ï»¿

Internet Engineering Task Force (IETF)
Request for Comments: 9222
Category: Informational
ISSN: 2070-1721

B. Carpenter
Univ. of Auckland
L. Ciavaglia
Rakuten Mobile
S. Jiang
Huawei Technologies Co., Ltd
P. Peloso
Nokia

March 2022

Guidelines for Autonomic Service Agents

Abstract

This document proposes guidelines for the design of Autonomic Service Agents for autonomic networks. Autonomic Service Agents, together with the Autonomic Network Infrastructure, the Autonomic Control Plane, and the GeneRic Autonomic Signaling Protocol, constitute base elements of an autonomic networking ecosystem.

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 autonomic networking reference model [RFC8993]. This infrastructure makes use of the Autonomic Control Plane (ACP) [RFC8994] and the GeneRic Autonomic Signaling Protocol (GRASP) [RFC8990]. A general introduction to this environment may be found at [IPJ], which also includes explanatory diagrams, and a summary of terminology is in Section 2.

This document is a contribution to the description of an autonomic networking ecosystem, recognizing that a deployable autonomic network needs more than just ACP and GRASP implementations. Such an autonomic network must achieve management tasks that a Network Operations Center (NOC) cannot readily achieve manually, such as continuous resource optimization or automated fault detection and repair. These tasks, and other management automation goals, are described at length in [RFC7575]. The net result should be significant operational improvement. To achieve this, the autonomic networking ecosystem must include at least a library of ASAs and corresponding GRASP technical objective definitions. A GRASP objective [RFC8990] is a data structure whose main contents are a name and a value. The value consists of a single configurable parameter or a set of parameters of some kind.

There must also be tools to deploy and oversee ASAs, and integration with existing operational mechanisms [RFC8368]. However, this document focuses on the design of ASAs, with some reference to implementation and operational aspects.

There is 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 components. They are components of network or service management; they do not in themselves provide services to end users. They do, however, provide management services to network operators and administrators. For example, the services envisaged for network function virtualization (NFV) [NFV] or for service function chaining (SFC) [RFC7665] might be managed by an ASA rather than by traditional configuration tools.

Another example is that an existing script running within a router to locally monitor or configure functions or services could be upgraded to an ASA that could communicate with peer scripts on neighboring or remote routers. A high-level API will allow such upgraded scripts to take full advantage of the secure ACP and the discovery, negotiation, and synchronization features of GRASP. Familiar tasks such as configuring an Interior Gateway Protocol (IGP) on neighboring routers or even exchanging IGP security keys could be performed securely in this way. This document mainly addresses issues affecting quite complex ASAs, but initially, the most useful ASAs may in fact be rather simple evolutions of existing scripts.

The reference model [RFC8993] for autonomic networks explains further the functionality of ASAs by adding the following:

> [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 GRASP [RFC8990] 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. It may therefore be a quite complex piece of software in its own right, forming part of the application layer above the ANI.

As mentioned, there will certainly be simple ASAs that manage a single objective in a straightforward way and do not need asynchronous operations. In nodes where computing power and memory space are limited, ASAs should run at a much lower frequency than the primary workload, so CPU load should not be a big issue, but memory footprint in a constrained node is certainly a concern. ASAs installed in constrained devices will have limited functionality. In such cases, many aspects of the current document do not apply. However, in the general case, an ASA may be a relatively complex software component that will in many cases control and monitor simpler entities in the same or remote host(s). 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 complex ASAs. Some of the material may be familiar to those experienced in distributed fault-tolerant and real-time control systems. Robustness and security are of particular importance in autonomic networks and are discussed in Sections 9 and 10.

2. Terminology

This section summarizes various acronyms and terminology used in the document. Where no other reference is given, please consult [RFC8993] or [RFC7575].

Autonomic: self-managing (self-configuring, self-protecting, self-healing, self-optimizing), but allowing high-level guidance by a central entity such as a NOC

Autonomic Function: a function that adapts on its own to a changing environment

Autonomic Node: a node that employs autonomic functions

ACP: Autonomic Control Plane [RFC8994]

AN: Autonomic Network; a network of autonomic nodes, which interact directly with each other

ANI: Autonomic Network Infrastructure

ASA: Autonomic Service Agent; an agent installed on an autonomic node that implements an autonomic function, either partially (in the case of a distributed function) or completely

BRSKI: Bootstrapping Remote Secure Key Infrastructure [RFC8995]

CBOR: Concise Binary Object Representation[RFC8949]

GRASP: GeneRric Autonomic Signaling Protocol [RFC8990]

GRASP API: GRASP Application Programming Interface [RFC8991]

NOC: Network Operations Center [RFC8368]

Objective: A GRASP technical objective is a data structure whose main contents are a name and a value. The value consists of a single configurable parameter or a set of parameters of some kind [RFC8990].

3. Logical Structure of an Autonomic Service Agent

As mentioned above, all but the simplest ASAs will need to support asynchronous operations. Different programming environments support asynchronicity in different ways. In this document, we use an explicit multi-threading model to describe operations. This is illustrative, and alternatives to multi-threading are discussed in detail in connection with the GRASP API (see Section 4.3).

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

* obtain authorization credentials, if needed
* register the ASA with GRASP
* acquire relevant policy parameters
* declare 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 may be launched. Examples of such threads 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 should all either exit after their job is done or enter a wait state for new work to avoid wasting system resources.

According to the degree of parallelism needed by the application, some of these threads 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 synchronization primitives 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 and, if appropriate, occasionally refresh authorization credentials. It should also do whatever is required to avoid unnecessary resource consumption, for example, by limiting its frequency of execution.

The self-monitoring thread is of considerable importance. Failure of autonomic service agents is highly undesirable. 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 A presents some example logic flows in informal pseudocode.

4. Interaction with the Autonomic Networking Infrastructure

4.1. 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, typically in accordance with [RFC8995]. 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 incorporates simple special-purpose ASAs that use a restricted mode of GRASP (Section 4 of [RFC8995]).

4.2. Interaction with the Autonomic Control Plane

In a normal autonomic network, ASAs will run as clients of the ACP, which 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 incorporates simple special-purpose ASAs that use a restricted mode of GRASP (Section 6.4 of [RFC8994]).

4.3. Interaction with GRASP and its API

In a node where a significant number of ASAs are installed, GRASP [RFC8990] is likely to run as a separate process with its API [RFC8991] 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 6), 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 serialize it for transmission in Concise Binary Object Representation (CBOR) [RFC8949], subject only to GRASP's maximum message size as discussed in Section 6.

As discussed in Section 3, GRASP is an asynchronous protocol, and this document uses a multi-threading model to describe operations. In many programming environments, an "event loop" model is used instead, in which case each thread would be implemented as an event handler called in turn by the main loop. For this case, the GRASP API must provide non-blocking calls and possibly support callbacks. This topic is discussed in more detail in [RFC8991], and other asynchronicity models are also possible. Whenever necessary, the GRASP session identifier will be used to distinguish simultaneous operations.

The GRASP API should offer the following features:

* Registration functions, so that an ASA can register itself and the objectives that it manages.

* A discovery function, by which an ASA can discover other ASAs supporting a given objective.

* 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. Once a negotiation starts, it is a symmetric process with both sides sending successive objective values to each other until agreement is reached (or the negotiation fails).

* 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. Unlike negotiation, synchronization is an asymmetric process in which the listener sends a single objective value to the requester.

* 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 [RFC8991].

The GRASP API is intended to support the various interactions expected between most ASAs, such as the interactions outlined in Section 3. However, if ASAs require additional communication between themselves, they can do so directly over the ACP to benefit from its security. One option is to use GRASP discovery and synchronization as a rendezvous mechanism between two ASAs, passing communication parameters such as a TCP port number via GRASP. The use of TLS over the ACP for such communications is advisable, as described in Section 6.9.2 of [RFC8994].

4.4. Interaction with Policy Mechanisms

At the time of writing, the policy mechanisms for the ANI are undefined. In particular, the use of declarative policies (aka Intents) for the definition and management of an ASA's behaviors remains a research topic [IBN-CONCEPTS].

In the cases where ASAs are defined as closed control loops, the specifications defined in [ZSM009-1] regarding imperative and declarative goal statements may be applicable.

In the ANI, policy dissemination is expected to operate by an information distribution mechanism (e.g., via GRASP [RFC8990]) 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.

5. Interaction with Non-autonomic Components and Systems

To have any external effects, an ASA 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 managing an entity that is also managed by local software must interact with that software. For example, if such management is performed by NETCONF [RFC6241], the ASA must interact with the NETCONF server as an independent NETCONF client in the same node to avoid any inconsistency between configuration changes delivered via NETCONF and configuration changes made by the ASA.

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.

6. Design of GRASP Objectives

The design of an ASA will often require the design of a new GRASP objective. The general rules for the format of GRASP objectives, their names, and IANA registration are given in [RFC8990]. Additionally, that document discusses various general considerations for the design of objectives, which are not repeated here. However, note that GRASP, like HTTP, does not provide transactional integrity. In particular, steps in a GRASP negotiation are not idempotent. The design of a GRASP objective and the logic flow of the ASA should take this into account. One approach, which should be used when possible, is to design objectives with idempotent semantics. If this is not possible, typically 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. The easiest way is to run only one negotiation at a time. If an ASA is capable of overlapping several negotiations, it must avoid interference between these negotiations.

Negotiations will always end, normally because one end or the other declares success or failure. If this does not happen, either a timeout or exhaustion of the loop count will occur. The definition of a GRASP objective should describe a specific negotiation policy if it is not self-evident.

GRASP allows a "dry run" mode of negotiation, where a negotiation session follows its normal course but is not committed at either end until a subsequent live negotiation session. 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) [RFC8949]. For some objectives, a single data item will suffice, for example, an integer, a floating point number, a UTF-8 string, or an arbitrary byte string. For more complex cases, a simple tuple structure such as [item1, item2, item3] could be used. Since CBOR is closely linked to JSON, it is also rather easy to define an objective whose value is a JSON structure. The formats acceptable by the GRASP API will limit the options in practice. A generic 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 (Section 2.3.2.4 of [RFC8991]).

The maximum size of the value field of an objective is limited by the GRASP maximum message size. If the default maximum size specified as GRASP_DEF_MAX_SIZE by [RFC8990] is not enough, the specification of the objective must indicate the required maximum message size for both unicast and multicast messages.

A mapping from YANG to CBOR is defined by [CBOR-YANG]. Subject to the size limit defined for GRASP messages, nothing prevents objectives transporting YANG in this way.

The flexibility of CBOR implies that the value field of many objectives can be extended in service, to add additional information or alternative content, especially if JSON-like structures are used. This has consequences for the robustness of ASAs, as discussed in Section 9.

7. Life Cycle

The ASA life cycle is discussed in [AUTONOMIC-FUNCTION], from which the following text was derived. It does not cover all details, and

   some of the terms used would require precise definitions in a given
   implementation.

   In simple cases, 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, in complex cases, a more likely situation
   is that ASAs need to be installed or updated dynamically because of
   new requirements or bugs. This section describes one approach to the
   resulting life cycle of individual ASAs. It does not consider wider
   issues such as updates of shared libraries.

   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

7.1. Installation Phase

   We define "installation" to mean that a piece of software is loaded
   into a device, along with any necessary libraries, but is not yet
   activated.

   Before being able to instantiate and run ASAs, the operator will
   first provision the infrastructure with the sets of ASA software
   corresponding to its needs and objectives. Such software must be
   checked for integrity and authenticity before installation. The
   provisioning of the infrastructure is realized in the installation
   phase and consists of installing (or checking the availability of)
   the pieces of software of the different ASAs in a set of Installation
   Hosts within the autonomic network.

   There are three 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 an ASA on one machine to control
      resources in another machine (known as "decoupled mode").

   * 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 could be
   installed on the infrastructure.

7.1.1. Installation Phase Inputs and Outputs

   Inputs are:

   * [ASA_type]: specifies which ASA to install.

* [Installation_target_infrastructure]: specifies the candidate installation Hosts.

* [ASA_placement_function]: specifies how the installation phase will meet the operator's needs and objectives for the provision of the infrastructure. This function is only useful in the decoupled mode. It can be as simple as an explicit list of hosts on which the ASAs are to be installed, or it could consist of operator-defined criteria and constraints.

The main output of the installation phase is a [List_of_ASAs] installed on [List_of_hosts]. This output is also useful for the coordination function where it acts as a static interaction map (see Section 8.1).

The condition to validate in order to pass to next phase is to ensure that [List_of_ASAs] are correctly installed on [List_of_hosts]. A minimum set of primitives to support the installation of ASAs could be the following: install (List_of_ASAs, Installation_target_infrastructure, ASA_placement_function) and uninstall (List_of_ASAs).

7.2. Instantiation Phase

We define "instantiation" as the operation of creating a single ASA instance from the corresponding piece of installed software.

Once the ASAs are installed on the appropriate hosts in the network, these ASAs 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.

There are two aspects to take into consideration. First, having a piece of code installed and available to run on a host is not the same as 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 ASA runs on the same autonomic node as the resources it is controlling). In a decoupled mode, determining this is a bit more complex: a starting agent will have to either discover the set of resources it ought to control, or such information has to be communicated to the ASA.

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

7.2.1. Operator's Goal

Through this phase, the operator wants to control its autonomic network regarding at least two aspects:

1. determine the scope of autonomic functions by instructing which network resources have to be managed by which autonomic function (and more precisely by which release of the ASA software code, e.g., version number or provider).

2. determine how the autonomic functions are organized by instantiating a set of ASAs across one or more autonomic nodes and instructing them accordingly about the other ASAs in the set as necessary.

In this phase, the operator may also want to set goals for autonomic functions, e.g., by configuring GRASP objectives.

The operator's goal can be summarized in an instruction to the autonomic ecosystem matching the following format, explained in detail in the next sub-section:

[Instances_of_ASA_type] ready to control
[Instantiation_target_infrastructure] with
[Instantiation_target_parameters]

7.2.2. Instantiation Phase Inputs and Outputs

Inputs are:

* [Instances_of_ASA_type]: specifies which ASAs to instantiate

* [Instantiation_target_infrastructure]: specifies which resources are to be managed by the autonomic function; this can be the whole network or a subset of it like a domain, a physical segment, or even a specific list of resources.

* [Instantiation_target_parameters]: specifies which GRASP objectives are to be sent to ASAs (e.g., an optimization target)

Outputs are:

* [Set_of_ASA_resources_relations]: describes which resources are managed by which ASA instances; this is not a formal message but a resulting configuration log for a set of ASAs.

7.2.3. Instantiation Phase Requirements

The instructions described in Section 7.2 could be either of the following:

* Sent to a targeted ASA. In this 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 this case, the ASAs would determine from the list which ASAs would handle which [Instantiation_target_infrastructure], with the [Instantiation_target_parameters].

These instructions may be grouped as a specific data structure referred to as an ASA Instance Mandate. The specification of such an ASA Instance Mandate is beyond the scope of this document.

The conclusion of this instantiation phase is a set of ASA instances ready to operate. These ASA instances are characterized by the resources they manage, the metrics being monitored, and the actions that can be executed (like modifying certain parameter values). The description of the ASA instance may be defined in an ASA Instance Manifest data structure. The specification of such an ASA Instance Manifest is beyond the scope of this document.

The ASA Instance Manifest does not only serve informational purposes such as acknowledgement of successful instantiation to the operator but is also necessary for further autonomic operations with:

* coordinated entities (see Section 8.1)

* collaborative entities with purposes such as to establish knowledge exchange (some ASAs may produce knowledge or monitor metrics that would be useful for other ASAs)

7.3. Operation Phase

During the operation phase, the operator can:

* activate/deactivate ASAs: enable/disable their autonomic loops

* modify ASA targets: set different technical objectives

* modify ASAs managed resources: update the Instance Mandate to specify a different set of resources to manage (only applicable to

decoupled ASAs)

During the operation phase, running ASAs can interact with other ASAs:

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

* in order to collaboratively reach an objective (e.g., ASAs pertaining to the same autonomic function will collaborate, e.g., in the case of a load balancing function, by modifying link metrics according to neighboring resource loads)

During the operation phase, running ASAs are expected to apply coordination schemes as per Section 8.1.

7.4. Removal Phase

When an ASA is removed from service and uninstalled, the above steps are reversed. It is important that its data, especially any security key material, is purged.

8. Coordination and Data Models

8.1. 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 for identifying ASAs that might interfere with each other and coordinating their actions when necessary.

8.2. 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, and RESTCONF. This is, of course, an existing problem whenever multiple configuration tools are in use by the NOC. Each ASA designer will need to consider this issue and how to avoid clashes and inconsistencies in various deployment scenarios. Some specific considerations for interaction with OAM tools are given in [RFC8368]. As another example, [RFC8992] 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.

8.3. Data Models

Management functions often include a shared 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 in Section 6, 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. Although ASA designers should aim for their component to never fail, it is more important to design the ASA to assume that failures will happen and to gracefully recover from those failures when they occur. Hence, 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 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; randomization and exponential backoff should be considered.

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 value known to be safe in all configurations.

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; randomization and exponential backoff should be considered.

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

5. In any case, it may be prudent to repeat discovery periodically, depending on the use case.

6. 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.

7. Any received GRASP objective should be checked. Basic formatting errors like invalid CBOR will likely be detected by GRASP itself, but the ASA is responsible for checking the precise syntax and semantics of a received objective. 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.

8. On the other hand, the definitions of GRASP objectives are very likely to be extended, using the flexibility of CBOR or JSON. Therefore, ASAs should be able to deal gracefully with unknown components within the values of objectives. The specification of an objective should describe how unknown components are to be handled (ignored, logged and ignored, or rejected as an error).

9. 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 the objective in question. In such a case, the ASA may choose to repeat the operation concerned without using that new feature.

10. 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).

At a slightly more general level, ASAs are not services in themselves, but they automate services. This has a fundamental impact on how to design robust ASAs. In general, when an ASA observes a particular state (1) of operations of the services/ resources it controls, it typically aims to improve this state to a better state, say (2). Ideally, the ASA is built so that it can ensure that any error encountered can still lead to returning to (1) instead of a state (3), which is worse than (1). One example instance of this principle is "make-before-break" used in reconfiguration of routing protocols in manual operations. This principle of operations can accordingly be coded into the operation of an ASA. The GRASP dry run option mentioned in Section 6 is another tool helpful for this ASA design goal of "test-before-make".

10. Security Considerations

ASAs are intended to run in an environment that is protected by the Autonomic Control Plane [RFC8994], admission to which depends on an initial secure bootstrap process such as BRSKI [RFC8995]. Those documents describe security considerations relating to the use of and properties provided by the ACP and BRSKI, respectively. Such an ACP can provide keying material for mutual authentication between ASAs as well as confidential communication channels for messages between ASAs. In some deployments, a secure partition of the link layer might be used instead. GRASP itself has significant security considerations [RFC8990]. However, this does not relieve ASAs of responsibility for security. When ASAs configure or manage network elements outside the ACP, potentially in a different physical node, they must interact with other non-autonomic software components to perform their management functions. The details are specific to each case, but this has an important security implication. An ASA might act as a loophole by which the managed entity could penetrate the security boundary of the ANI. Thus, ASAs must be designed to avoid loopholes such as passing on executable code or proxying unverified commands and should, if possible, operate in an unprivileged mode. In particular, they must use secure coding practices, e.g., carefully validate all incoming information and avoid unnecessary elevation of privilege. This will apply in particular when an ASA interacts with a management component such as a NETCONF server.

A similar situation will arise if an ASA acts as a gateway between two separate autonomic networks, i.e., it has access to two separate ACPs. Such an ASA must also be designed to avoid loopholes and to validate incoming information from both sides.

As a reminder, GRASP does not intrinsically provide transactional integrity (Section 6).

As appropriate to their specific functions, ASAs should take account of relevant privacy considerations [RFC6973].

The initial version of the autonomic infrastructure assumes that all autonomic nodes are trusted by virtue of their admission to the ACP. ASAs are therefore trusted to manipulate any GRASP objective simply because they are installed on a node that has successfully joined the ACP. In the general case, a node may have multiple roles, and a role may use multiple ASAs, each using multiple GRASP objectives. Additional mechanisms for the fine-grained authorization of nodes and ASAs to manipulate specific GRASP objectives could be designed. Meanwhile, we repeat that ASAs should run without special privilege if possible. Independently of this, interfaces between ASAs and the router configuration and monitoring services of the node can be subject to authentication that provides more fine-grained authorization for specific services. These additional authentication parameters could be passed to an ASA during its instantiation phase.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC8990] Bormann, C., Carpenter, B., Ed., and B. Liu, Ed., "GeneRic Autonomic Signaling Protocol (GRASP)", RFC 8990, DOI 10.17487/RFC8990, May 2021, <https://www.rfc-editor.org/info/rfc8990>.

[RFC8994] Eckert, T., Ed., Behringer, M., Ed., and S. Bjarnason, "An Autonomic Control Plane (ACP)", RFC 8994, DOI 10.17487/RFC8994, May 2021, <https://www.rfc-editor.org/info/rfc8994>.

[RFC8995] Pritikin, M., Richardson, M., Eckert, T., Behringer, M., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructure (BRSKI)", RFC 8995, DOI 10.17487/RFC8995, May 2021, <https://www.rfc-editor.org/info/rfc8995>.

12.2. Informative References

[AUTONOMIC-FUNCTION]
Pierre, P. and L. Ciavaglia, "A Day in the Life of an Autonomic Function", Work in Progress, Internet-Draft, draft-peloso-anima-autonomic-function-01, 21 March 2016, <https://datatracker.ietf.org/doc/html/draft-peloso-anima-autonomic-function-01>.

[CBOR-YANG]
Veillette, M., Ed., Petrov, I., Ed., Pelov, A., Bormann, C., and M. Richardson, "CBOR Encoding of Data Modeled with YANG", Work in Progress, Internet-Draft, draft-ietf-core-yang-cbor-18, December 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-core-yang-cbor-18>.

[DEMOLA06] De Mola, F. and R. Quitadamo, "Towards an Agent Model for Future Autonomic Communications", Proceedings of the 7th WOA 2006 Workshop From Objects to Agents 51-59, September 2006.

[GANA13] ETSI, "Autonomic network engineering for the self-managing Future Internet (AFI); Generic Autonomic Network Architecture (An Architectural Reference Model for Autonomic Networking, Cognitive Networking and Self-Management)", GS AFI 002, V1.1.1, April 2013, <https://www.etsi.org/deliver/etsi_gs/AFI/001_099/002/01.01.01_60/gs_afi002v010101p.pdf>.

[HUEBSCHER08]
Huebscher, M. C. and J. A. 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, <https://doi.org/10.1145/1380584.1380585>.

[IBN-CONCEPTS]
Clemm, A., Ciavaglia, L., Granville, L. Z., and J. Tantsura, "Intent-Based Networking - Concepts and Definitions", Work in Progress, Internet-Draft, draft-irtf-nmrg-ibn-concepts-definitions-09, 24 March 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-09>.

[IPJ] Behringer, M., Bormann, C., Carpenter, B. E., Eckert, T., Campos Nobre, J., Jiang, S., Li, Y., and M. C. Richardson, "Autonomic Networking Gets Serious", The Internet Protocol Journal, Volume 24, Issue 3, Page(s) 2 - 18, ISSN 1944-1134, October 2021, <https://ipj.dreamhosters.com/wp-content/uploads/2021/10/243-ipj.pdf>.

[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, Pages 464 - 490, DOI 10.1109/SURV.2011.042711.00078, 2012, <https://doi.org/10.1109/SURV.2011.042711.00078>.

[NFV] ETSI, "Network Functions Virtualisation", SDN and OpenFlow World Congress, October 2012, <https://portal.etsi.org/NFV/NFV_White_Paper.pdf>.

[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>.

[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>.

[RFC8991] Carpenter, B., Liu, B., Ed., Wang, W., and X. Gong, "GeneRic Autonomic Signaling Protocol Application Program Interface (GRASP API)", RFC 8991, DOI 10.17487/RFC8991, May 2021, <https://www.rfc-editor.org/info/rfc8991>.

[RFC8992] Jiang, S., Ed., Du, Z., Carpenter, B., and Q. Sun, "Autonomic IPv6 Edge Prefix Management in Large-Scale Networks", RFC 8992, DOI 10.17487/RFC8992, May 2021, <https://www.rfc-editor.org/info/rfc8992>.

[RFC8993] Behringer, M., Ed., Carpenter, B., Eckert, T., Ciavaglia, L., and J. Nobre, "A Reference Model for Autonomic Networking", RFC 8993, DOI 10.17487/RFC8993, May 2021, <https://www.rfc-editor.org/info/rfc8993>.

[ZSM009-1] ETSI, "Zero-touch network and Service Management (ZSM); Closed-Loop Automation; Part 1: Enablers", GS ZSM 009-1, Version 1.1.1, June 2021, <https://www.etsi.org/deliver/etsi_gs/ZSM/001_099/00901/01.01.01_60/gs_ZSM00901v010101p.pdf>.

Appendix A. Example Logic Flows

This appendix describes generic logic flows that combine to act as an Autonomic Service Agent (ASA) for resource management. Note that these are illustrative examples and are 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 [RFC8990] and its conceptual API [RFC8991].

A complete autonomic function for a distributed resource will 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 [RFC8992]. In this case, an instance of the ASA will exist in each delegating router.

An underlying assumption is that there is an initial source of the resource in question, referred to here as an origin ASA. The other ASAs, known as delegators, obtain supplies of the resource from the origin, 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 origin 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: origin and delegator. When running as an origin, 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, acquires the policy parameters by GRASP synchronization, and delegates quantities of the resource to consumers that request it. Both as an origin 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 origin will be shared among all the delegators according to demand.

In theory, a network could include any number of origins and any number of delegators, with the only condition being that each origin’s initial resource pool is unique. A realistic scenario is to have exactly one origin and as many delegators as you like. A scenario with no origin 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 an origin exits, its entire spare pool is lost. The logic for using stable storage and for crash recovery is not included in the pseudocode below, which focuses on communication between ASAs. Since GRASP operations are not intrinsically idempotent, data integrity during failure scenarios is the responsibility of the ASA designer. This is a complex topic in its own right that is not discussed in the present document.

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 occurs.

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

* resource_pool: an ordered list of available resources, for example. 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.

* delegated_list: where a delegator stores the resources it has given to subsidiary devices.

Possible main logic flows are below, using a threaded implementation model. As noted above, alternative approaches to asynchronous operations are possible. 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 origin
if origin:
    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 origin:
    Start FLOODER thread to flood EX1.Params
    Start SYNCHRONIZER listener for EX1.Params
Start MAIN_NEGOTIATOR thread for EX1.Resource
if not origin:
    Obtain value of EX1.Params from GRASP flood or synchronization
    Start DELEGATOR thread
Start GARBAGE_COLLECTOR thread
good_peer = none
do forever:
    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 negotiation request
        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      #remember this choice
            else:
                Send M_END + O_DECLINE #negotiation failed
                good_peer = none       #forget this choice
    sleep() #periodic timer suitable for application scenario

MAIN_NEGOTIATOR thread:

do forever:
    grasp.listen_negotiate("EX1.Resource")
    #i.e., wait for negotiation request
    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
        Return resource to resource_pool
else:
    Send M_END + O_DECLINE #negotiation failed
#thread exits

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 #atomic
            Record in delegated_list      #operation
```

```
          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() #periodic timer suitable for application scenario

   GARBAGE_COLLECTOR thread:

   do forever:
       Search resource_pool for adjacent resources
       Merge adjacent resources
       sleep() #periodic timer suitable for application scenario
```

Acknowledgements

Valuable comments were received from Michael Behringer, Menachem Dodge, Martin DÃ¼rst, Toerless Eckert, Thomas Fossati, Alex Galis, Bing Liu, Benno Overeinder, Michael Richardson, Rob Wilton, and other IESG members.

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
Rakuten Mobile
Paris
France
Email: laurent.ciavaglia@rakuten.com

Sheng Jiang
Huawei Technologies Co., Ltd
Q14 Huawei Campus
156 Beiqing Road
Hai-Dian District
Beijing
100095
China
Email: jiangsheng@huawei.com

Pierre Peloso
Nokia
Villarceaux
91460 Nozay
France
Email: pierre.peloso@nokia.com