Internet Engineering Task Force (IETF)
Request for Comments: 8991
Category: Informational
ISSN: 2070-1721

B. Carpenter
Univ. of Auckland
B. Liu, Ed.
Huawei Technologies
W. Wang
X. Gong
BUPT University

May 2021

GeneRic Autonomic Signaling Protocol Application Program Interface (GRASP API)

Abstract

This document is a conceptual outline of an Application Programming Interface (API) for the GeneRic Autonomic Signaling Protocol (GRASP). Such an API is needed for Autonomic Service Agents (ASAs) calling the GRASP protocol module to exchange Autonomic Network messages with other ASAs. Since GRASP is designed to support asynchronous operations, the API will need to be adapted according to the support for asynchronicity in various programming languages and operating systems.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8991.

Table of Contents

1. Introduction

As defined in [RFC8993], the Autonomic Service Agent (ASA) is the atomic entity of an autonomic function, and it is instantiated on autonomic nodes. These nodes are members of a secure Autonomic Control Plane (ACP) such as defined by [RFC8994].

When ASAs communicate with each other, they should use the GeneRic Autonomic Signaling Protocol (GRASP) [RFC8990]. GRASP relies on the message confidentiality and integrity provided by the ACP; a consequence of this is that all nodes in a given Autonomic Network share the same trust boundary, i.e., the boundary of the ACP. Nodes that have not successfully joined the ACP cannot send, receive, or intercept GRASP messages via the ACP and cannot usurp ACP addresses. An ASA runs in an ACP node and therefore benefits from the node's security properties when transmitting over the ACP, i.e., message integrity, message confidentiality, and the fact that unauthorized nodes cannot join the ACP. All ASAs within a given Autonomic Network therefore trust each other's messages. For these reasons, the API defined in this document has no explicit security features.

An important feature of GRASP is the concept of a GRASP objective. This is a data structure encoded, like all GRASP messages, in Concise Binary Object Representation (CBOR) [RFC8949]. Its main contents are a name and a value, explained at more length in the Terminology section of [RFC8990]. When an objective is passed from one ASA to another using GRASP, its value is either conveyed in one direction (by a process of synchronization or flooding) or negotiated bilaterally. The semantics of the value are opaque to GRASP and therefore to the API. Each objective must be accurately specified in a dedicated specification, as discussed in "Objective Options" (Section 2.10 of [RFC8990]). In particular, the specification will define the syntax and semantics of the value of the objective, whether and how it supports a negotiation process, whether it supports a dry-run mode, and any other details needed for interoperability. The use of CBOR, with Concise Data Definition Language (CDDL) [RFC8610] as the data definition language, allows the value to be passed between ASAs regardless of the programming languages in use. Data storage and consistency during negotiation are the responsibility of the ASAs involved. Additionally, GRASP needs to cache the latest values of objectives that are received by flooding.

As Figure 1 shows, a GRASP implementation could contain several sub-layers. The bottom layer is the GRASP base protocol module, which is only responsible for sending and receiving GRASP messages and

maintaining shared data structures. Above that is the basic API described in this document. The upper layer contains some extended API functions based upon the GRASP basic protocol. For example, [GRASP-DISTRIB] describes a possible extended function.

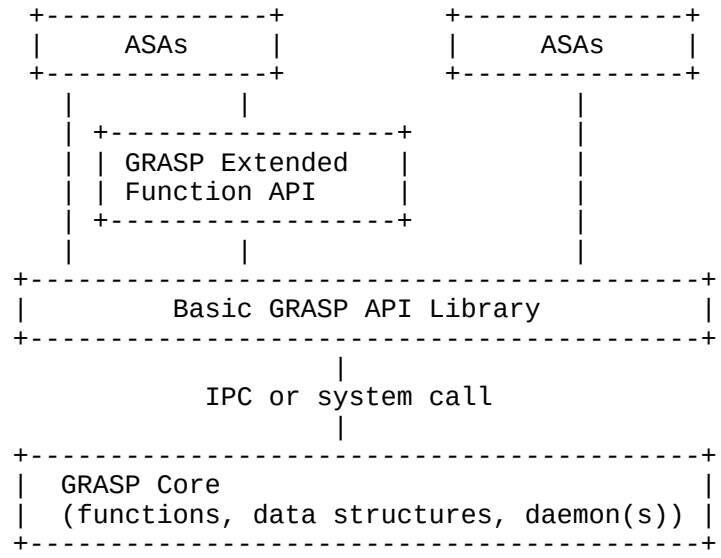

Figure 1: Software Layout

Multiple ASAs in a single node will share the same instance of GRASP, much as multiple applications share a single TCP/IP stack. This aspect is hidden from individual ASAs by the API and is not further discussed here.

It is desirable that ASAs be designed as portable user-space programs using a system-independent API. In many implementations, the GRASP code will therefore be split between user space and kernel space. In user space, library functions provide the API and communicate directly with ASAs. In kernel space, a daemon, or a set of sub-services, provides GRASP core functions that are independent of specific ASAs, such as multicast handling and relaying, and common data structures, such as the discovery cache. The GRASP API library would need to communicate with the GRASP core via an interprocess communication (IPC) or a system call mechanism. The details of this are system-dependent.

Both the GRASP library and the extended function modules should be available to the ASAs. However, since the extended functions are expected to be added in an incremental manner, they will be the subject of future documents. This document only describes the basic GRASP API.

The functions provided by the API do not map one-to-one onto GRASP messages. Rather, they are intended to offer convenient support for message sequences (such as a discovery request followed by responses from several peers or a negotiation request followed by various possible responses). This choice was made to assist ASA programmers in writing code based on their application requirements rather than needing to understand protocol details.

In addition to containing the autonomic infrastructure components described in [RFC8994] and [RFC8995], a simple autonomic node might contain very few ASAs. Such a node might directly integrate a GRASP protocol stack in its code and therefore not require this API to be installed. However, the programmer would need a deeper understanding of the GRASP protocol than what is needed to use the API.

This document gives a conceptual outline of the API. It is not a

formal specification for any particular programming language or operating system, and it is expected that details will be clarified in individual implementations.

2. GRASP API for ASA

2.1. Design Assumptions

The design assumes that an ASA needs to call a separate GRASP implementation. The latter handles protocol details (security, sending and listening for GRASP messages, waiting, caching discovery results, negotiation looping, sending and receiving synchronization data, etc.) but understands nothing about individual GRASP objectives (see Section 2.10 of [RFC8990]). The semantics of objectives are unknown to the GRASP protocol and are handled only by the ASAs. Thus, this is an abstract API for use by ASAs. Individual language bindings should be defined in separate documents.

Different ASAs may utilize GRASP features differently, by using GRASP for:

* discovery purposes only.

* negotiation but only as an initiator (client).

* negotiation but only as a responder.

* negotiation as an initiator or responder.

* synchronization but only as an initiator (recipient).

* synchronization but only as a responder and/or flooder.

* synchronization as an initiator, responder, and/or flooder.

The API also assumes that one ASA may support multiple objectives. Nothing prevents an ASA from supporting some objectives for synchronization and others for negotiation.

The API design assumes that the operating system and programming language provide a mechanism for simultaneous asynchronous operations. This is discussed in detail in Section 2.2.

A few items are out of scope in this version, since practical experience is required before including them:

* Authorization of ASAs is not defined as part of GRASP and is a subject for future study.

* User-supplied explicit locators for an objective are not supported. The GRASP core will supply the locator, using the IP address of the node concerned.

* The rapid mode of GRASP (Section 2.5.4 of [RFC8990]) is not supported.

2.2. Asynchronous Operations

GRASP depends on asynchronous operations and wait states, and some of its messages are not idempotent, meaning that repeating a message may cause repeated changes of state in the recipient ASA. Many ASAs will need to support several concurrent operations; for example, an ASA might need to negotiate one objective with a peer while discovering and synchronizing a different objective with a different peer. Alternatively, an ASA that acts as a resource manager might need to run simultaneous negotiations for a given objective with multiple

different peers. Such an ASA will probably need to support uninterruptible atomic changes to its internal data structures, using a mechanism provided by the operating system and programming language in use.

2.2.1. Alternative Asynchronous Mechanisms

Some ASAs need to support asynchronous operations; therefore, the GRASP core must do so. Depending on both the operating system and the programming language in use, there are various techniques for such parallel operations, three of which we consider here: multithreading, an event loop structure using polling, and an event loop structure using callback functions.

1. In multithreading, the operating system and language will provide the necessary support for asynchronous operations, including creation of new threads, context switching between threads, queues, locks, and implicit wait states. In this case, API calls can be treated as simple synchronous function calls within their own thread, even if the function includes wait states, blocking, and queueing. Concurrent operations will each run in their own threads. For example, the discover() call may not return until discovery results have arrived or a timeout has occurred. If the ASA has other work to do, the discover() call must be in a thread of its own.

2. In an event loop implementation with polling, blocking calls are not acceptable. Therefore, all calls must be non-blocking, and the main loop could support multiple GRASP sessions in parallel by repeatedly polling each one for a change of state. To facilitate this, the API implementation would provide non-blocking versions of all the functions that otherwise involve blocking and queueing. In these calls, a 'noReply' code will be returned by each call instead of blocking, until such time as the event for which it is waiting (or a failure) has occurred. Thus, for example, discover() would return 'noReply' instead of waiting until discovery has succeeded or timed out. The discover() call would be repeated in every cycle of the main loop until it completes. Effectively, it becomes a polling call.

3. It was noted earlier that some GRASP messages are not idempotent; in particular, this applies to each step in a negotiation session -- sending the same message twice might produce unintended side effects. This is not affected by event loop polling: repeating a call after a 'noReply' does not repeat a message; it simply checks whether a reply has been received.

4. In an event loop implementation with callbacks, the ASA programmer would provide a callback function for each asynchronous operation. This would be called asynchronously when a reply is received or a failure such as a timeout occurs.

2.2.2. Multiple Negotiation Scenario

The design of GRASP allows the following scenario. Consider an ASA "A" that acts as a resource allocator for some objective. An ASA "B" launches a negotiation with "A" to obtain or release a quantity of the resource. While this negotiation is under way, "B" chooses to launch a second simultaneous negotiation with "A" for a different quantity of the same resource. "A" must therefore conduct two separate negotiation sessions at the same time with the same peer and must not mix them up.

Note that ASAs could be designed to avoid such a scenario, i.e., restricted to exactly one negotiation session at a time for a given objective, but this would be a voluntary restriction not required by

the GRASP protocol. In fact, GRASP assumes that any ASA managing a resource may need to conduct multiple parallel negotiations, possibly with the same peer. Communication patterns could be very complex, with a group of ASAs overlapping negotiations among themselves, as described in [ANIMA-COORD]. Therefore, the API design allows for such scenarios.

In the callback model, for the scenario just described, the ASAs "A" and "B" will each provide two instances of the callback function, one for each session. For this reason, each ASA must be able to distinguish the two sessions, and the peer's IP address is not sufficient for this. It is also not safe to rely on transport port numbers for this, since future variants of GRASP might use shared ports rather than a separate port per session. Hence, the GRASP design includes a Session ID. Thus, when necessary, a session handle (see the next section) is used in the API to distinguish simultaneous GRASP sessions from each other, so that any number of sessions may proceed asynchronously in parallel.

2.2.3. Overlapping Sessions and Operations

A GRASP session consists of a finite sequence of messages (for discovery, synchronization, or negotiation) between two ASAs. It is uniquely identified on the wire by a pseudorandom Session ID plus the IP address of the initiator of the session. Further details are given in "Session Identifier (Session ID)" (Section 2.7 of [RFC8990]).

On the first call in a new GRASP session, the API returns a 'session_handle' handle that uniquely identifies the session within the API, so that multiple overlapping sessions can be distinguished. A likely implementation is to form the handle from the underlying GRASP Session ID and IP address. This handle must be used in all subsequent calls for the same session. Also see Section 2.3.2.8.

An additional mechanism that might increase efficiency for polling implementations is to add a general call, say notify(), which would check the status of all outstanding operations for the calling ASA and return the session_handle values for all sessions that have changed state. This would eliminate the need for repeated calls to the individual functions returning a 'noReply'. This call is not described below as the details are likely to be implementation specific.

An implication of the above for all GRASP implementations is that the GRASP core must keep state for each GRASP operation in progress, most likely keyed by the GRASP Session ID and the GRASP source address of the session initiator. Even in a threaded implementation, the GRASP core will need such state internally. The session_handle parameter exposes this aspect of the implementation.

2.2.4. Session Termination

GRASP sessions may terminate for numerous reasons. A session ends when discovery succeeds or times out, negotiation succeeds or fails, a synchronization result is delivered, the other end fails to respond before a timeout expires, a loop count expires, or a network socket error occurs. Note that a timeout at one end of a session might result in a timeout or a socket error at the other end, since GRASP does not send error messages in this case. In all cases, the API will return an appropriate code to the caller, which should then release any reserved resources. After failure cases, the GRASP specification recommends an exponential backoff before retrying.

2.3. API Definition

2.3.1. Overview of Functions

The functions provided by the API fall into several groups:

Registration: These functions allow an ASA to register itself with the GRASP core and allow a registered ASA to register the GRASP objectives that it will manipulate.

Discovery: This function allows an ASA that needs to initiate negotiation or synchronization of a particular objective to discover a peer willing to respond.

Negotiation: These functions allow an ASA to act as an initiator (requester) or responder (listener) for a GRASP negotiation session. After initiation, negotiation is a symmetric process, so most of the functions can be used by either party.

Synchronization: These functions allow an ASA to act as an initiator (requester) or responder (listener and data source) for a GRASP synchronization session.

Flooding: These functions allow an ASA to send and receive an objective that is flooded to all nodes of the ACP.

Some example logic flows for a resource management ASA are given in [ASA-GUIDE], which may be of help in understanding the following descriptions. The next section describes parameters and data structures used in multiple API calls. The following sections describe various groups of function APIs. Those APIs that do not list asynchronous mechanisms are implicitly synchronous in their behavior.

2.3.2. Parameters and Data Structures

2.3.2.1. Integers

In this API, integers are assumed to be 32-bit unsigned integers (uint32_t) unless otherwise indicated.

2.3.2.2. Errorcode

All functions in the API have an unsigned ’errorcode’ integer as their return value (the first return value in languages that allow multiple return values). An errorcode of zero indicates success. Any other value indicates failure of some kind. The first three errorcodes have special importance:

1 - Declined: used to indicate that the other end has sent a GRASP Negotiation End message (M_END) with a Decline option (O_DECLINE).

2 - No reply: used in non-blocking calls to indicate that the other end has sent no reply so far (see Section 2.2).

3 - Unspecified error: used when no more specific error codes apply.

Appendix A gives a full list of currently suggested error codes, based on implementation experience. While there is no absolute requirement for all implementations to use the same error codes, this is highly recommended for portability of applications.

2.3.2.3. Timeout

Whenever a ’timeout’ parameter appears, it is an unsigned integer expressed in milliseconds. If it is zero, the GRASP default timeout (GRASP_DEF_TIMEOUT; see [RFC8990]) will apply. An exception is the discover() function, which has a different interpretation of a zero

timeout. If no response is received before the timeout expires, the call will fail unless otherwise noted.

2.3.2.4. Objective

An 'objective' parameter is a data structure with the following components:

name (UTF-8 string): The objective's name

neg (Boolean flag): True if objective supports negotiation (default False)

synch (Boolean flag): True if objective supports synchronization (default False)

dry (Boolean flag): True if objective supports dry-run negotiation (default False)

Note 1: Only one of 'synch' or 'neg' may be True.
Note 2: 'dry' must not be True unless 'neg' is also True.
Note 3: In some programming languages, the preferred implementation may be to represent the Boolean flags as bits in a single byte, which is how they are encoded in GRASP messages. In other languages, an enumeration might be preferable.

loop_count (unsigned integer, uint8_t): Limit on negotiation steps, etc. (default GRASP_DEF_LOOPCT; see [RFC8990]). The 'loop_count' is set to a suitable value by the initiator of a negotiation, to prevent indefinite loops. It is also used to limit the propagation of discovery and flood messages.

value: A specific data structure expressing the value of the objective. The format is language dependent, with the constraint that it can be validly represented in CBOR [RFC8949].

An important advantage of CBOR is that the value of an objective can be completely opaque to the GRASP core yet pass transparently through it to and from the ASA. Although the GRASP core must validate the format and syntax of GRASP messages, it cannot validate the value of an objective; all it can do is detect malformed CBOR. The handling of decoding errors depends on the CBOR library in use, but a corresponding error code ('CBORfail') is defined in the API and will be returned to the ASA if a faulty message can be assigned to a current GRASP session. However, it is the responsibility of each ASA to validate the value of a received objective, as discussed in Section 5.3 of [RFC8949]. If the programming language in use is suitably object-oriented, the GRASP API may deserialize the value and present it to the ASA as an object. If not, it will be presented as a CBOR data item. In all cases, the syntax and semantics of the objective value are the responsibility of the ASA.

A requirement for all language mappings and all API implementations is that, regardless of what other options exist for a language-specific representation of the value, there is always an option to use a raw CBOR data item as the value. The API will then wrap this with CBOR Tag 24 as an encoded CBOR data item for transmission via GRASP, and unwrap it after reception. By this means, ASAs will be able to communicate regardless of programming language.

The 'name' and 'value' fields are of variable length. GRASP does not set a maximum length for these fields, but only for the total length of a GRASP message. Implementations might impose length limits.

An example data structure definition for an objective in the C language, using at least the C99 version, and assuming the use of a particular CBOR library [libcbor], is:

```
 typedef struct {
    unsigned char *name;
    uint8_t flags;            // flag bits as defined by GRASP
    uint8_t loop_count;
    uint32_t value_size;      // size of value in bytes
    cbor_mutable_data cbor_value;
                             // CBOR bytestring (libcbor/cbor/data.h)
                } objective;
```

An example data structure definition for an objective in the Python language (version 3.4 or later) is:

```
 class objective:
    """A GRASP objective"""
    def __init__(self, name):
        self.name = name        #Unique name (string)
        self.negotiate = False  #True if negotiation supported
        self.dryrun = False     #True if dry-run supported
        self.synch = False      #True if synchronization supported
        self.loop_count = GRASP_DEF_LOOPCT  # Default starting value
        self.value = None       #Place holder; any Python object
```

2.3.2.5. asa_locator

An 'asa_locator' parameter is a data structure with the following contents:

locator: The actual locator, either an IP address or an ASCII string.

ifi (unsigned integer): The interface identifier index via which this was discovered (of limited use to most ASAs).

expire (system dependent type): The time on the local system clock when this locator will expire from the cache.

The following covers all locator types currently supported by GRASP:

* is_ipaddress (Boolean) - True if the locator is an IP address.

* is_fqdn (Boolean) - True if the locator is a Fully Qualified Domain Name (FQDN).

* is_uri (Boolean) - True if the locator is a URI.

These options are mutually exclusive. Depending on the programming language, they could be represented as a bit pattern or an enumeration.

diverted (Boolean): True if the locator was discovered via a Divert option.

protocol (unsigned integer): Applicable transport protocol (IPPROTO_TCP or IPPROTO_UDP). These constants are defined in the CDDL specification of GRASP [RFC8990].

port (unsigned integer): Applicable port number.

The 'locator' field is of variable length in the case of an FQDN or a URI. GRASP does not set a maximum length for this field, but only for the total length of a GRASP message. Implementations might

impose length limits.

It should be noted that when one ASA discovers the asa_locator of another, there is no explicit authentication mechanism. In accordance with the trust model provided by the secure ACP, ASAs are presumed to provide correct locators in response to discovery. See "Locator Options" (Section 2.9.5 of [RFC8990]) for further details.

2.3.2.6. Tagged_objective

A 'tagged_objective' parameter is a data structure with the following contents:

objective: An objective.

locator: The asa_locator associated with the objective, or a null value.

2.3.2.7. asa_handle

Although an authentication and authorization scheme for ASAs has not been defined, the API provides a very simple hook for such a scheme. When an ASA starts up, it registers itself with the GRASP core, which provides it with an opaque handle that, although not cryptographically protected, would be difficult for a third party to predict. The ASA must present this handle in future calls. This mechanism will prevent some elementary errors or trivial attacks such as an ASA manipulating an objective it has not registered to use.

Thus, in most calls, an 'asa_handle' parameter is required. It is generated when an ASA first registers with GRASP, and the ASA must then store the asa_handle and use it in every subsequent GRASP call. Any call in which an invalid handle is presented will fail. It is an up to 32-bit opaque value (for example, represented as a uint32_t, depending on the language). Since it is only used locally, and not in GRASP messages, it is only required to be unique within the local GRASP instance. It is valid until the ASA terminates. It should be unpredictable; a possible implementation is to use the same mechanism that GRASP uses to generate Session IDs (see Section 2.3.2.8).

2.3.2.8. Session_handle and Callbacks

In some calls, a 'session_handle' parameter is required. This is an opaque data structure as far as the ASA is concerned, used to identify calls to the API as belonging to a specific GRASP session (see Section 2.2.3). It will be provided as a parameter in callback functions. As well as distinguishing calls from different sessions, it also allows GRASP to detect and ignore calls from non-existent or timed-out sessions.

In an event loop implementation, callback functions (Section 2.2.1) may be supported for all API functions that involve waiting for a remote operation:

discover() whose callback would be discovery_received().

request_negotiate() whose callback would be negotiate_step_received().

negotiate_step() whose callback would be negotiate_step_received().

listen_negotiate() whose callback would be negotiate_step_received().

synchronize() whose callback would be synchronization_received().

Further details of callbacks are implementation dependent.

2.3.3. Registration

These functions are used to register an ASA, and the objectives that it modifies, with the GRASP module. In the absence of an authorization model, these functions are very simple, but they will avoid multiple ASAs choosing the same name and will prevent multiple ASAs manipulating the same objective. If an authorization model is added to GRASP, these API calls would need to be modified accordingly.

* register_asa()

  All ASAs must use this call before issuing any other API calls.

  - Input parameter:

    name of the ASA (UTF-8 string)

  - Return value:

    errorcode (unsigned integer)

    asa_handle (unsigned integer)

  - This initializes the state in the GRASP module for the calling entity (the ASA). In the case of success, an 'asa_handle' is returned, which the ASA must present in all subsequent calls. In the case of failure, the ASA has not been authorized and cannot operate. The 'asa_handle' value is undefined.

* deregister_asa()

  - Input parameters:

    asa_handle (unsigned integer)

    name of the ASA (UTF-8 string)

  - Return value:

    errorcode (unsigned integer)

  - This removes all state in the GRASP module for the calling entity (the ASA) and deregisters any objectives it has registered. Note that these actions must also happen automatically if an ASA exits.

  - Note -- the ASA name is, strictly speaking, redundant in this call but is present to detect and reject erroneous deregistrations.

* register_objective()

  ASAs must use this call for any objective whose value they need to transmit by negotiation, synchronization, or flooding.

  - Input parameters:

    asa_handle (unsigned integer)

    objective (structure)

    ttl (unsigned integer -- default GRASP_DEF_TIMEOUT)

discoverable (Boolean -- default False)

overlap (Boolean -- default False)

local (Boolean -- default False)

- Return value:

errorcode (unsigned integer)

- This registers an objective that this ASA may modify and transmit to other ASAs by flooding or negotiation. It is not necessary to register an objective that is only received by GRASP synchronization or flooding. The 'objective' becomes a candidate for discovery. However, discovery responses should not be enabled until the ASA calls listen_negotiate() or listen_synchronize(), showing that it is able to act as a responder. The ASA may negotiate the objective or send synchronization or flood data. Registration is not needed for "read-only" operations, i.e., the ASA only wants to receive synchronization or flooded data for the objective concerned.

- The 'ttl' parameter is the valid lifetime (time to live) in milliseconds of any discovery response generated for this objective. The default value should be the GRASP default timeout (GRASP_DEF_TIMEOUT; see [RFC8990]).

- If the parameter 'discoverable' is True, the objective is immediately discoverable. This is intended for objectives that are only defined for GRASP discovery and that do not support negotiation or synchronization.

- If the parameter 'overlap' is True, more than one ASA may register this objective in the same GRASP instance. This is of value for life cycle management of ASAs [ASA-GUIDE] and must be used consistently for a given objective (always True or always False).

- If the parameter 'local' is True, discovery must return a link-local address. This feature is for objectives that must be restricted to the local link.

- This call may be repeated for multiple objectives.

* deregister_objective()

- Input parameters:

asa_handle (unsigned integer)

objective (structure)

- Return value:

errorcode (unsigned integer)

- The 'objective' must have been registered by the calling ASA; if not, this call fails. Otherwise, it removes all state in the GRASP module for the given objective.

2.3.4. Discovery

* discover()

This function may be used by any ASA to discover peers handling a

given objective.

- Input parameters:

  asa_handle (unsigned integer)

  objective (structure)

  timeout (unsigned integer)

  minimum_TTL (unsigned integer)

- Return values:

  errorcode (unsigned integer)

  locator_list (structure)

- This returns a list of discovered 'asa_locators' for the given objective. An empty list means that no locators were discovered within the timeout. Note that this structure includes all the fields described in Section 2.3.2.5.

- The parameter 'minimum_TTL' must be greater than or equal to zero. Any locally cached locators for the objective whose remaining time to live in milliseconds is less than or equal to 'minimum_TTL' are deleted first. Thus, 'minimum_TTL' = 0 will flush all entries. Note that this will not affect sessions already in progress using the deleted locators.

- If the parameter 'timeout' is zero, any remaining locally cached locators for the objective are returned immediately, and no other action is taken. (Thus, a call with 'minimum_TTL' and 'timeout' both equal to zero is pointless.)

- If the parameter 'timeout' is greater than zero, GRASP discovery is performed, and all results obtained before the timeout in milliseconds expires are returned. If no results are obtained, an empty list is returned after the timeout. That is not an error condition. GRASP discovery is not a deterministic process. If there are multiple nodes handling an objective, none, some, or all of them will be discovered before the timeout expires.

- Asynchronous Mechanisms:

  Threaded implementation: This should be called in a separate thread if asynchronous operation is required.

  Event loop implementation: An additional in/out 'session_handle' parameter is used. If the 'errorcode' parameter has the value 2 ('noReply'), no response has been received so far. The 'session_handle' parameter must be presented in subsequent calls. A callback may be used in the case of a non-zero timeout.

2.3.5. Negotiation

Since the negotiation mechanism is different from a typical client/server exchange, Figure 2 illustrates the sequence of calls and GRASP messages in a negotiation. Note that after the first protocol exchange, the process is symmetrical, with negotiating steps strictly alternating between the two sides. Either side can end the negotiation. Also, the side that is due to respond next can insert a delay at any time, to extend the other side's timeout. This would be used, for example, if an ASA needed to negotiate with a third party

before continuing with the current negotiation.

The loop count embedded in the objective that is the subject of negotiation is initialized by the ASA that starts a negotiation and is then decremented by the GRASP core at each step, prior to sending each M_NEGOTIATE message. If it reaches zero, the negotiation will fail, and each side will receive an error code.

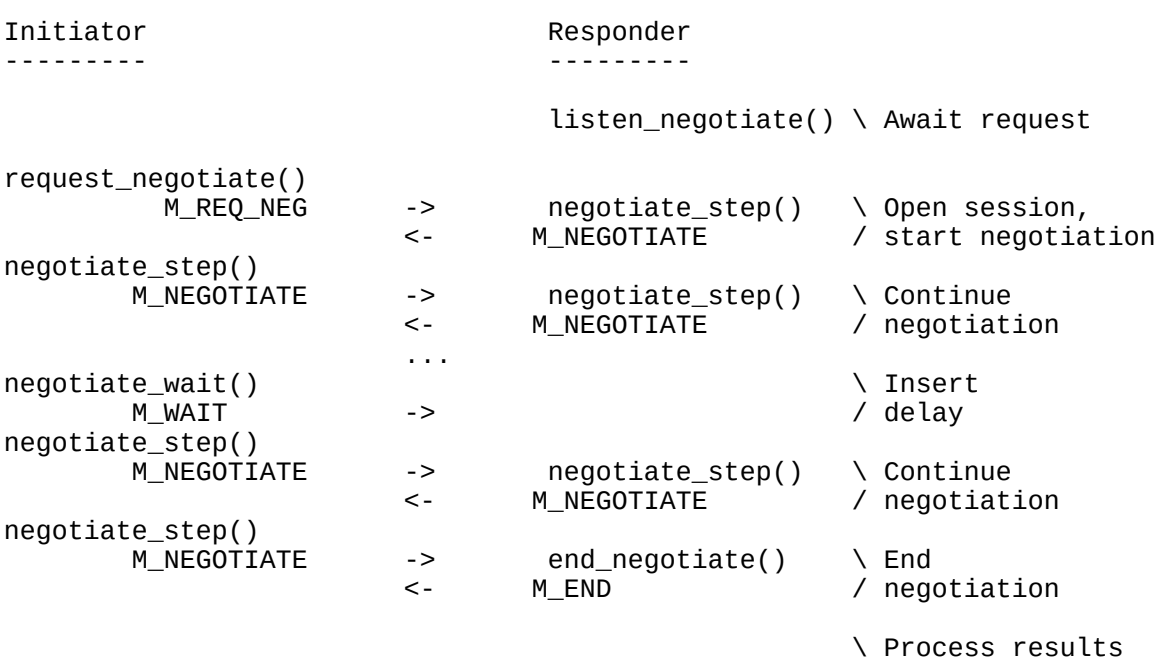

Figure 2: Negotiation Sequence

As the negotiation proceeds, each side will update the value of the objective in accordance with its particular semantics, defined in the specification of the objective. Although many objectives will have values that can be ordered, so that negotiation can be a simple bidding process, it is not a requirement.

Failure to agree, a timeout, or loop count exhaustion may all end a negotiation session, but none of these cases are protocol failures.

* request_negotiate()

  This function is used by any ASA to initiate negotiation of a GRASP objective as a requester (client).

  - Input parameters:

    asa_handle (unsigned integer)

    objective (structure)

    peer (asa_locator)

    timeout (unsigned integer)

  - Return values:

    errorcode (unsigned integer)

    session_handle (structure) (undefined unless successful)

    proffered_objective (structure) (undefined unless successful)

reason (string) (empty unless negotiation declined)

- This function opens a negotiation session between two ASAs. Note that GRASP currently does not support multiparty negotiation, which would need to be added as an extended function.

- The 'objective' parameter must include the requested value, and its loop count should be set to a suitable starting value by the ASA. If not, the GRASP default will apply.

- Note that a given negotiation session may or may not be a dry-run negotiation; the two modes must not be mixed in a single session.

- The 'peer' parameter is the target node; it must be an 'asa_locator' as returned by discover(). If 'peer' is null, GRASP discovery is automatically performed first to find a suitable peer (i.e., any node that supports the objective in question).

- The 'timeout' parameter is described in Section 2.3.2.3.

- If the 'errorcode' return value is 0, the negotiation has successfully started. There are then two cases:

  1. The 'session_handle' parameter is null. In this case, the negotiation has succeeded with one exchange of messages, and the peer has accepted the request. The returned 'proffered_objective' contains the value accepted by the peer, which is therefore equal to the value in the requested 'objective'. For this reason, no session handle is needed, since the session has ended.

  2. The 'session_handle' parameter is not null. In this case, negotiation must continue. The 'session_handle' must be presented in all subsequent negotiation steps. The returned 'proffered_objective' contains the first value proffered by the negotiation peer in the first exchange of messages; in other words, it is a counter-offer. The contents of this instance of the objective must be used to prepare the next negotiation step (see negotiate_step() below) because it contains the updated loop count, sent by the negotiation peer. The GRASP code automatically decrements the loop count by 1 at each step and returns an error if it becomes zero. Since this terminates the negotiation, the other end will experience a timeout, which will terminate the other end of the session.

     This function must be followed by calls to 'negotiate_step' and/or 'negotiate_wait' and/or 'end_negotiate' until the negotiation ends. 'request_negotiate' may then be called again to start a new negotiation.

- If the 'errorcode' parameter has the value 1 ('declined'), the negotiation has been declined by the peer (M_END and O_DECLINE features of GRASP). The 'reason' string is then available for information and diagnostic use, but it may be a null string. For this and any other error code, an exponential backoff is recommended before any retry (see Section 3).

- Asynchronous Mechanisms:

  Threaded implementation: This should be called in a separate thread if asynchronous operation is required.

Event loop implementation: The 'session_handle' parameter is used to distinguish multiple simultaneous sessions. If the 'errorcode' parameter has the value 2 ('noReply'), no response has been received so far. The 'session_handle' parameter must be presented in subsequent calls.

- Use of dry-run mode must be consistent within a GRASP session. The state of the 'dry' flag in the initial request_negotiate() call must be the same in all subsequent negotiation steps of the same session. The semantics of the dry-run mode are built into the ASA; GRASP merely carries the flag bit.

- Special note for the ACP infrastructure ASA: It is likely that this ASA will need to discover and negotiate with its peers in each of its on-link neighbors. It will therefore need to know not only the link-local IP address but also the physical interface and transport port for connecting to each neighbor. One implementation approach to this is to include these details in the 'session_handle' data structure, which is opaque to normal ASAs.

* listen_negotiate()

This function is used by an ASA to start acting as a negotiation responder (listener) for a given GRASP objective.

- Input parameters:

  asa_handle (unsigned integer)

  objective (structure)

- Return values:

  errorcode (unsigned integer)

  session_handle (structure) (undefined unless successful)

  requested_objective (structure) (undefined unless successful)

- This function instructs GRASP to listen for negotiation requests for the given 'objective'. It also enables discovery responses for the objective, as mentioned under register_objective() in Section 2.3.3.

- Asynchronous Mechanisms:

  Threaded implementation: It will block waiting for an incoming request, so it should be called in a separate thread if asynchronous operation is required. Unless there is an unexpected failure, this call only returns after an incoming negotiation request. If the ASA supports multiple simultaneous transactions, a new sub-thread must be spawned for each new session, so that listen_negotiate() can be called again immediately.

  Event loop implementation: A 'session_handle' parameter is used to distinguish individual sessions. If the ASA supports multiple simultaneous transactions, a new event must be inserted in the event loop for each new session, so that listen_negotiate() can be reactivated immediately.

- This call only returns (threaded model) or triggers (event loop) after an incoming negotiation request. When this occurs, 'requested_objective' contains the first value requested by the

negotiation peer. The contents of this instance of the objective must be used in the subsequent negotiation call because it contains the loop count sent by the negotiation peer. The 'session_handle' must be presented in all subsequent negotiation steps.

- This function must be followed by calls to 'negotiate_step' and/or 'negotiate_wait' and/or 'end_negotiate' until the negotiation ends.

- If an ASA is capable of handling multiple negotiations simultaneously, it may call 'listen_negotiate' simultaneously from multiple threads, or insert multiple events. The API and GRASP implementation must support re-entrant use of the listening state and the negotiation calls. Simultaneous sessions will be distinguished by the threads or events themselves, the GRASP session handles, and the underlying unicast transport sockets.

* stop_listen_negotiate()

This function is used by an ASA to stop acting as a responder (listener) for a given GRASP objective.

- Input parameters:

  asa_handle (unsigned integer)

  objective (structure)

- Return value:

  errorcode (unsigned integer)

- Instructs GRASP to stop listening for negotiation requests for the given objective, i.e., cancels 'listen_negotiate'.

- Asynchronous Mechanisms:

  Threaded implementation: Must be called from a different thread than 'listen_negotiate'.

  Event loop implementation: No special considerations.

* negotiate_step()

This function is used by either ASA in a negotiation session to make the next step in negotiation.

- Input parameters:

  asa_handle (unsigned integer)

  session_handle (structure)

  objective (structure)

  timeout (unsigned integer) as described in Section 2.3.2.3

- Return values:

  Exactly as for 'request_negotiate'

- Executes the next negotiation step with the peer. The 'objective' parameter contains the next value being proffered by the ASA in this step. It must also contain the latest

'loop_count' value received from request_negotiate() or negotiate_step().

- Asynchronous Mechanisms:

  Threaded implementation: Usually called in the same thread as the preceding 'request_negotiate' or 'listen_negotiate', with the same value of 'session_handle'.

  Event loop implementation: Must use the same value of 'session_handle' returned by the preceding 'request_negotiate' or 'listen_negotiate'.

* negotiate_wait()

This function is used by either ASA in a negotiation session to delay the next step in negotiation.

- Input parameters:

  asa_handle (unsigned integer)

  session_handle (structure)

  timeout (unsigned integer)

- Return value:

  errorcode (unsigned integer)

- Requests the remote peer to delay the negotiation session by 'timeout' milliseconds, thereby extending the original timeout. This function simply triggers a GRASP Confirm Waiting message (see [RFC8990] for details).

- Asynchronous Mechanisms:

  Threaded implementation: Called in the same thread as the preceding 'request_negotiate' or 'listen_negotiate', with the same value of 'session_handle'.

  Event loop implementation: Must use the same value of 'session_handle' returned by the preceding 'request_negotiate' or 'listen_negotiate'.

* end_negotiate()

This function is used by either ASA in a negotiation session to end a negotiation.

- Input parameters:

  asa_handle (unsigned integer)

  session_handle (structure)

  result (Boolean)

  reason (UTF-8 string)

- Return value:

  errorcode (unsigned integer)

- End the negotiation session:

'result' = True for accept (successful negotiation), and False for decline (failed negotiation).

'reason' = string describing reason for decline (may be null; ignored if accept).

- Asynchronous Mechanisms:

Threaded implementation: Called in the same thread as the preceding 'request_negotiate' or 'listen_negotiate', with the same value of 'session_handle'.

Event loop implementation: Must use the same value of 'session_handle' returned by the preceding 'request_negotiate' or 'listen_negotiate'.

2.3.6. Synchronization and Flooding

* synchronize()

This function is used by any ASA to cause synchronization of a GRASP objective as a requester (client).

- Input parameters:

asa_handle (unsigned integer)

objective (structure)

peer (asa_locator)

timeout (unsigned integer)

- Return values:

errorcode (unsigned integer)

result (structure) (undefined unless successful)

- This call requests the synchronized value of the given 'objective'.

- If the 'peer' parameter is null, and the objective is already available in the local cache, the flooded objective is returned immediately in the 'result' parameter. In this case, the 'timeout' is ignored.

- If the 'peer' parameter is not null, or a cached value is not available, synchronization with a discovered ASA is performed. If successful, the retrieved objective is returned in the 'result' value.

- The 'peer' parameter is an 'asa_locator' as returned by discover(). If 'peer' is null, GRASP discovery is automatically performed first to find a suitable peer (i.e., any node that supports the objective in question).

- The 'timeout' parameter is described in Section 2.3.2.3.

- This call should be repeated whenever the latest value is needed.

- Asynchronous Mechanisms:

Threaded implementation: Call in a separate thread if asynchronous operation is required.

Event loop implementation: An additional in/out 'session_handle' parameter is used, as in request_negotiate(). If the 'errorcode' parameter has the value 2 ('noReply'), no response has been received so far. The 'session_handle' parameter must be presented in subsequent calls.

- In the case of failure, an exponential backoff is recommended before retrying (Section 3).

* listen_synchronize()

This function is used by an ASA to start acting as a synchronization responder (listener) for a given GRASP objective.

- Input parameters:

  asa_handle (unsigned integer)

  objective (structure)

- Return value:

  errorcode (unsigned integer)

- This instructs GRASP to listen for synchronization requests for the given objective and to respond with the value given in the 'objective' parameter. It also enables discovery responses for the objective, as mentioned under register_objective() in Section 2.3.3.

- This call is non-blocking and may be repeated whenever the value changes.

* stop_listen_synchronize()

This function is used by an ASA to stop acting as a synchronization responder (listener) for a given GRASP objective.

- Input parameters:

  asa_handle (unsigned integer)

  objective (structure)

- Return value:

  errorcode (unsigned integer)

- This call instructs GRASP to stop listening for synchronization requests for the given 'objective', i.e., it cancels a previous listen_synchronize.

* flood()

This function is used by an ASA to flood one or more GRASP objectives throughout the Autonomic Network.

Note that each GRASP node caches all flooded objectives that it receives, until each one's time to live expires. Cached objectives are tagged with their origin as well as an expiry time, so multiple copies of the same objective may be cached simultaneously. Further details are given in "Flood Synchronization Message" (Section 2.8.11 of [RFC8990]).

- Input parameters:

  asa_handle (unsigned integer)

  ttl (unsigned integer)

  tagged_objective_list (structure)

- Return value:

  errorcode (unsigned integer)

- This call instructs GRASP to flood the given synchronization objective(s) and their value(s) and associated locator(s) to all GRASP nodes.

- The 'ttl' parameter is the valid lifetime (time to live) of the flooded data in milliseconds (0 = infinity).

- The 'tagged_objective_list' parameter is a list of one or more 'tagged_objective' couplets. The 'locator' parameter that tags each objective is normally null but may be a valid 'asa_locator'. Infrastructure ASAs needing to flood an {address, protocol, port} 3-tuple with an objective create an asa_locator object to do so. If the IP address in that locator is the unspecified address ('::'), it is replaced by the link-local address of the sending node in each copy of the flood multicast, which will be forced to have a loop count of 1. This feature is for objectives that must be restricted to the local link.

- The function checks that the ASA registered each objective.

- This call may be repeated whenever any value changes.

* get_flood()

  This function is used by any ASA to obtain the current value of a flooded GRASP objective.

- Input parameters:

  asa_handle (unsigned integer)

  objective (structure)

- Return values:

  errorcode (unsigned integer)

  tagged_objective_list (structure) (undefined unless successful)

- This call instructs GRASP to return the given synchronization objective if it has been flooded and its lifetime has not expired.

- The 'tagged_objective_list' parameter is a list of 'tagged_objective' couplets, each one being a copy of the flooded objective and a corresponding locator. Thus, if the same objective has been flooded by multiple ASAs, the recipient can distinguish the copies.

- Note that this call is for advanced ASAs. In a simple case, an ASA can simply call synchronize() in order to get a valid flooded objective.

* expire_flood()

  This function may be used by an ASA to expire specific entries in the local GRASP flood cache.

  - Input parameters:

    asa_handle (unsigned integer)

    tagged_objective (structure)

  - Return value:

    errorcode (unsigned integer)

  - This is a call that can only be used after a preceding call to get_flood() by an ASA that is capable of deciding that the flooded value is stale or invalid. Use with care.

  - The 'tagged_objective' parameter is the one to be expired.

2.3.7. Invalid Message Function

* send_invalid()

  This function may be used by any ASA to stop an ongoing GRASP session.

  - Input parameters:

    asa_handle (unsigned integer)

    session_handle (structure)

    info (bytes)

  - Return value:

    errorcode (unsigned integer)

  - Sends a GRASP Invalid message (M_INVALID), as described in [RFC8990]. It should not be used if end_negotiate() would be sufficient. Note that this message may be used in response to any unicast GRASP message that the receiver cannot interpret correctly. In most cases, this message will be generated internally by a GRASP implementation.

    'info' = optional diagnostic data supplied by the ASA. It may be raw bytes from the invalid message.

3. Security Considerations

Security considerations for the GRASP protocol are discussed in [RFC8990]. These include denial-of-service issues, even though these are considered a low risk in the ACP. In various places, GRASP recommends an exponential backoff. An ASA using the API should use exponential backoff after failed discover(), req_negotiate(), or synchronize() operations. The timescale for such backoffs depends on the semantics of the GRASP objective concerned. Additionally, a flood() operation should not be repeated at shorter intervals than is useful. The appropriate interval depends on the semantics of the GRASP objective concerned. These precautions are intended to assist the detection of denial-of-service attacks.

As a general precaution, all ASAs able to handle multiple negotiation

or synchronization requests in parallel may protect themselves against a denial-of-service attack by limiting the number of requests they handle simultaneously and silently discarding excess requests. It might also be useful for the GRASP core to limit the number of objectives registered by a given ASA, the total number of ASAs registered, and the total number of simultaneous sessions, to protect system resources. During times of high autonomic activity, such as recovery from widespread faults, ASAs may experience many GRASP session failures. Guidance on making ASAs suitably robust is given in [ASA-GUIDE].

As noted earlier, the trust model is that all ASAs in a given Autonomic Network communicate via a secure autonomic control plane; therefore, they trust each other's messages. Specific authorization of ASAs to use particular GRASP objectives is a subject for future study, also briefly discussed in [RFC8990].

The careful reader will observe that a malicious ASA could extend a negotiation session indefinitely by use of the negotiate_wait() function or by manipulating the loop count of an objective. A robustly implemented ASA could detect such behavior by a peer and break off negotiation.

The 'asa_handle' is used in the API as a first line of defense against a malware process attempting to imitate a legitimately registered ASA. The 'session_handle' is used in the API as a first line of defense against a malware process attempting to hijack a GRASP session. Both these handles are likely to be created using GRASP's 32-bit pseudorandom Session ID. By construction, GRASP avoids the risk of Session ID collisions (see "Session Identifier (Session ID)", Section 2.7 of [RFC8990]). There remains a finite probability that an attacker could guess a Session ID, session_handle, or asa_handle. However, this would only be of value to an attacker that had already penetrated the ACP, which would allow many other simpler forms of attack than hijacking GRASP sessions.

4. IANA Considerations

This document has no IANA actions.

5. References

5.1. Normative References

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[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>.

5.2. Informative References

[ANIMA-COORD] Ciavaglia, L. and P. Peloso, "Autonomic Functions Coordination", Work in Progress, Internet-Draft, draft-ciavaglia-anima-coordination-01, 21 March 2016,

<https://tools.ietf.org/html/draft-ciavaglia-anima-coordination-01>.

[ASA-GUIDE] Carpenter, B., Ciavaglia, L., Jiang, S., and P. Peloso, "Guidelines for Autonomic Service Agents", Work in Progress, Internet-Draft, draft-ietf-anima-asa-guidelines-00, 14 November 2020, <https://tools.ietf.org/html/draft-ietf-anima-asa-guidelines-00>.

[GRASP-DISTRIB] Liu, B., Xiao, X., Hecker, A., Jiang, S., Despotovic, Z., and B. Carpenter, "Information Distribution over GRASP", Work in Progress, Internet-Draft, draft-ietf-anima-grasp-distribution-02, 8 March 2021, <https://tools.ietf.org/html/draft-ietf-anima-grasp-distribution-02>.

[libcbor] Kalvoda, P., "libcbor - libcbor 0.8.0 documentation", April 2021, <https://libcbor.readthedocs.io/>.

[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>.

[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>.

Appendix A. Error Codes

This appendix lists the error codes defined so far on the basis of implementation experience, with suggested symbolic names and corresponding descriptive strings in English. It is expected that complete API implementations will provide for localization of these descriptive strings, and that additional error codes will be needed according to implementation details.

The error codes that may only be returned by one or two functions are annotated accordingly, and the others may be returned by numerous functions. The 'noSecurity' error will be returned to most calls if GRASP is running in an insecure mode (i.e., with no secure substrate such as the ACP), except for the specific DULL usage mode described in "Discovery Unsolicited Link-Local (DULL) GRASP" (Section 2.5.2 of [RFC8990].

Name	Error Code	Description
ok	0	"OK"
declined	1	"Declined" (req_negotiate, negotiate_step)
noReply	2	"No reply" (indicates waiting state in event loop calls)
unspec	3	"Unspecified error"

ASAfull	4	"ASA registry full" (register_asa)
dupASA	5	"Duplicate ASA name" (register_asa)
noASA	6	"ASA not registered"
notYourASA	7	"ASA registered but not by you" (deregister_asa)
notBoth	8	"Objective cannot support both negotiation and synchronization" (register_obj)
notDry	9	"Dry-run allowed only with negotiation" (register_obj)
notOverlap	10	"Overlap not supported by this implementation" (register_obj)
objFull	11	"Objective registry full" (register_obj)
objReg	12	"Objective already registered" (register_obj)
notYourObj	13	"Objective not registered by this ASA"
notObj	14	"Objective not found"
notNeg	15	"Objective not negotiable" (req_negotiate, listen_negotiate)
noSecurity	16	"No security"
noDiscReply	17	"No reply to discovery" (req_negotiate)
sockErrNegRq	18	"Socket error sending negotiation request" (req_negotiate)
noSession	19	"No session"
noSocket	20	"No socket"
loopExhausted	21	"Loop count exhausted" (negotiate_step)
sockErrNegStep	22	"Socket error sending negotiation step" (negotiate_step)
noPeer	23	"No negotiation peer" (req_negotiate, negotiate_step)
CBORfail	24	"CBOR decode failure" (req_negotiate, negotiate_step, synchronize)
invalidNeg	25	"Invalid Negotiate message" (req_negotiate, negotiate_step)

invalidEnd	26	"Invalid end message" (req_negotiate, negotiate_step)
noNegReply	27	"No reply to negotiation step" (req_negotiate, negotiate_step)
noValidStep	28	"No valid reply to negotiation step" (req_negotiate, negotiate_step)
sockErrWait	29	"Socket error sending wait message" (negotiate_wait)
sockErrEnd	30	"Socket error sending end message" (end_negotiate, send_invalid)
IDclash	31	"Incoming request Session ID clash" (listen_negotiate)
notSynch	32	"Not a synchronization objective" (synchronize, get_flood)
notFloodDisc	33	"Not flooded and no reply to discovery" (synchronize)
sockErrSynRq	34	"Socket error sending synch request" (synchronize)
noListener	35	"No synch listener" (synchronize)
noSynchReply	36	"No reply to synchronization request" (synchronize)
noValidSynch	37	"No valid reply to synchronization request" (synchronize)
invalidLoc	38	"Invalid locator" (flood)

Table 1: Error Codes

Acknowledgements

Excellent suggestions were made by Ignas Bagdonas, Carsten Bormann, Laurent Ciavaglia, Roman Danyliw, Toerless Eckert, Benjamin Kaduk, Erik Kline, Murray Kucherawy, Paul Kyzivat, Guangpeng Li, Michael Richardson, Joseph Salowey, ric Vyncke, Magnus Westerlund, Rob Wilton, and other participants in the ANIMA WG and the IESG.

Authors' Addresses

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

Email: brian.e.carpenter@gmail.com

Bing Liu (editor)
Huawei Technologies
Q14, Huawei Campus
No.156 Beiqing Road
Hai-Dian District, Beijing
100095
China

Email: leo.liubing@huawei.com

Wendong Wang
BUPT University
Beijing University of Posts & Telecom.
No.10 Xitucheng Road
Hai-Dian District, Beijing 100876
China

Email: wdwang@bupt.edu.cn

Xiangyang Gong
BUPT University
Beijing University of Posts & Telecom.
No.10 Xitucheng Road
Hai-Dian District, Beijing 100876
China

Email: xygong@bupt.edu.cn