Draft: Modeling Autonomic Network Infrastructure Functions using GRASP

Brian E. Carpenter
brian.e.carpenter@gmail.com
6 April 2016

Boilerplate

This is a contribution to the IETF. It isn't in I-D format because it really needs fonts to be easily readable. If it needs to become a monospaced I-D later, so be it. In any case, as an IETF contribution, it falls under IETF rules.

Table of Contents

Introduction

Several functions in the Autonomic Networking Infrastructure (ANI) are candidates to use GRASP, because GRASP is expected to be supported in every Autonomic Node (AN). This would be achieved by implementing the functions as Autonomic Service Agents (ASAs) because that is the way GRASP works – its API is called by ASAs. There is no magic in this; an ASA is just a module (probably multi-threaded) with a given job to do. These ASAs would be considered part of the ANI.

This document is not a specification or design. It's intended to model how these infrastructure ASAs could use GRASP, as an existence proof that such a solution is possible. It is hand-waving and comments are very much wanted.

Formally, we should define GRASP objectives in CDDL, to capture them in a format that is guaranteed to be translatable to and from CBOR at run time. However, we expect that ASAs will usually be written in a language such as C++ or Python that allows them to handle composite objects in a convenient way. Here, I will use Python examples, based on the existing Python prototype implementation of GRASP (https://www.cs.auckland.ac.nz/~brian/graspy/).

Warning: the code fragments haven't been syntax-checked or tested.

API

This is a summary; the prototype Python API is documented in full at https://www.cs.auckland.ac.nz/~brian/graspy/graspy.pdf.

An infrastructure ASA will need to understand a couple of data structures in order to communicate with GRASP:

`objective()`

This holds a GRASP objective. Its attributes are:

`.name`	Unicode string – the objective's name
`.neg`	Boolean – True if objective supports negotiation (default False)
`.synch`	Boolean – True if objective supports synchronization (default False)
`.loop_count`	integer – Limit on negotiation steps etc. (default GRASP_DEF_LOOPCT)
`.value`	any valid CBOR* object – i.e. the value of the objective (default 0)

* that is, on the wire it will be CBOR. In the API the representation will depend on the programming language.

`asa_locator()`

Objects of this type are returned by a GRASP discovery action. Its attributes are:

`.locator`	The actual locator, either an `ipaddress` or a string
`.ifi`	The interface identifier via which this was discovered – probably no use to a normal ASA
`.diverted`	Boolean – `True` if the locator was discovered via a Divert option
`.is_ipaddress`	Boolean – if `True`, the locator is an `ipaddress` (Python object)
`.is_fqdn`	Boolean – if `True`, the locator is an FQDN (string)
`.is_uri`	Boolean – if `True`, the locator is a URI (string)

Here is a summary of the API functions:

`register_asa, deregister_asa`

Each ASA must use these to register itself when it starts and to sign off when it exits. Registration returns a nonce that must be presented in all subsequent calls to the API.

`register_obj, deregister_obj`

Each ASA must register each GRASP objective that it supports either for negotiation or as a data source for synchronization or flooding. Deregistration of objectives is possible (but is done automatically by `deregister_asa)`. It isn't necessary to register for an objective that is receive-only for the ASA.

`discover`

This is used to discover peers for negotiation or synchronization. It returns results of type `asa_locator.` They can be used in subsequent calls, but if they are omitted GRASP will perform discovery implicitly.

`req_negotiate`

This is used by a negotiation initiator to start a negotiation sequence (with a GRASP Request message).

`listen_negotiate, stop_negotiate`

An ASA that wishes to respond to negotiation requests calls `listen_negotiate` to start listening and `stop_negotiate` to stop listening.

`negotiate_step, negotiate_wait, end_negotiate`

These are used by negotiation initiators and responders to conduct a negotiation sequence, following `req_negotiate` or `listen_negotiate`.

`synchronize`

This is used by a synchronization initiator to start synchronization (with a GRASP Request message). That is a receive-only operation which will collect a flooded value if present, but otherwise perform unicast synchronization.

`listen_synchronize, stop_synchronize`

An ASA that wishes to respond to synchronization requests calls `listen_ synchronize` to start listening and `stop_ synchronize` to stop listening.

`flood`

This is used by an ASA that wishes to flood one or more GRASP objectives to the AN.

Secure Bootstrap

Three infrastructure ASAs could support secure bootstrap. We'll call them "AN_Registrar", "AN_BootProxy" and "AN_Pledge". We'll assume for now that the code for all three is installed on every autonomic node. There are two cases to consider:

a) the "normal" case where the Autonomic Control Plane is already running (see next section). This means that all operational nodes will have ACP addresses and can talk to each other securely. The pledge (the joining node) doesn't yet have an ACP address because it doesn't have the necessary credentials. As far as GRASP is concerned, it is running in insecure mode. Sensitive packets must not travel off-link (achieved by setting the GRASP loop count to 1 and/or using GTSM [RFC5082]) except via ACP.

b) the "cold start" case in which the ACP is not yet operational, so neither the proxy nor the registrar have ACP addresses. We can force the pledge to wait in that situation, simply by ensuring that no proxy can be discovered. But even so, each potential proxy needs to communicate with the registrar, although there is no pre-defined secure channel. When appropriate, we can ensure that sensitive packets don't travel off-link as above, but it seems likely that communication with the registrar may need to travel off link anyway.

This will probably be resolved because the secure network can grow outwards from the registrar – first its on-link neighbours can be bootsrapped, then they can start forming the ACP among themselves, then the next level neighbors can bootstrap and the ACP can expand, and so on throughout the domain.

An important TBD outside the scope of this note is how a given node is authorized to be the registrar. We'll assume that exactly one node has that role.

First consider the Registrar. It will support an objective also called "AN_Registrar", so that it can be discovered by the proxies. If the proxies need information from the registrar beyond its address, this could be supplied as the value of the "AN_Registrar" objective. The Registrar's startup logic would be something like:

```
ok, asa_nonce = grasp.register_asa("AN_Registrar")
if not ok:
    # Error, could not register
my_obj = grasp.objective("AN_Registrar")
my_obj.loop_count = 10
my_obj.synch = True
my_obj.value = "some information for proxies"
ok, temp = grasp.register_obj(asa_nonce, my_obj)
 if not ok:
    # Error, could not register
ok, result = grasp.listen_synchronize(asa_nonce,
      my_obj)
if not ok:
    # We have a very basic problem...
```

From this point on, the Registrar is available for GRASP discovery (even in insecure mode) and synchronization. Note that we set the loop count to 10, which limits the diameter if we decided to flood the objective as well. But flooding is insecure.

A potential proxy, once it has determined that it's in a node capable of acting as a proxy (because it has multiple physical interfaces) will do something like:

```
ok, asa_nonce = grasp.register_asa("AN_BootProxy")
if not ok:
    # Error, could not register
reg_obj = grasp.objective("AN_Registrar")
reg_obj.loop_count = 10
reg_obj.synch = True
while True: # will keep trying for ever
    locators = grasp.discover(asa_nonce, reg_obj, 5000)
    if locators==[]:
        # Discovery failed, wait before retry
        time.sleep(5)
        continue
    # We have at least one locator for the Registrar.
    # Pick a locator from the set of discovered locators,
    # for example:
    registrar_loc = locators[0]
    # Do any validity checks that we need
    break
# We found the Registrar, now get its info
ok, result = grasp.synchronize(asa_nonce, reg_obj,
                  registrar_loc, 5000)
if not ok:
    # Error...
registrar_info = result.value
# Now we can be available for pledges
proxy_obj = grasp.objective("AN_BootProxy")
proxy_obj.loop_count = 1 # on-link only
proxy_obj.synch = True
proxy_obj.value = "some info for each pledge"
ok, temp = grasp.register_obj(asa_nonce, proxy_obj)
 if not ok:
    # Error, could not register
ok, result = grasp.listen_negotiate(asa_nonce,
      proxy_obj)
if not ok:
    # We have a very basic problem...
```

Alternatively it could flood the information to its neighbors:

```
ok, result = grasp.flood(asa_nonce, proxy_obj)
```

Now any pledge can discover us and get the info. For the dialogue between new pledges and the proxy, and between the proxy and the registrar, see draft-ietf-anima-bootstrapping-keyinfra.

The pledge's logic looks something like this:

```
ok, asa_nonce = grasp.register_asa("AN_Pledge")
if not ok:
    # Error, could not register
seek_obj = grasp.objective("AN_BootProxy")
seek_obj.loop_count = 1
seek_obj.synch = True
while True: # will keep trying for ever
    locators = grasp.discover(asa_nonce, seek_obj, 5000)
    if locators==[]:
        # Discovery failed, wait before retry
        time.sleep(5)
        continue
    # We have at least one locator for the Proxy.
    # Pick a locator from the set of discovered locators,
    # for example:
    proxy_loc = locators[0]
    # Do any validity checks that we need
    break
# We found a proxy, now get its info
ok, result = grasp.synchronize(asa_nonce, seek_obj,
                  proxy_loc, 5000)
if not ok:
    # Error...
proxy_info = result.value
```

Now the pledge has the proxy's locator and its info, and can start the dialogue with the proxy.

ACP formation

The ACP has to be formed by a bunch of nodes that initially have no knowledge of each other or of the network topology (which could of course include loops and point-to-point links). We also may not yet have much in the way of secure identities, but as noted above the secure domain will grow outwards during a cold start. We assume only that each node has a link-local address on each interface, and that each node contains an active copy of an infrastructure ASA called "AN_ACP" that supports an objective of the same name.

When a node starts up it needs to find its AN neighbours and allow them to find it. Therefore, "AN_ACP" has to behave symmetrically, both offering and discovering the "AN_ACP" objective. We'll assume that means (at least) two separate threads, which meet at a shared data structure (the adjacency table). Here I will miss out the syntactic sugar for threading and for locks to make the adjacency table thread safe.

So the main thread first does something like:

```
ok, asa_nonce = grasp.register_asa("AN_ACP")
if not ok:
    # Error, could not register
```

Theh it can start sub-threads which act as a single ASA by sharing the nonce. Since the ACP nodes need to negotiate their secure connections, we need a negotiation objective. The offering thread does something like:

```
my_obj = grasp.objective("AN_ACP")
my_obj.loop_count = 1 # Keep it link-local
my_obj.neg = True
ok, temp = grasp.register_obj(asa_nonce, my_obj)
 if not ok:
    # Error, could not register
ok, new_nonce, result = grasp.listen_negotiate(asa_nonce,
                my_obj)
if not ok:
    # We have a very basic problem...

# We got an incoming request.
neighbor_request = result.value
# Continue negotiation loop using
# negotiate_step, negotiate_wait, end_negotiate
# as necessary.
```

The discovering thread would do the converse:

```
seek_obj = grasp.objective("AN_ACP")
seek_obj.loop_count = 1 # Keep it link-local
seek_obj.neg = True
while True: # Keep trying for ever
    locators = grasp.discover(asa_nonce, seek_obj, 5000)
    if locators==[]:
        # Discovery failed, wait before retry
        time.sleep(5)
        continue
    # We have at least one locator for neighbors

    seek_obj.value = whatever #initial request to neighbor
    for neighbor_loc in ll:
        if not neighbor_loc in adjacency_table:
            ok, new_nonce, result = grasp.req_negotiate(asa_nonce,
                           seek_obj, neighbor_loc, 5000)
            if not ok:
                # Error...
            neighbor_offer = result.value
            # Put new neighbor_loc into adjacency table.
            # Continue negotiation loop using
            # negotiate_step, negotiate_wait, end_negotiate
            # as necessary.
```

Intent distribution

Intent is not yet well defined in Anima, so this section is very provisional.

We'll assume that although Intent is conceptually a single construct for the whole AN, it might come in subsets from more than one source. We'll further assume that it is structured according to the GRASP objectives that it applies too, plus an abstract "general" part the applies everywhere. So we can assume that there is a single objective called "Intent" whose value is the Intent for the whole AN (format, syntax and semantics TBD, but for GRASP it's just another objective). And we'll assume that there are subsets of this "Intent" named "Intent.General" (for the general intent) and "Intent.X" for each objective "X".

Then Intent is very simply modeled in GRASP. An Intent server is an ASA that supports synchronization or flooding of the objectives "Intent" or "Intent.General" or any number of "Intent.X". An Intent client is any ASA that requests synchronization of "Intent" or its subsets (whether flooded or unicast). "Intent.General" needs to be understood by every ASA. The rest can be specific.

Putting it another way, this is an absolutely straightforward application of GRASP synchronization. The only loose end is how Intent servers are authorized. In a simplistic approach, the code for the (assumed unique) Intent server looks like:

```
ok, asa_nonce = grasp.register_asa("AN_IntentServer")
if not ok:
    # Error, could not register
my_obj = grasp.objective("AN_Intent")
my_obj.loop_count = 10 # Assume diameter of AN
my_obj.synch = True
ok, temp = grasp.register_obj(asa_nonce, my_obj)
 if not ok:
    # Error, could not register
while true:
    # Wait until new or updated Intent is available,
    # and then:
    my_obj.value = # The whole of Intent as one object
    # Flood the Intent everywhere
    ok, result = grasp.flood(asa_nonce, my_obj)
    if not ok:
        # We have a very basic problem...
    # Support unicast requests for Intent
    ok, result = grasp.listen_synchronize(asa_nonce,
                       my_obj)
    if not ok:
        # We have a very basic problem...
```

Then every ASA (since Intent is mandatory) will need to do something like this, probably in a separate thread. This version assumes we check Intent every 10 seconds. If the Intent server floods as shown above, this loop will generate no network traffic because the result is already cached. Unicast synchronization will kick in automatically if the flood cache doesn't contain the `AN_Intent` object for some reason, such as the node was off-line when the flood was sent.

```
intent_obj = grasp.objective("AN_Intent")
intent_obj.synch = True
while True:
    ok, result = grasp.synchronize(asa_nonce,
                               intent_obj, None, 5000)
    if not OK:
        time.sleep(10) # Error, wait and retry
        continue
    intent = result.value # Store new intent object
    # Trigger analysis and application of new intent
    time.sleep(10) # Wait and retry
```

Even if Intent is large, synchronization shouldn't be a problem since it runs over TCP. It could be a problem for flooding (multicast). In that we would have to chop Intent into smaller pieces for flooding. TBD.

If instead of assuming one unique Intent for the whole AN, we assume partial Intents, then we'd need ASAs sourcing `AN_Intent.general` and `AN_Intent.X` for each `X`. I'm inclined to agree with Michael Behringer that this is too complicated for now, but we can easily model it in GRASP.