One example is IP address prefix management, as specified in [I-D.ietf-anima-prefix-management]. In this case, an instance of the ASA would exist in each delegating router.

An underlying assumption is that there is an initial source of the resource in question, referred to here as a master ASA. The other ASAs, known as delegators, obtain supplies of the resource from the

master, and then delegate quantities of the resource to consumers that request it, and recover it when no longer needed.

Another assumption is there is a set of network wide policy parameters, which the master will provide to the delegators. These parameters will control how the delegators decide how much resource to provide to consumers. Thus the ASA logic has two operating modes: master and delegator. When running as a master, it starts by obtaining a quantity of the resource from the NOC, and it acts as a source of policy parameters, via both GRASP flooding and GRASP synchronization. (In some scenarios, flooding or synchronization alone might be sufficient, but this example includes both.)

When running as a delegator, it starts with an empty resource pool, it acquires the policy parameters by GRASP synchronization, and it delegates quantities of the resource to consumers that request it. Both as a master and as a delegator, when its pool is low it seeks quantities of the resource by requesting GRASP negotiation with peer ASAs. When its pool is sufficient, it hands out resource to peer ASAs in response to negotiation requests. Thus, over time, the initial resource pool held by the master will be shared among all the delegators according to demand.

In theory a network could include any number of masters and any number of delegators, with the only condition being that each master's initial resource pool is unique. A realistic scenario is to have exactly one master and as many delegators as you like. A scenario with no master is useless.

An implementation requirement is that resource pools are kept in stable storage. Otherwise, if a delegator exits for any reason, all the resources it has obtained or delegated are lost. If a master exits, its entire spare pool is lost. The logic for using stable storage and for crash receovery is not included below.

The description below does not implement GRASP's 'dry run' function. That would require temporarily marking any resource handed out in a dry run negotiation as reserved, until either the peer obtains it in a live run, or a suitable timeout expires.

The main data structures used in each instance of the ASA are:

- The resource_pool, for example an ordered list of available resources. Depending on the nature of the resource, units of resource are split when appropriate, and a background garbage collector recombines split resources if they are returned to the pool.

- The delegated_list, where a delegator stores the resources it has given to consumers routers.

Possible main logic flows are below, using a threaded implementation model. The transformation to an event loop model should be apparent - each thread would correspond to one event in the event loop.

The GRASP objectives are as follows:

> ["EX1.Resource", flags, loop_count, value] where the value depends on the resource concerned, but will typically include its size and identification.

> ["EX1.Params", flags, loop_count, value] where the value will be, for example, a JSON object defining the applicable parameters.

In the outline logic flows below, these objectives are represented simply by their names.

```
MAIN PROGRAM:

Create empty resource_pool (and an associated lock)
Create empty delegated_list
Determine whether to act as master
if master:
    Obtain initial resource_pool contents from NOC
    Obtain value of EX1.Params from NOC
Register ASA with GRASP
Register GRASP objectives EX1.Resource and EX1.Params
if master:
    Start FLOODER thread to flood EX1.Params
    Start SYNCHRONIZER listener for EX1.Params
Start MAIN_NEGOTIATOR thread for EX1.Resource
if not master:
    Obtain value of EX1.Params from GRASP flood or synchronization
    Start DELEGATOR thread
Start GARBAGE_COLLECTOR thread
do forever:
    good_peer = none
    if resource_pool is low:
        Calculate amount A of resource needed
        Discover peers using GRASP M_DISCOVER / M_RESPONSE
        if good_peer in peers:
            peer = good_peer
        else:
            peer =  #any choice among peers
            grasp.request_negotiate("EX1.Resource", peer)
            i.e., send M_REQ_NEG
            Wait for response (M_NEGOTIATE, M_END or M_WAIT)
            if OK:
                if offered amount of resource sufficient:
                    Send M_END + O_ACCEPT #negotiation succeeded
                    Add resource to pool
                    good_peer = peer
                else:
                    Send M_END + O_DECLINE #negotiation failed
    sleep() #sleep time depends on application scenario

MAIN_NEGOTIATOR thread:

do forever:
    grasp.listen_negotiate("EX1.Resource")
    i.e., wait for M_REQ_NEG
    Start a separate new NEGOTIATOR thread for requested amount A
```

```
NEGOTIATOR thread:

Request resource amount A from resource_pool
if not OK:
    while not OK and A > Amin:
        A = A-1
        Request resource amount A from resource_pool
if OK:
    Offer resource amount A to peer by GRASP M_NEGOTIATE
    if received M_END + O_ACCEPT:
        #negotiation succeeded
    elif received M_END + O_DECLINE or other error:
        #negotiation failed
else:
    Send M_END + O_DECLINE #negotiation failed

DELEGATOR thread:

do forever:
    Wait for request or release for resource amount A
    if request:
        Get resource amount A from resource_pool
        if OK:
            Delegate resource to consumer
            Record in delegated_list
        else:
            Signal failure to consumer
            Signal main thread that resource_pool is low
    else:
        Delete resource from delegated_list
        Return resource amount A to resource_pool

SYNCHRONIZER thread:

do forever:
  Wait for  M_REQ_SYN message for EX1.Params
  Reply with M_SYNCH message for EX1.Params

FLOODER thread:

do forever:
  Send M_FLOOD message for EX1.Params
  sleep() #sleep time depends on application scenario
```

```
GARBAGE_COLLECTOR thread:

do forever:
    Search resource_pool for adjacent resources
    Merge adjacent resources
    sleep() #sleep time depends on application scenario
```

Authors' Addresses

Brian Carpenter
School of Computer Science
University of Auckland
PB 92019
Auckland 1142
New Zealand

Email: brian.e.carpenter@gmail.com

Laurent Ciavaglia
Nokia
Villarceaux
Nozay 91460
FR

Email: laurent.ciavaglia@nokia.com

Sheng Jiang
Huawei Technologies Co., Ltd
Q14, Huawei Campus, No.156 Beiqing Road
Hai-Dian District, Beijing, 100095
P.R. China

Email: jiangsheng@huawei.com

Pierre Peloso
Nokia
Villarceaux
Nozay 91460
FR

Email: pierre.peloso@nokia.com