Workgroup: Network Working Group
Internet-Draft: draft-ietf-anima-grasp-api-07
Published: 13 October 2020
Intended Status: Informational
Expires: 16 April 2021
Authors: B. E. Carpenter, Univ. of Auckland
B. Liu, Ed., Huawei Technologies
W. Wang, BUPT University
X. Gong, BUPT University

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 (ASA) 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 to the support for asynchronicity in various programming languages and operating systems.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

As defined in [I-D.ietf-anima-reference-model], the Autonomic Service Agent (ASA) is the atomic entity of an autonomic function, and it is instantiated on autonomic nodes. When ASAs communicate with each other, they should use the Generic Autonomic Signaling Protocol (GRASP) [I-D.ietf-anima-grasp].

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 GRASP basic protocol. For example, [I-D.ietf-anima-grasp-distribution] describes a possible extended function.

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 can 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 is a daemon, or a set of sub-services, providing 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 inter-process communication (IPC) mechanism. The details of this are system-dependent.

```
    +--------------+          +--------------+
    |     ASAs     |          |     ASAs     |
    +--------------+          +--------------+
      |          |                    |
      | +-------------------+         |
      | | GRASP Extended    |         |
      | | Function API      |         |
      | +-------------------+         |
      |          |                    |
   +------------------------------------------+
   |            GRASP API Library             |
   +------------------------------------------+
                       |
                      IPC
                       |
   +------------------------------------------+
   |  GRASP Core                              |
   |  (functions, data structures, daemon(s)) |
   +------------------------------------------+
```

Figure 1: Software layout

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.

Note that a simple autonomic node might contain very few ASAs in addition to the autonomic infrastructure components described in [I-D.ietf-anima-bootstrapping-keyinfra] and [I-D.ietf-anima-autonomic-control-plane]. 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 then need a deeper understanding of the GRASP protocol than 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 assumption of this document is that any Autonomic Service Agent (ASA) needs to call a GRASP module. The latter handles protocol details (security, sending and listening for GRASP messages, waiting, caching discovery results, negotiation looping, sending and receiving sychronization data, etc.) but understands nothing about individual GRASP objectives (Section 2.10 of [I-D.ietf-anima-grasp]). The semantics of objectives are unknown to the GRASP module 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 make different use of GRASP features:

* Use GRASP only for discovery purposes.

* Use GRASP negotiation but only as an initiator (client).

* Use GRASP negotiation but only as a responder.

* Use GRASP negotiation as an initiator or responder.

* Use GRASP synchronization but only as an initiator (recipient).

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

* Use GRASP 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 not supported.

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

* The Rapid mode of GRASP (Section 2.5.4 of [I-D.ietf-anima-grasp]) is not supported.

2.2. Asynchronous Operations

GRASP depends on asynchronous operations and wait states, and 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 which 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

Thus, some ASAs need to support asynchronous operations, and 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: multi-threading, an event loop structure using polling, and an event loop structure using callback functions.

1. In multi-threading, 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. In an event loop implementation with callbacks, the ASA programmer would provide a callback function for each asynchronous operation, e.g. discovery_received(). This would be called asynchronously when a reply is received or a failure such as a timeout occurs.

The following calls involve waiting for a remote operation, so they could use a polling or callback mechanism. In a threaded mechanism, they will usually require to be called in a separate thread:

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().

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 negotatition 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 it is an assumption of GRASP 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 [I-D.ciavaglia-anima-coordination]. 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 negotiate_step_received(), 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 identifier. Thus, when necessary, a 'session_nonce' parameter 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

On the first call in a new GRASP session, the API returns a 'session_nonce' value based on the GRASP session identifier. This value must be used in all subsequent calls for the same session, and will be provided as a parameter in the callback functions. By this mechanism, multiple overlapping sessions can be distinguished, both in the ASA and in the GRASP core. The value of the 'session_nonce" is opaque to the ASA.

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_nonce 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_nonce parameter exposes this aspect of the implementation.

2.3. API definition

Some example logic flows for a resource management ASA are given in [I-D.carpenter-anima-asa-guidelines], 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 behaviour.

2.3.1. Parameters and data structures

2.3.1.1. Errorcode

All functions in the API have an unsigned 'errorcode' integer as their return value (the first returned value in languages that allow multiple returned parameters). 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 code applies.

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.

2.3.1.2. Timeout

Wherever a 'timeout' parameter appears, it is an integer expressed in milliseconds. If it is zero, the GRASP default timeout (GRASP_DEF_TIMEOUT, see [I-D.ietf-anima-grasp]) will apply. If no response is received before the timeout expires, the call will fail unless otherwise noted.

2.3.1.3. 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 a language such as C the preferred implementation may be to represent the Boolean flags as bits in a single byte.

* loop_count (integer) - Limit on negotiation steps etc. (default GRASP_DEF_LOOPCT, see [I-D.ietf-anima-grasp])

* 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.

  An essential requirement for all language mappings and all 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 [RFC7049] for transmission via GRASP, and unwrap it after reception.

  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, assuming the use of a particular CBOR library, is:

```
typedef struct {
   char *name;
   uint8_t flags;            // flag bits as defined by GRASP
   int loop_count;
   int 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 is:

```
class objective:
   """A GRASP objective"""
   def __init__(self, name):
       self.name = name    # Unique name (string)
       self.negotiate = False  #True if objective supports negotiation
       self.dryrun = False     #True if objective supports dry-run neg.
       self.synch = False  # True if objective supports synch
       self.loop_count = GRASP_DEF_LOOPCT  # Default starting value
       self.value = 0      # Place holder; any valid Python object
```

2.3.1.4. 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 (integer) - The interface identifier index via which this was discovered - probably no use to a normal ASA

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

* The following cover 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 an FQDN

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

* diverted (Boolean) - True if the locator was discovered via a Divert option

* protocol (integer) - Applicable transport protocol (IPPROTO_TCP or IPPROTO_UDP)

*port (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.

2.3.1.5. 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.

2.3.1.6. Asa_nonce

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 nonce that, although not cryptographically protected, would be difficult for a third party to predict. The ASA must present this nonce 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_nonce' parameter is required. It is generated when an ASA first registers with GRASP, and the ASA must then store the asa_nonce and use it in every subsequent GRASP call. Any call in which an invalid nonce is presented will fail. It is an up to 32-bit opaque value (for example represented as a uint32_t, depending on the language). It should be unpredictable; a possible implementation is to use the same mechanism that GRASP uses to generate Session IDs [I-D.ietf-anima-grasp]. Another possible implementation is to hash the name of the ASA with a locally defined secret key.

2.3.1.7. Session_nonce

In some calls, a 'session_nonce' 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). In fully threaded implementations this parameter might not be needed, but it is included to act as a session handle if necessary. It will also allow GRASP to detect and ignore malicious calls or calls from timed-out sessions. A possible implementation is to form the nonce from the underlying GRASP Session ID and the source address of the session.

2.3.2. Registration

These functions are used to register an ASA and the objectives that it supports with the GRASP module. If an authorization model is added to GRASP, these API calls would need to be modified accordingly.

*register_asa()

-Input parameter:

name of the ASA (UTF-8 string)

-Return parameters:

errorcode (integer)

asa_nonce (integer) (if successful)

-This initialises state in the GRASP module for the calling entity (the ASA). In the case of success, an 'asa_nonce' 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.

*deregister_asa()

-Input parameters:

asa_nonce (integer)

name of the ASA (UTF-8 string)

-Return parameter:

errorcode (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 crashes.

-Note - the ASA name is strictly speaking redundant in this call, but is present for clarity.

```
*register_objective()

   -Input parameters:

       asa_nonce (integer)

       objective (structure)

       ttl (integer - default GRASP_DEF_TIMEOUT)

       discoverable (Boolean - default False)

       overlap (Boolean - default False)

       local (Boolean - default False)

   -Return parameter:

       errorcode (integer)

   -This registers an objective that this ASA supports and may
    modify. 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 for this objective. The
    default value should be the GRASP default timeout
    (GRASP_DEF_TIMEOUT, see [I-D.ietf-anima-grasp]).

   -If the parameter 'discoverable' is True, the objective is
    immediately discoverable. This is intended for objectives that
    are only defined for GRASP discovery, and which 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.

   -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_nonce (integer)

objective (structure)

-Return parameter:

errorcode (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.

2.3.3. Discovery

*discover()

-Input parameters:

asa_nonce (integer)

objective (structure)

timeout (integer)

age_limit (integer)

-Return parameters:

errorcode (integer)

locator_list (structure)

-This returns a list of discovered 'ASA_locator's for the given objective. Note that this structure includes all the fields described in Section 2.3.1.4.

-If the parameter 'age_limit' is greater than zero, any locally cached locators for the objective whose remaining lifetime in milliseconds is less than or equal to 'age_limit' are deleted first. Thus 'age_limit' = 0 will flush all entries.

-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 'age_limit' 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.

-Asynchronous Mechanisms:

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

oEvent loop implementation: An additional read/write 'session_nonce' parameter is used. A callback may be used in the case of a non-zero tiemout.

2.3.4. Negotiation

*request_negotiate()

-Input parameters:

asa_nonce (integer)

objective (structure)

peer (ASA_locator)

timeout (integer)

-Return parameters:

errorcode (integer)

session_nonce (structure) (if successful)

proffered_objective (structure) (if successful)

reason (string) (if negotiation declined)

-This function opens a negotiation session between two ASAs. Note that GRASP currently does not support multi-party 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).

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

  1. The 'session_nonce' parameter is null. In this case the negotiation has succeeded immediately (the peer has accepted the request). The returned 'proffered_objective' contains the value accepted by the peer.

  2. The 'session_nonce' parameter is not null. In this case negotiation must continue. The 'session_nonce' must be presented in all subsequent negotiation steps. The returned 'proffered_objective' contains the first value proffered by the negotiation peer. The contents of this instance of the objective must be used in the subsequent negotiation call 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.

     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.

- Asynchronous Mechanisms:

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

  - Event loop implementation: The 'session_nonce' parameter is used to distinguish multiple simultaneous sessions.

- Use of dry run mode: This 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_nonce' data structure, which is opaque to normal ASAs.

* listen_negotiate()

  - Input parameters:

    asa_nonce (integer)

    objective (structure)

  - Return parameters:

    errorcode (integer)

    session_nonce (structure) (if successful)

    requested_objective (structure) (if 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 2.3.2.

  - Asynchronous Mechanisms:

    o Threaded implementation: It will block waiting for an incoming request, so 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 thread must be spawned for each new session.

    o Event loop implementation: A 'session_nonce' 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.

- 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_nonce' 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. 'listen_negotiate' may then be called again to await a new negotiation.

- 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 nonces, and the underlying unicast transport sockets.

*stop_listen_negotiate()

- Input parameters:

  asa_nonce (integer)

  objective (structure)

- Return parameter:

  errorcode (integer)

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

- Asynchronous Mechanisms:

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

  o Event loop implementation: no special considerations.

*negotiate_step()

-Input parameters:

asa_nonce (integer)

session_nonce (structure)

objective (structure)

timeout (integer)

-Return parameters:

Exactly as for 'request_negotiate'

-Executes the next negotation step with the peer. The 'objective' parameter contains the next value being proffered by the ASA in this step.

-Asynchronous Mechanisms:

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

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

*negotiate_wait()

-Input parameters:

asa_nonce (integer)

session_nonce (structure)

timeout (integer)

-Return parameters:

errorcode (integer)

-Delay negotiation session by 'timeout' milliseconds, thereby extending the original timeout. This function simply triggers a GRASP Confirm Waiting message (see [I-D.ietf-anima-grasp] for details).

-Asynchronous Mechanisms:

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

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

*end_negotiate()

-Input parameters:

asa_nonce (integer)

session_nonce (structure)

result (Boolean)

reason (UTF-8 string)

-Return parameters:

errorcode (integer)

-End the negotiation session.

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

'reason' = optional string describing reason for decline.

-Asynchronous Mechanisms:

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

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

2.3.5. Synchronization and Flooding

*synchronize()

-Input parameters:

asa_nonce (integer)

objective (structure)

peer (ASA_locator)

timeout (integer)

-Return parameters:

errorcode (integer)

objective (structure) (if successful)

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

-Since this is essentially a read operation, any ASA can do it, unless an authorization model is added to GRASP in future. Therefore the API checks that the ASA is registered, but the objective does not need to be registered by the calling ASA.

-If the objective was already flooded, the flooded value is returned immediately in the 'result' parameter. In this case, the 'peer' and 'timeout' are ignored.

-Otherwise, synchronization with a discovered ASA is performed. 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).

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

-Asynchronous Mechanisms:

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

oEvent loop implementation: An additional read/write 'session_nonce' parameter is used.

-Since this is essentially a read operation, any ASA can use it. Therefore GRASP checks that the calling ASA is registered but the objective doesn't need to be registered by the calling ASA.

-In the case of failure, an exponential backoff is recommended before retrying.

*listen_synchronize()

-Input parameters:

asa_nonce (integer)

objective (structure)

-Return parameters:

errorcode (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 2.3.2.

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

*stop_listen_synchronize()

-Input parameters:

asa_nonce (integer)

objective (structure)

-Return parameters:

errorcode (integer)

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

*flood()

-Input parameters:

asa_nonce (integer)

ttl (integer)

tagged_objective_list (structure)

-Return parameters:

errorcode (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()

-Input parameters:

asa_nonce (integer)

objective (structure)

-Return parameters:

errorcode (integer)

tagged_objective_list (structure) (if successful)

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

-Since this is essentially a read operation, any ASA can do it. Therefore the API checks that the ASA is registered but the objective doesn't need to be registered by the calling ASA.

-The 'tagged_objective_list' parameter is a list of 'tagged_objective' couplets, each one being a copy of the flooded objective and a coresponding 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()

-Input parameters:

asa_nonce (integer)

tagged_objective (structure)

-Return parameters:

errorcode (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.

2.3.6. Invalid Message Function

*send_invalid()

-Input parameters:

asa_nonce (integer)

session_nonce (structure)

info (bytes)

-Return parameters:

errorcode (integer)

-Sends a GRASP Invalid Message (M_INVALID) message, as described in [I-D.ietf-anima-grasp]. 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. May be raw bytes from the invalid message.

3. Implementation Status [RFC Editor: please remove]

A prototype open source Python implementation of GRASP, including an API similar to this document, has been used to verify the concepts for the threaded model. It may be found at https://github.com/becarpenter/graspy with associated documentation and demonstration ASAs.

4. Security Considerations

Security issues for the GRASP protocol are discussed in [I-D.ietf-anima-grasp]. Authorization of ASAs is a subject for future study.

The 'asa_nonce' parameter is used in the API as a first line of defence against a malware process attempting to imitate a legitimately registered ASA. The 'session_nonce' parameter is used in the API as a first line of defence against a malware process attempting to hijack a GRASP session.

5. IANA Considerations

This document makes no request of the IANA.

6. Acknowledgements

Excellent suggestions were made by Ignas Bagdonas, Laurent Ciavaglia, Toerless Eckert, Guangpeng Li, Michael Richardson, Rob Wilton, and other participants in the ANIMA WG.

7. References

7.1. Normative References

[I-D.ietf-anima-grasp] Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", Work in Progress, Internet-Draft, draft-ietf-anima-grasp-15, 13 July 2017, <https://tools.ietf.org/html/draft-ietf-anima-grasp-15>.

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

7.2. Informative References

[I-D.carpenter-anima-asa-guidelines]

Carpenter, B., Ciavaglia, L., Jiang, S., and P. Pierre, "Guidelines for Autonomic Service Agents", Work in Progress, Internet-Draft, draft-carpenter-anima-asa-guidelines-09, 25 July 2020, <https://tools.ietf.org/html/draft-carpenter-anima-asa-guidelines-09>.

[I-D.ciavaglia-anima-coordination]
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>.

[I-D.ietf-anima-autonomic-control-plane]
Eckert, T., Behringer, M., and S. Bjarnason, "An Autonomic Control Plane (ACP)", Work in Progress, Internet-Draft, draft-ietf-anima-autonomic-control-plane-29, 11 September 2020, <https://tools.ietf.org/html/draft-ietf-anima-autonomic-control-plane-29>.

[I-D.ietf-anima-bootstrapping-keyinfra]
Pritikin, M., Richardson, M., Eckert, T., Behringer, M., and K. Watsen, "Bootstrapping Remote Secure Key Infrastructures (BRSKI)", Work in Progress, Internet-Draft, draft-ietf-anima-bootstrapping-keyinfra-44, 21 September 2020, <https://tools.ietf.org/html/draft-ietf-anima-bootstrapping-keyinfra-44>.

[I-D.ietf-anima-grasp-distribution]
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-01, 1 September 2020, <https://tools.ietf.org/html/draft-ietf-anima-grasp-distribution-01>.

[I-D.ietf-anima-reference-model]
Behringer, M., Carpenter, B., Eckert, T., Ciavaglia, L., and J. Nobre, "A Reference Model for Autonomic Networking", Work in Progress, Internet-Draft, draft-ietf-anima-reference-model-10, 22 November 2018, <https://tools.ietf.org/html/draft-ietf-anima-reference-model-10>.

Appendix A. Error Codes

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

```
ok                0 "OK"
declined          1 "Declined"
noReply           2 "No reply"
unspec            3 "Unspecified error"
ASAfull           4 "ASA registry full"
dupASA            5 "Duplicate ASA name"
noASA             6 "ASA not registered"
notYourASA        7 "ASA registered but not by you"
notBoth           8 "Objective cannot support both negotiation
                     and synchronization"
notDry            9 "Dry-run allowed only with negotiation"
notOverlap       10 "Overlap not supported by this implementation"
objFull          11 "Objective registry full"
objReg           12 "Objective already registered"
notYourObj       13 "Objective not registered by this ASA"
notObj           14 "Objective not found"
notNeg           15 "Objective not negotiable"
noSecurity       16 "No security"
noDiscReply      17 "No reply to discovery"
sockErrNegRq     18 "Socket error sending negotiation request"
noSession        19 "No session"
noSocket         20 "No socket"
loopExhausted    21 "Loop count exhausted"
sockErrNegStep   22 "Socket error sending negotiation step"
noPeer           23 "No negotiation peer"
CBORfail         24 "CBOR decode failure"
invalidNeg       25 "Invalid Negotiate message"
invalidEnd       26 "Invalid end message"
noNegReply       27 "No reply to negotiation step"
noValidStep      28 "No valid reply to negotiation step"
sockErrWait      29 "Socket error sending wait message"
sockErrEnd       30 "Socket error sending end message"
IDclash          31 "Incoming request Session ID clash"
notSynch         32 "Not a synchronization objective"
notFloodDisc     33 "Not flooded and no reply to discovery"
sockErrSynRq     34 "Socket error sending synch request"
noListener       35 "No synch listener"
noSynchReply     36 "No reply to synchronization request"
noValidSynch     37 "No valid reply to synchronization request"
invalidLoc       38 "Invalid locator"
```

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

draft-ietf-anima-grasp-api-07, 2020-10-13:

*Improved diagram and its description

*Added pointer to example logic flows

*Added note on variable length parameters

*Clarified that API decrements loop count automatically

*Other corrections and clarifications from AD review

draft-ietf-anima-grasp-api-06, 2020-06-07:

*Improved diagram

*Numerous clarifications and layout changes

draft-ietf-anima-grasp-api-05, 2020-05-08:

*Converted to xml2rfc v3

*Editorial fixes.

draft-ietf-anima-grasp-api-04, 2019-10-07:

*Improved discussion of layering, mentioned daemon.

*Added callbacks and improved description of asynchronous operations.

*Described use case for 'session_nonce'.

*More explanation of 'asa_nonce'.

*Change 'discover' to use 'age_limit' instead of 'flush'.

*Clarified use of 'dry run'.

*Editorial improvements.

draft-ietf-anima-grasp-api-03, 2019-01-21:

*Replaced empty "logic flows" section by "implementation status".

*Minor clarifications.

*Editorial improvements.

draft-ietf-anima-grasp-api-02, 2018-06-30:

*Additional suggestion for event-loop API.

*Discussion of error code values.

draft-ietf-anima-grasp-api-01, 2018-03-03:

*Editorial updates

draft-ietf-anima-grasp-api-00, 2017-12-23:

*WG adoption

*Editorial improvements.

draft-liu-anima-grasp-api-06, 2017-11-24:

*Improved description of event-loop model.

*Changed intended status to Informational.

*Editorial improvements.

draft-liu-anima-grasp-api-05, 2017-10-02:

*Added send_invalid()

draft-liu-anima-grasp-api-04, 2017-06-30:

*Noted that simple nodes might not include the API.

*Minor clarifications.

draft-liu-anima-grasp-api-03, 2017-02-13:

*Changed error return to integers.

*Required all implementations to accept objective values in CBOR.

*Added non-blocking alternatives.

draft-liu-anima-grasp-api-02, 2016-12-17:

*Updated for draft-ietf-anima-grasp-09

draft-liu-anima-grasp-api-02, 2016-09-30:

*Added items for draft-ietf-anima-grasp-07

*Editorial corrections

draft-liu-anima-grasp-api-01, 2016-06-24:

*Updated for draft-ietf-anima-grasp-05

*Editorial corrections

draft-liu-anima-grasp-api-00, 2016-04-04:

*Initial version

Authors' Addresses

Brian 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
P.R. China

Email: leo.liubing@huawei.com

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

Email: wdwang@bupt.edu.cn

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

Email: xygong@bupt.edu.cn