Python prototype code of a GRASP engine and API

Brian E. Carpenter
brian.e.carpenter@gmail.com
22 May 2021

Table of Contents

Introduction

This note describes a prototype open source implementation of the GeneRic Autonomic Signaling Protocol (GRASP) originally written in Python 3.4, tested up to 3.7.2. It is based on Proposed Standard RFC8990. GRASP is intended for use by Autonomic Service Agents (ASAs). For further background, see Informational RFC8993.

The implementation provides two slightly different versions of the application programming interface (API) for use by ASAs. The principal API is provided by module **`graspi.py`** and implements Informational RFC8991. A slightly different older version, retained for backwards compatibility, referred to below as the "old API", is provided by the underlying module **`grasp.py`**.

The code is not guaranteed or validated and is both incomplete and probably wrong. Its main purpose was to help improve the specifications. It can also be used to help test other implementations. Python being an interpreted language, performance is slow.

SECURITY WARNINGS:

- There is no security in the GRASP protocol itself; it relies on an Autonomic Control Plane (ACP) substrate.
- The code assumes that an Autonomic Control Plane (ACP) up on all interfaces or none; does not watch for interface up/down changes.
- By default, the code runs with an elementary ACP that offers no security.
- There is no TLS support.
- Unless layer 2 security is provided, the user is RECOMMENDED to run QUick And Dirty Security (QUADS) for GRASP, which acts as a virtual ACP, protecting all traffic. This is described below (page 21).
- The code optionally supports the DULL security instance describd in the GRASP specification. See below (page 21).

LIMITATIONS:

- Only coded for IPv6, no IPv4 support
- FQDN and URI locators are allowed in discovery responses, but otherwise they are not handled at all.
- No code for rapid mode negotiation.
- The relay code is lazy (no rate control).
- There are work-arounds for issues in the Python socket module and Windows socket peculiarities. The code is intended to be portable between Windows (Winsock2) and POSIX environments but YMMV.

This document covers the APIs for use by ASAs, plus implementation and testing.

The latest version of this document, and the code, can be found at https://github.com/becarpenter/graspy.

Acknowledgements: 神明達哉 (JINMEI Tatuya) for help with POSIX compatibility, Ulrich Speidel for loaning a Linux test box, and at IETF meetings: Michael Richardson, William Atwood, Jéferson Campos Nobre and Bing Liu.

API

The approach is procedural in style, i.e. mainly a collection of functions. This is not very Pythonic, i.e. there is no such thing as a GRASP session defined as a Python class. A session is identified by a session handle[1] that the programmer must present to some API functions. To use the API, Python 3 code needs to **`import graspi`**. This will **`import grasp`**, which will in turn **`import acp`**. All three modules must be in the Python path. Code using the old API uses **`import grasp`**.

1 previously 'nonce'

Data Structures

Three Python classes are provide by **`graspi`** (and **`grasp`**) as follows:

`class objective(name)`

A Python object of this class holds a GRASP objective. Its atributes are:

`.name`	Unicode string – the objective's name
`.neg`	Boolean – `True` if objective supports negotiation (default `False`)
`.dry`	Boolean – `True` if objective supports dry-run 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 Python object – the value of the objective (default `None`)

An ASA would create a new instance of the EX1 objective thus:

```
new_obj = grasp.objective("EX1")
```

and then set any attributes that it needs to. For most cases, either `neg` or `synch` needs to be `True`; they must not both be `True`. Alternatively, for a dry-run negotiation, `dry` should be `True`. Note that `neg` and `dry` are mutually exclusive for a given negotiation session.

`class asa_locator()`

Most ASAs don't need to create such an object, but it will be returned by a GRASP `discover` or `get_flood` function. Its attributes are:

`.locator`	The actual locator, either an `ipaddress` or a string
`.ifi`	The interface identifier via which this was discovered
`.expire`	`int(time.clock())` value when this entry expires (0=never)
`.diverted`	Boolean – `True` if the locator was discovered via a Divert option
`.protocol`	Applicable transport protocol (`IPPROTO_TCP` or `IPPROTO_UDP`)
`.port`	Applicable port number
`.is_ipaddress`	Boolean – if `True`, the locator is a Python `ipaddress`
`.is_fqdn`	Boolean – if `True`, the locator is an FQDN (string)
`.is_uri`	Boolean – if `True`, the locator is a URI (string)

`class tagged_objective()`

Most ASAs don't need to create such an object, but it is used by the GRASP `flood` and `get_flood` functions. Its attributes are:

`.objective`	A flooded objective

`.source` The `asa_locator` from which the flooded objective came

Functions

The ASA can call a bunch of Python functions. Almost all are identical in **`graspi`** and **`grasp`**. Here is a summary, followed by detailed descriptions:

`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 handle[2] that must be presented in all subsequent API calls.

`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. Registration enables discovery to occur, including assigning a dynamic port. Deregistration of objectives is possible (but is done automatically by `deregister_asa).`

`discover()`

This is used to discover peers for negotiation or synchronization.

`request_negotiate()`

Old API: **`grasp.req_negotiate()`**

This is used by a negotiation initiator to start a negotiation sequence (with a GRASP Request Negotiate 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`.

`send_invalid()`

Abrupt stop after invalid message. Normally, **`end_negotiate()`** is preferable.

`synchronize()`

This is used by a synchronization initiator to fetch a synchronized objective (normally with a GRASP Request Synchronize message).

`listen_synchronize(), stop_synchronize()`

2 previously 'nonce'

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 wishing to flood one or more GRASP objectives to the AN.

`get_flood()`

This is used by an ASA to fetch flooded objectives.

`expire_flood()`

This is used in special cases to mark a flooded objective as expired.

Old API only: **`grasp.gsend(), grasp.grecv()`**

These are additional to the GRASP specification and allow two ASAs to send arbitrary CBOR objects to each other.

All functions return an error code (integer) as their first return parameter. Zero means success. Positive integer means failure. Both **`graspi`** and **`grasp`** provide **`errors`** and **`etext`** as data. For error code `e`, the corresponding English language error string is `etext[e]`. The error codes have useful names, such as `errors.declined`. Error code details are in RFC8991.

There are also several utility functions: **`skip_dialogue(), tprint(), ttprint(), init_bubble_text(), dump_all()`**.

Registration functions:

```
####################################################################
# register_asa(asa_name)
#
# Tells the GRASP engine that a new ASA is starting up.
# Also triggers GRASP initialisation if needed.
#
# return zero, asa_handle  if successful
# return errorcode, None if failure
#
# Note - the ASA must store the asa_handle (an opaque Python
# object) and use it in every subsequent GRASP call.
####################################################################
```

The ASA name must be unique within a given GRASP instance. For ASA life cycle support, it could be a basic functional name plus a version number or timestamp.

```
####################################################################
# deregister_asa(asa_handle, asa_name)
#
# Tells GRASP that an ASA is exiting, deregisters its objectives.
#
# return zero if successful
# return errorcode if failure
####################################################################
```

```
#################################################################
# register_obj(asa_handle, objective, ttl=None,
#              discoverable=False, overlap=False,
#              local=False, rapid=False, locators=[])
#
# Store an objective that this ASA supports and may modify.
#
# The objective becomes available for discovery only after
# a call to listen_negotiate() or listen_synchronize()
# unless the optional parameter discoverable is True.
#
# ttl is discovery time to live in milliseconds; the default
# is the GRASP default timeout.
#
# if discoverable==True, the objective is *immediately*
# discoverable even if the ASA is not listening.
#
# if overlap==True, more than one ASA may register this objective.
#
# if local==True, discovery must return a link-local address.
# (also applies in DULL mode)
#
# if rapid==True, the supplied objective value will be sent
# in rapid mode (only implemented for synchronization).
#
# locators is an optional list of explicit asa_locators,
# trumping normal discovery.
#
# After registration, the ASA may negotiate the objective
# or use it to send synchronized or flooded data.
# Registration is not needed if the ASA only wants to
# receive synchronized or flooded data.
#
# May be repeated for multiple objectives.
#
# return zero if successful
# return errorcode if failure
#################################################################
```

`discoverable = True` is **not recommended** for normal use. It is for objectives that do not support negotiation or synchronization. **locators** is intended for such special objectives and is **not recommended** for normal use.

`overlap = True` is intended for ASA life cycle support, where old and new versions of the same ASA may need to overlap in time. It slightly complicates how objectives are registered and deregistered.

`local = True` is intended for infrastructure ASAs that must work on-link only.

```
####################################################################
# deregister_obj(asa_handle, objective)
#
# Stops all operations on this objective (if registered)
# by removing it from the registry. (Except for an objective
# with overlapped registrations.)
#
# return zero if successful
# return errorcode if failure
####################################################################
```

Discovery function:

```
##############################################################
# discover(asa_handle, objective, timeout,
#          flush=False, minimum_TTL=-1)
#
# Call in separate thread if asynchronous operation required.
# timeout in milliseconds (None for default).
#
# If there are cached results, they are returned immediately.
# If not, results will be collected until the timeout occurs.
#
# Optional parameter minimum_TTL (in milliseconds) will flush
# stale cached results first, if non-negative.
#
# Optional parameter flush=True will flush cached results first
# (equivalent to minimum_TTL=0).
#
# return zero, list of asa_locator if successful
#    If no peers discovered, the list is empty
# return errorcode, [] if failure
#                      Exponential backoff RECOMMENDED.
##############################################################
```

Example:

```
obj1 = grasp.objective("EX9")
err, locators = graspi.discover(asa_handle, obj1, None)
if err:
    #error handling goes here
elif locators = []:
    #nothing discovered
else:
    if locators[0].is_ipaddress:
        peer = locators[0]
        # we'll use the first discovered peer...
        # peer.locator is the IP address
        # peer.protocol is normally IPPROTO_TCP
        # peer.port is the port number to use
```

Use of **`minimum_TTL=0`** is recommended after several failed negotiation or synchronization attempts. Otherwise, no new discovery multicasts will be sent until the discovery cache times out.

ASAs requiring flooded values or mode synchronization need not call `discover()`. See `synchronize()`. Similarly, ASAs wishing to negotiate an objective with any available peer need not call `discover()`. See `request_negotiate()`.

Negotiation functions:

```
##############################################################
# request_negotiate(asa_handle, objective, peer, timeout)
#
# Request negotiation session with a peer ASA.
#
# asa_handle identifies the calling ASA
#
# objective must include the requested value
#
# The objective's loop_count value should be set to a suitable
# value by the ASA. If not, the GRASP default will apply.
#
# peer is the target node, an asa_locator returned by discover()
# If peer is None, discovery is performed first.
#
# timeout in milliseconds (None for default)
#
# Launch in a new thread if asynchronous operation required.
#
# Four possible return conditions are possible:
#
# 1) return zero, None, objective, None
#
# The peer has agreed; the returned objective contains
# the agreed value.
#
# 2) return zero, session_handle, objective, None
#
# Negotiation continues.
#
# The returned objective contains the first value offered by the
# negotiation peer. This instance of the objective MUST be used in
# subsequent negotiation steps because it contains the loop count.
#
# The ASA MUST store the session_handle (an opaque Python object)
# and use it in the subsequent negotiation steps.
#
# 3) return errors.declined, None, None, string
#
# The peer declined further negotiation, the string gives a reason
# if provided by the peer.
#
# 4) For any non-zero errorcode except errors.declined:
#    return errorcode, None, None, None
#
# The negotiation failed, errorcode gives reason.
# Exponential backoff RECOMMENDED before retry.
##############################################################
```

Special note for infrastructure ASAs:
`session_handle.id_source` will be the IP address of the remote ASA. This is expected to be needed by the ACP infrastructure ASA.

```
##############################################################
#
# Old API version available in the grasp module:
# grasp.req_negotiate(asa_handle, objective, peer, timeout)
#
# Request negotiation session with a peer ASA.
#
# asa_handle identifies the calling ASA
#
# objective must include the requested value
#
# Note that the objective's loop_count value should be set to a
# suitable value by the ASA. If not, the GRASP default will apply.
#
# peer is the target node; it must be an asa_locator as returned
# by discover().
# If peer is None, discovery is performed first.
#
# timeout in milliseconds (None for default)
#
# Launch in a new thread if asynchronous operation required.
#
# Four possible return conditions are possible:
#
# 1) return zero, None, objective
#
# The peer has agreed; the returned objective contains the
# agreed value.
#
# 2) return zero, session_handle, objective
#
# Negotiation continues.
#
# The returned objective contains the first value offered by the
# negotiation peer. This instance of the objective MUST be used in
# subsequent negotiation steps because it contains the loop count.
#
# The ASA MUST store the session_handle (an opaque Python object)
# and use it in the subsequent negotiation steps.
#
# 3) return errors.declined, None, string
#
# The peer declined further negotiation, the string gives a reason
# if provided by the peer.
#
# 4) For any non-zero errorcode except errors.declined:
#    return errorcode, None, None
#
# The negotiation failed, errorcode gives reason.
# Exponential backoff RECOMMENDED before retry.
##############################################################
```

```
##############################################################
# listen_negotiate(asa_handle, objective)
#
# Instructs GRASP to listen for negotiation requests
# for the given objective. Its current value is not significant.
#
# This function will block waiting for an incoming request.
# Call in a separate thread if asynchronous operation required.
#
# This call only returns after an incoming negotiation request
# and must be followed by negotiate_step and/or negotiate_wait
# and/or end_negotiate.
# listen_negotiate must then be repeated to restart listening.
#
# return zero, session_handle, requested_objective
#
# The requested_objective contains the first value requested by
# the negotiation peer. Note that this instance of the objective
# MUST be used in the subsequent negotiation calls because
# it contains the loop count.
#
# The ASA MUST store the session_handle (an opaque Python object)
# and use it in the subsequent negotiation calls.
#
# return errorcode, None, None    if failure
##############################################################
```

```
##############################################################
# stop_negotiate(asa_handle, objective)
#
# Instructs GRASP to stop listening for negotiation
# requests for the given objective.
#
# return zero if successful
# return errorcode if failure
##############################################################
```

```
############################################################
# negotiate_step(asa_handle, session_handle, objective, timeout)
#
# Continue negotiation session
#
# objective contains the next proffered value
# Note that this instance of the objective
# MUST be used in the subsequent negotiation calls because
# it contains the loop count.
#
# timeout in milliseconds (None for default)
#
# return: exactly like request_negotiate
#
# NOTE: The old API return for grasp.negotiate_step is exactly
# like grasp.req_negotiate)
############################################################

############################################################
# negotiate_wait(asa_handle, session_handle, timeout)
#
# Delay negotiation session
#
# timeout in milliseconds (None for default)
#
# return zero if successful, errorcode if failure
############################################################

############################################################
# end_negotiate(asa_handle, session_handle, result, reason="why")
#
# End negotiation session
#
# result = True for accept, False for decline
# reason = optional string describing reason for decline
#
# return zero if successful, errorcode if failure
#
# Note that a redundant call to end_negotiate will get a
# reply such as errors.noSession, which does not need
# to be treated as an error.
############################################################

############################################################
# send_invalid(asa_handle, session_handle, info="Diagnostic data")
#
# End session abruptly (see M_INVALID in GRASP specification)
#
# info = optional diagnostic data
# return zero if successful, errorcode if failure
############################################################
```

Synchronization and Flood functions:

```
################################################################
# synchronize(asa_handle, objective, locator, timeout)
#
# Request synchronized value of the given objective.
#
# locator is an asa_locator as returned by discover()
#
# timeout in milliseconds (None for default)
#
# If the locator is None and the objective was already flooded,
# the first non-expired flooded value in the cache is returned.
#
# Otherwise, synchronization with a discovered ASA is performed.
# In that case, if the locator is None, discovery is performed
# first, unless the objective is in the discovery cache already.
# If the discovery response provides a rapid mode objective,
# synchronization is skipped and that objective is returned.
#
# This call should be repeated whenever the value is needed.
# Call in a separate thread if asynchronous operation required.
#
# Since this is essentially a read operation, any ASA can do
# it. GRASP checks that the ASA is registered, but the
# objective doesn't need to be registered by the calling ASA.
#
# return zero, synch_objective   returns objective with its
#                                synchronized value
#
# return errorcode, None    synchronization failed
#                           errorcode gives reason.
#                           Exponential backoff RECOMMENDED.
################################################################
```

Note – a normal ASA can simply call `synchronize()` without concern whether the objective has been flooded or is available in rapid mode; it will simply receive the objective, if available, by the fastest possible method.

```
##############################################################
# listen_synchronize(asa_handle, objective)
#
# Instructs GRASP to listen for synchronization
# requests for the given objective, and to
# respond with the objective value given in the call.
#
# This call should be repeated whenever the value changes.
#
# return zero if successful
# return errorcode if failure
##############################################################

##############################################################
# stop_synchronize(asa_handle, objective)
#
# Instructs GRASP to stop listening for synchronization
# requests for the given objective.
#
# return zero if successful
# return errorcode if failure
##############################################################

##############################################################
# flood(asa_handle, ttl, *tagged_objective)
#
# Instructs GRASP to flood the given synchronization
# objective(s) and their value(s) to all GRASP nodes.
# Checks that the ASA registered each objective.
# This call may be repeated whenever the value changes.
#
# The 3rd parameter can be a single list of [tagged_objective,]
# as per the official API, or a repeated parameter
# of type tagged_objective.
# The tagged objective(s) are in the class tagged_objective,
# so must be tagged with a locator, which is either None or
# a valid asa_locator
#
# If the first objective is tagged with the unspecified
# address, the entire flood is treated as link-local:
#
#  - the address in the locator is replaced by the
#    relevant link local address
#  - the loop count is forced to 1
#
# ttl is in milliseconds (0 = infinity)
#
# return zero if successful
# return errorcode if failure
##############################################################
```

The tag will normally be **None**. Infrastructure ASAs needing to flood an {*address, protocol, port*} 3-tuple create an **asa_locator** object to do so. If *address* is the unspecified address ('::') it is replaced by the link-local address of the sending node in each copy of the multicast, which will be forced to have a loop count of 1.

```
################################################################
# get_flood(asa_handle, objective)
#
# Request unexpired flooded values of the given objective.
# This call should be repeated whenever the value is needed.
#
# Since this is essentially a read operation, any ASA can do
# it. GRASP checks that the ASA is registered but the
# objective doesn't need to be registered by the calling ASA.
#
# return zero, tagged_objectives   returns a list of
#                                  tagged_objective
#
# return errorcode, None    failed, errorcode gives reason.
#                           Exponential backoff RECOMMENDED.
################################################################
```

```
################################################################
# expire_flood(asa_handle, tagged_obj)
#
# Mark a flooded objective as expired
#
# 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.
#
# tagged_obj   the tagged_objective to be expired
#
# return zero if successful
# return errorcode if failure
################################################################
```

Send/Receive functions

These functions are available only in the old API (the **grasp.py** module) as an alternative to the standard negotiation functions. Two peers can send arbitrary CBOR messages to each other, in support of any non-GRASP protocol exchanges. They are sent within a GRASP session, encrypted if QUADS is in use.

Procedurally, the peer acting as a server must support a specific objective and call `listen_negotiate()`. When a peer wants to talk as a client, `listen_negotiate()` will return with a valid session handle. After that the server can use `grecv()` and `gsend()` alternately. Other negotiation functions are not used.

A peer that wants to act as a client uses normal GRASP discovery for the specific objective and then calls `grasp.req_negotiate()` with an extra optional parameter `noloop=True`. Then `req_negotiate()` will return with error code `noReply` and a valid session handle. After that the client can use `gsend()` and `grecv()` alternately. Other negotiation functions are not used. Whatever value the client provided in the objective sent with `req_negotiate()` will be delivered to the server by `listen_negotiate()`, and could be used as an initial message.

To allow multiple simultaneous clients, the server should spawn a new thread when `listen_negotiate()` returns, and then listen again.

It's a bit complicated to describe but simple enough to use. See the example peers `testserver.py` and `testclient.py`.

```
##############################################################
# gsend(asa_handle, session_handle, message)
#
# Sends over the socket for an opened negotiation session
#
# message is a Python object.
#
# return zero if successful
# return errorcode if failure
##############################################################

##############################################################
# grecv(asa_handle, session_handle, timeout)
#
# Receives over the socket for an opened negotiation session
#
# return zero, message if successful
#     message is a Python object.
# return errorcode, None if failure
##############################################################
```

Utility functions:

None of these functions returns a value.

```
#####################################################################
# skip_dialogue(testing=False, selfing=False, diagnosing=False,
#               quadsing=True, be_dull=False)
#
# Tells GRASP to skip its initial dialogue.
#
# Default is not test mode and not listening to own multicasts
# and not printing message syntax diagnostics
# and try QUADS security (unless DULL)
# and not DULL
# Must be called before register_asa()
#####################################################################

######################################################
# tprint(*whatever)
#
# Thread-safe printing; precedes the output with the
# thread's name and number.
#
# Call exactly like print()
######################################################

######################################################
# ttprint(*whatever)
#
# Thread-safe printing iff GRASP is running in test mode
# (test_mode == True). Used for detailed diagnostics during
# debugging & testing.
#
# Call exactly like print()
######################################################

######################################################
# init_bubble_text(caption)
#
# Switch on pretty bubble printing via tprint(),
# if tkinter is available.
#
# caption: a string that labels the bubble window.
######################################################

######################################################
# dump_all() prints various global data structures
#
# Intended only for interactive debugging
# and not thread-safe
######################################################
```

Implementation Notes

Ignoring comments and docstrings, there are about 2500 lines of code. Suggestions for improvement will be very welcome. The reader is assumed to be familiar with Python and have a Python 3.7 (or higher) environment available. There is a lot of threading and considerable use of sockets. Important Python modules used include: `threading`, `queue`, `socket`, `ipaddress`, `cryptography`.

The API module is **`graspi.py`**, which is mainly a small wrapper for the core. The core of GRASP, including the old API, is **`grasp.py`**. There is also a separate module called **`acp.py`**, whose API is briefly described below this section. They are not yet Python packages available via PIP. Some test modules are described below.

Everything is in a GitHub repository at https://github.com/becarpenter/graspy. The code is under a Simplified BSD licence.

The code is very talkative when running in test mode – lots of diagnostic prints.

Main global data structures and variables (for details, see comments in the source):

`_asa_registry` – where ASAs are registered

`_obj_registry` – where objectives are registered

`_discovery_cache` – where locators for discovered objectives are cached

`_session_id_cache` – where GRASP sesssion ids and ASA handles are cached.

`_flood_cache` – where flooded objectives and their values are cached.

All five of these are protected by locks, which must be used rigorously due to the amount of multithreading involved. Some other global variables:

`DULL`	`True` if running in DULL mode
`_grasp_initialised`	Initially `False`
`_skip_dialogue`	`True` if ASA called `skip_dialogue()`
`_crypto`	`True` if QUADS cryptography is running
`_secure`	`True` if GRASP is secured
`_tls_required`	`True` if TLS is needed (not supported)
`_rapid_supported`	`True` if rapid mode allowed (not used)
`_mcq`	FIFO queue for incoming multicasts
`_drq`	FIFO queue for pending discovery responses
`_my_address`	this node's preferred IPv6 address
`_session_locator`	address used to disambiguate session ids
`_ll_zone_ids`	list of the host's link-local zones etc.
`_mcssocks`	list of sockets for sending link-local multicasts

`_relay_needed`	`True` if multiple interfaces require multicast relaying
`test_mode`	`True` if module is running in test mode
`_mess_check`	`True` iff message parse error diagnostics are enabled
`_dobubbles`	`True` to enable bubbles (pretty printing)
`_listen_self`	`True` iff listening to own LL multicasts for testing. `_listen_self` may be set when `grasp.py` initialises, for testing within a single node.
`_i_sent_it`	A hack used to ignore own discovery multicasts when not in test mode – needed to run multiple instances in one node.

`_test_divert`
`_make_invalid`
`_make_badmess`
	Flags used for internal testing.

Threads:

The main GRASP thread exits after initialisation. Other threads may be active:

`_synch_listen`: This is a class of threads that will be activated by calls to `listen_synchronize`, one for each active listener.

(Note – this doesn't apply to `listen_negotiate`, which listens in-line rather than activating a separate thread.)

`_mclisten`: Listens for GRASP link-local multicasts (Discovery messages and Flood messages) and queues them for handling by `_mchandler`.

`_mchandler`: Handles Discovery and Flood messages. It's separate from `_mclisten` so that the sockets don't get jammed up waiting for the previous message to be handled.

`_disc_relay`: A class of threads activated when Discovery messages have to be relayed to another interface, one for each discovery action.

`_drlisten`: A class of threads activated during disvovery to wait for TCP Responses, one for each discovery session.

`_tcp_listen`: `A` class of threads activated by `_listen_synchronize` or `_listen_negotiate` to await TCP synch and negotiate Requests and queue them for the appropriate negotiation or synchronization listener.

`_watcher`: Keep an eye on things. In the prototype, this makes an attempt to recover from address renumbering or CPU sleep/wakeup. In production code, it would monitor link interfaces, add newly active ones, and delete inactive ones, updating data structures accordingly. It would force a switch to TLS if the ACP goes down, and strictly limit operations when neither ACP nor TLS is available.

ACP interface

The **acp.py** module provides three functions:

acp.new2019(): Always returns **True**. Its purpose is to allow the GRASP core to check that it has loaded a recent ACP module. Older versions do not provide this.

acp.status(): Returns a security status text. The intention is that an insecure ACP would return **False**.

acp._get_my_address(build_zone=False):

This returns the current global scope IPv6 address that GRASP should use as its primary locator (preferring a ULA if available). If there is no such address (i.e. only link-local addresses are available) it will return **None**.

If the optional parameter **build_zone** is **True** the function returns a second parameter, a list of **[Interface_index, LL_address]** pairs for all valid IPv6 interfaces. The interface index is an integer, not an interface name, to maximise portability between operating systems.

In all cases addresses are returned in the **ipaddress.IPv6Address** class.

Discovery Unsolicited Link-Local (DULL)

This is intended for a couple of infrastructure components during secure enrolment of devices and the formation of the ACP. It should not be used otherwise.

If the **DULL** flag is set during the initial dialogue or by **skip_dialogue()**, the code will operate in a restricted link-local mode. The **_secure** flag will be **False** and most API calls will return a **noSecurity** error. QUADS will not be available. Even if implemented, TLS usage would not be available. The various restrictions for DULL mode in the GRASP specification will be applied.

QUick And Dirty Security for GRASP (QUADS)

This is an implementation of draft-carpenter-anima-quads-grasp, using the Python **cryptography** module. A recent version is needed to ensure that SHA256 is fully supported. (I have tested with **cryptography** 2.7 and 2.8.)

Encryption Only

Firstly, pick a keying password for the domain (any sequence of printing characters that exist on your keyboards). When GRASP starts up, it will ask for this password unless cryptography keys are already installed. If you don’t want to be secure, enter an empty password and QUADS will not be used. Use the same password for all instances of GRASP in the same domain.

If you run modules that do not trigger the initial dialogue, i.e. they call **`skip_dialogue()`**, you need to generate the crypto keys in advance. In the same directory as **`grasp.py`**, run the utility **`quadsmaker.py`**, which will ask for the keying password and create a file **`quadsk.py`** in the same directory. A module that needs this is the **`gremlina.py`** daemon (see below).

If you want to stop using QUADS, delete all copies of **`quadsk.py`**.

Your GRASP domain will be exactly as secret as your keying password and, if applicable, as your **`quadsk.py`** file.

If you happen to mix GRASP instances with and without QUADS, or instances with different keying passwords, each set of instances will work fine on its own. However, multicasts go everywhere, so frequent "decode error" warnings may appear, since each subset cannot decrypt multicasts from any other set.

QUADS Key Infrastructure

An elementary way of securely distributing QUADS keys is available, as a proof of concept (so it is a bit clumsy to use). It uses the term "pledge" for a node wishing to join the QUADS domain, i.e. wishing to obtain the QUADS keys. This terminology is adopted from BRSKI (RFC8995). The pledge uses asymmetric cryptography (RSA) over insecure GRASP to fetch the keys from a master, after which GRASP can run securely in the pledge as well as the master.

Choose one particular node as the master; it logically replaces the BRSKI domain registrar. Run **`quadsmaker.py`** in that node as above to generate a **`quadsk.py`** file. Then run **`quadski.py`** in the master node for ever. It will start by asking you to enter a password, known as the pledge password. This should be different from the keying password. Your GRASP domain will be exactly as secret as your two passwords.

In each node that wants to join the domain, initially run **`qpledge.py`** which will start by asking you to enter the pledge password. It will then use GRASP to fetch the keys from the master and store them locally; if this succeeds *and* the node is a GRASP relay node it will do two more things:

1. Start an encrypted version of the **`gremlina.py`** daemon, which is necessary in a relay node.

2. Remain active as an unencrypted GRASP daemon, which is necessary to support other pledges running the QUADSKI process. It logically replaces the BRSKI proxy.

Note that it would be possible to create a domain-specific version of **`qpledge.py`** with the pledge password built in. But that would not be very secure, so at present the pledge password must be entered manually. This is really the downside of QUADSKI compared to BRSKI.

Testing

If you don't have IPv6, you may as well stop reading here. The code assumes you have IPv6 up and running, with either a globally routable or ULA prefix. It will run with only link-local addresses, but then off-link sessions will fail, of course.

Two cases do not work (and should not work):

1. Some hosts have a global scope IPv6 address, and others only have link-local.
2. Some hosts have ULAs, and others only have globally-routeable unicast addresses. (Having both is OK; the ULA is preferred by the ACP.)

The code uses the IANA-assigned port:
GRASP_LISTEN_PORT = 7071

The code uses the IANA-assigned link-local multicast address:
ALL_GRASP_NEIGHBOR_6 = ff02::13

You need Python 3 and the code has not been tested on any version before Python 3.4. It will not run on Python 2. (On most Linux installations, `python` invokes Python 2, and `python3` invokes Python 3.)

You need various standard Python modules that everybody should have, and also the CBOR module. `pip3 install cbor` should fetch it for you. (If you don't have `pip`, see https://packaging.python.org/installing/ .) On Linux, you also need `netifaces`.

Make a local `graspy` directory/folder containing the relevant `.py` files (especially `graspi.py`, `grasp.py`, `acp.py`), and make that your working directory. Because of the socket usage in GRASP, you may need Administrator or `sudo su` privilege. On Windows 10 you may need to authorize Python in Windows Defender Firewall.

To run tests on Windows, open `graspitests.py` in the IDLE editor and run the module from there. On Linux or MacOS, run from IDLE or try `python3 -i graspitests.py.` Once it has initialised, type `ASAtest()` at the Python prompt and stuff will happen.

The `graspitests` module does nothing very exciting but it tests many pathways in the `grasp` module. Hopefully there will be a lot of output (many screens) for several minutes, and no Python exceptions. If there are any, please tell me. Sending fixed code would be even better ;-).

The code and test suite are set up to run on a single host that hears its own broadcasts. Also, if the host has more than one IPv6 interface, it will relay its own broadcasts between its own interfaces. That multiplies the amount of output in an entertaining way, but all the relay threads should terminate after one loop. (If that fails, they eventually expire their loop counts or they time out).

When running in a single node listening to itself, the discovery cache ends up containing duplicate addresses. That's an artefact of the testing environment.

The final part of the test is a negotiation between two threads. The starting point for each negotiator is a random value, and there are a couple of other randomised conditions, so each run will be different, including some error cases. Look for output messages from threads Neg1 and Neg2 (example below).

Do not run this test suite more than once without restarting the Python context; it leaves the GRASP data structures dirty so that they can be analysed afterwards.

`grasp.py` isn't yet structured as a fully-fledged Python package. You could test it as follows: Write your test case as its own Python suite, for example as a file `MyTests.py`. Save it in the same directory as `graspi.py`, `grasp.py` and `acp.py`. The test code needs to `import graspi`. Run the code; when GRASP is first called it will perform its initialisations (with a modest amount of FYI printout - example below). Then the test code can do what it needs to do, starting with `graspi.register_asa(...)` and `graspi.register_obj(...)`.

To get clean printout from the testing thread(s) use `graspi.tprint(...)`.

To get diagnostic printouts from inside GRASP, set `graspi.grasp._test_mode=True` . (But some diagnostic printouts are commented out in the code.)

When running on multiple nodes, be careful to ensure that GRASP is **not** listening to its own multicasts (answer No during initialisation, which sets `_listen_self` to `False)`. I had a problem during such tests because the building switch blocked link local multicasts to 'unknown' addresses, so I had to put the nodes on a dumb switch of their own. If you can't do that, the last resort is to change the relevant constant inside `grasp.py` to ff02::1, which is the all-nodes link-local multicast address.

Good switches allow ff02::13 when GRASP nodes send an MLD listener report (join) for that address. Windows 7 and above, and Linux, send MLD, for example. But badly designed or misconfigured switches might fail to correctly implement MLD.

You can have fun by running two toy ASAs on multiple machines. My examples are called "ASA Briggs" and "ASA Gray" (they both have Wikipedia entries). They negotiate with each other (Gray is the initiator and Briggs is the listener).

They randomize the starting conditions for each negotiation, so any kind of result is possible. Note that if you run them both in separate Python instances on one machine, <u>and</u> tell them to listen to their own multicasts, they will interact correctly in the one machine. The latest version of Briggs can support multiple simultaneous instances of Gray. As with **`graspitests`**, I would like to hear details of any exceptions or issues.

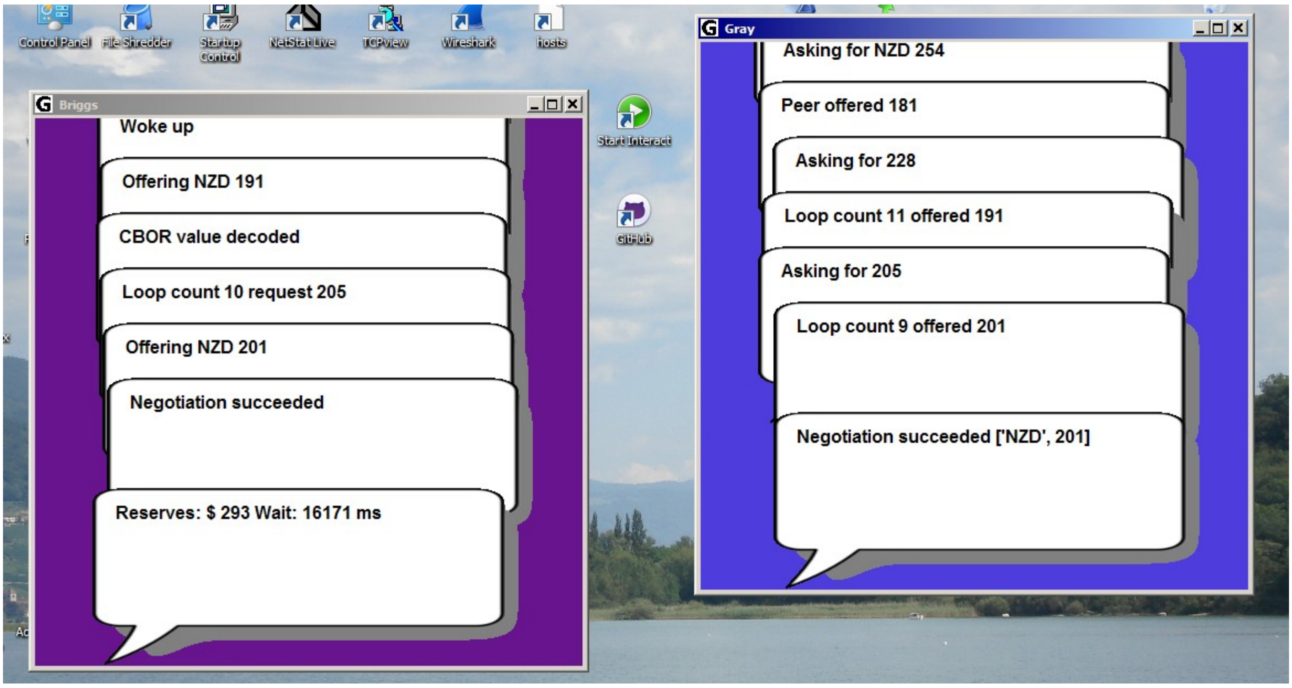
A node that is a GRASP relay with no ASAs must run a plain GRASP daemon. Use **`gremlin.py`** if you want the initial dialogue, or **`gremlina.py`** if you want no dialogue. This is not necessary if running **`qpledge.py`** as described above.

Another sample ASA is **`pfxm3.py`**. It models the IPv6 prefix management use case ([RFC8992](RFC8992)). Multiple instances can interoperate.

Some example outputs follow (they are not from the latest code, so details may vary).

Screen shot extracted from the Gray/Briggs ASA test

(See the pretty bubbles; this might not work on all systems. It uses the `tkinter` package.):



GRASP initialisation example:

```
WARNING: This is insecure prototype code unsuitable
for production use, used at your own risk.
Version 05.0BC-20160503 released under the simplified BSD license.
Starting in 10 9 8 7 6 5 4 3 2 1
_MainThread 652  Initialised global variables, registries and caches.
_MainThread 652  ACP status is  True
_MainThread 652  My address:  2406:e007:56d8:1:28cc:dc4c:9703:6781
_MainThread 652  Session locator:  2406:e007:56d8:1:28cc:dc4c:9703:6781
_MainThread 652  Link local zone index(es):
_MainThread 652  [12, IPv6Address('fe80::28cc:dc4c:9703:6781')]
_MainThread 652  Listen to own multicasts? Y/N
y
_MainThread 652  WARNING: Will listen to own LL multicasts
_drlisten 2908  Discovery response listener for interface  12  is up
_MainThread 652  Multicast relay not needed
_mclisten 1520  LL multicast listener is up
_mchandler 4728  Multicast queue handler up
_watcher 572  ACP watcher is up; thread count:  5
_MainThread 652  GRASP startup thread exiting
```

Sample negotiation output extracted from the test suite

(Detailed diagnostics have been deleted.)

```
Neg1 3356  Reserves: $  209  wait:  38018
Neg1 3356  listen_negotiate: Waiting for a negotiate request
Neg2 4900  Asking for $  335
Neg2 4900  Got nonce  2212917
Neg2 4900  Assembled Python message  [1, 4417678, ['EX2', 1, 6, 0]]
Neg2 4900  Waiting for discovery response
Neg2 4900  Entering drloop
Neg2 4900  Adding objective to discovery cache
Neg2 4900  Waiting for discovery response
Neg2 4900  Entering drloop
Neg2 4900  Adding locator to discovery cache
Neg2 4900  Waiting for discovery response
Neg2 4900  Discovered locator  2406:e007:59f5:1:28cc:dc4c:9703:6781
Neg2 4900  Sending req_negotiate to  2406:e007:59f5:1:28cc:dc4c:9703:6781
Neg2 4900  Assembled Python message  [3, 2260325, ['EX2', 1, 6, ['NZD', 335]]]
Neg1 3356  listen_negotiate: Got negotiate request from queue
Neg1 3356  listened, answer  EX2  ['NZD', 335]
Neg1 3356  Assembled Python message  [4, 2260325, ['EX2', 1, 5, ['NZD', 104.5]]]
Neg2 4900  negloop: CBOR->Python:  [4, 2260325, ['EX2', 1, 5, ['NZD', 104.5]]]
Neg2 4900  negloop: got NEGOTIATE
Neg2 4900  Assembled Python message  [4, 2260325, ['EX2', 1, 5, 251.25]]
Neg1 3356  negloop: CBOR->Python:  [4, 2260325, ['EX2', 1, 5, 251.25]]
Neg1 3356  negloop: got NEGOTIATE
Neg1 3356  Assembled Python message  [6, 2260325, 38018]
Neg1 3356  Tried wait:  True  None
Neg2 4900  negloop: CBOR->Python:  [6, 2260325, 38018]
Neg2 4900  negloop: got WAIT
Neg1 3356  Woke up
Neg1 3356  Assembled Python message  [4, 2260325, ['EX2', 1, 4, ['NZD', 156.75]]]
Neg2 4900  negloop: CBOR->Python:  [4, 2260325, ['EX2', 1, 4, ['NZD', 156.75]]]
Neg2 4900  negloop: got NEGOTIATE
Neg2 4900  Assembled Python message  [4, 2260325, ['EX2', 1, 4, 201.0]]
Neg1 3356  negloop: CBOR->Python:  [4, 2260325, ['EX2', 1, 4, 201.0]]
Neg1 3356  negloop: got NEGOTIATE
Neg1 3356  Assembled Python message  [5, 2260325, [102, 'Insufficient funds']]
Neg2 4900  negloop: CBOR->Python:  [5, 2260325, [102, 'Insufficient funds']]
Neg1 3356  Exit
Neg2 4900  Negotiate_step: got END
Neg2 4900  Step2 gave:  False  None  Insufficient funds
Neg2 4900  Peer reject:  Insufficient funds
Neg2 4900  Exit
```

Output extracted from the Gray/Briggs ASA test

(some messages omitted)

```
_MainThread 6112  ASA Gray is starting up
_MainThread 6112  ASA Gray registered OK
_MainThread 6112  Objective EX3 registered OK
_MainThread 6112  Ready to negotiate EX3 as requester

_MainThread 6112  Asking for  NZD  498
_MainThread 6112  Peer offered  172
_MainThread 6112  Asking for  448
_MainThread 6112  Loop ct  15  offered  182
_MainThread 6112  Asking for  403
_MainThread 6112  Loop ct  13  offered  192
_MainThread 6112  Asking for  362
_MainThread 6112  Loop ct  11  offered  202
_MainThread 6112  Rejecting unacceptable offer

_MainThread 6112  Asking for  NZD  281
_MainThread 6112  Peer offered  130
_MainThread 6112  Asking for  252
_MainThread 6112  Loop ct  2  offered  140
_MainThread 6112  Asking for  226
_MainThread 6112  Loop ct  0  offered  150
_MainThread 6112  Asking for  203
_MainThread 6112  Peer reject:  Loop count exhausted

_MainThread 6112  Asking for  NZD  135
_MainThread 6112  Peer offered  130
_MainThread 6112  Negotiation succeeded  ['NZD', 130]
-------------------------------------------------------------------
_MainThread 188  ASA Briggs is starting up
_MainThread 188  ASA Briggs registered OK
_MainThread 188  Objective EX3 registered OK
_MainThread 188  Ready to negotiate EX3 as listener

_MainThread 188  Reserves: $  345  wait:  12517
_MainThread 188  Got request for  NZD  498
_MainThread 188  Starting negotiation
_MainThread 188  Offering NZD  172
_MainThread 188  Loop ct  16  request  448
_MainThread 188  Offering NZD  182
_MainThread 188  Loop ct  14  request  403
_MainThread 188  Offering NZD  192
_MainThread 188  Loop ct  12  request  362
_MainThread 188  Offering NZD  202
_MainThread 188  Failed:  You are mean!

_MainThread 188  Reserves: $  261  wait:  12774
_MainThread 188  Got request for  NZD  281
_MainThread 188  Starting negotiation
_MainThread 188  Offering NZD  130
_MainThread 188  Loop ct  3  request  252
_MainThread 188  Tried wait:  True  None
_MainThread 188  Woke up
_MainThread 188  Offering NZD  140
_MainThread 188  Loop ct  1  request  226
_MainThread 188  Tried wait:  True  None
_MainThread 188  Woke up
_MainThread 188  Offering NZD  150
_MainThread 188  Failed:  No reply to negotiation step
_MainThread 188  Reserves: $  260  wait:  11616
_MainThread 188  Got request for  NZD  135
_MainThread 188  Starting negotiation
_MainThread 188  Offering NZD  130
_MainThread 188  Negotiation succeeded
```