Python prototype code of a GRASP engine and API

Brian E. Carpenter
brian.e.carpenter@gmail.com
June 2017

Introduction

This note describes a prototype implementation of GRASP written in Python 3.4, based on draft-ietf-anima-grasp-13.txt. It is not guaranteed or validated in any way and is both incomplete and probably wrong. It makes no claim to be production-quality code. Its main purpose is to help improve the protocol specification.

It's demonstration code written in an interpreted language, so performance is slow.

SECURITY WARNINGS:

- Assumes ACP up on all interfaces or none; does not watch for interface up/down changes.
- Runs with a dummy ACP module that is always "up".
- Does not explicitly support the "limited security instances" described in the spec.

So actually there is no security whatever.

LIMITATIONS:

- Only coded for IPv6, any IPv4 is accidental
- Survival of address changes and CPU sleep/wakeup is patchy
- FQDN and URI locators are allowed in discovery responses, but otherwise they are not handled at all.
- No code for rapid mode
- The relay code is lazy (no rate control)
- All unicast transactions use TCP (no unicast UDP)
- Optional Objective in Response messages: not implemented
- There are work-arounds for defects 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 API for use by ASAs, plus implementation and testing.

Acknowledgements: 神明達哉 (JINMEI Tatuya) for help with POSIX compatibility, Ulrich Speidel for loaning a Linux test box, and at IETF 98: Michael Richardson, William Atwood, Jéferson Campos Nobre and Bing Liu.

API

An ASA[1] written in Python 3 will need to **`import grasp`**.

Data Structures

It can then use data structures (Python classes) 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 0)

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)

1 or any other program wishing to use GRASP directly

***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 also use a bunch of Python functions. 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 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. 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.

`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`.

synchronize()

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

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 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.

WARNING – error handling in the API was changed in January 2017. Any code written earlier must be updated accordingly. All functions now return an error code (integer) as their first return parameter.

Error code convention: zero means success. Positive integer means failure. For error code `errorcode`, the corresponding English language error string is `grasp.etext[errorcode]`. The error codes have useful names, such as `grasp.errors.declined`, which is the only one likely to be needed by an ASA programmer. Full details are in the Python source.

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

The ASA name is arbitrary, and the only requirement is that it is unique within a given GRASP instance. For ASA life cycle support, it could for example consist of a basic functional name concatenated with a version number or timestamp.

```
#############################################################
# deregister_asa(asa_nonce, asa_name)
#
# Tells the GRASP engine that an ASA is going away.
# Deregisters its objectives too.
# We need this to happen automatically when an ASA crashes.
#
# return zero if successful
# return errorcode if failure
#############################################################
```

```
#################################################################
# register_obj(asa_nonce, objective,ttl=None
#              discoverable=False, overlap=False, local=False)
#
# 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.
# (NOT supported in this implementation.)
#
# if local==True, discovery must return a link-local address.
#
# May be repeated for multiple objectives.
#
# return zero if successful
# return errorcode if failure
#################################################################
```

`discoverable = True` is **not recommended** for normal objectives. It is intended for objectives that are only defined for GRASP discovery, and which do not support negotiation or synchronization.

`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 significantly complicates how objectives are registered and discovered.

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

```
#################################################################
# deregister_obj(asa_nonce, objective)
#
# Stops all operations on this objective (if registered)
# by removing it from the registry.
#
# return zero if successful
# return errorcode if failure
#################################################################
```

```
#############################################################
# discover(asa_nonce, objective, timeout, flush=False)
#
# 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 flush=True will flush cached results first
#
# return zero, list of asa_locator if successful
#    If no peers discovered, the list is empty
# return errorcode, [] if failure
# Exponential backoff RECOMMENDED before retry.
#############################################################
```

Example:

```
obj1 = grasp.objective("EX9")
err, locators = grasp.discover(asa_nonce, 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 IPPROTO_TCP or IPPROTO_UDP
        # peer.port is the port number to use
```

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

```
############################################################
# req_negotiate(asa_nonce, objective, peer, timeout)
#
# Request negotiation session with a peer ASA.
#
# asa_nonce 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.
#
# return zero, session_nonce, objective
#
# The returned objective contains the first value proffered 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_nonce (an opaque Python object)
# and use it in the subsequent negotiation calls
#
# return zero, None, objective - returns accepted value
# return grasp.errors.declined, None, string - other end declined,
#                                              string gives reason
# return errorcode, None, None  - negotiation failed,
#                                 errorcode gives reason,
#                                 exponential backoff RECOMMENDED
#                                 before retry.
############################################################
```

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

```
#################################################################
# listen_negotiate(asa_nonce, objective)
#
# Instructs GRASP to listen for negotiation
# requests for the given objective.
#
# Parameter is the objective of interest
#
# 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 req_negotiate
# 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_nonce, 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_nonce (an opaque Python object)
# and use it in the subsequent negotiation calls.
#
# return errorcode, None, None    if failure
#################################################################



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

```
#################################################################
# negotiate_step(asa_nonce, session_nonce, 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 req_negotiate
#################################################################


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


#################################################################
# end_negotiate(asa_nonce, session_nonce, result, reason="why")
#
# End negotiation session
#
# result = True for accept, False for decline
# reason = optional string describing reason for decline
#
# return zero if successful
# return errorcode if failure
#
# Note that a redundant call to end_negotiate will get a
# reply such as (False, "No session") which does not need
# to be treated as an error.
#################################################################
```

```
#############################################################
# synchronize(asa_nonce, 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.
#
# 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. Therefore we check 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
#                            before retry.
#############################################################
```

```
#################################################################
# listen_synchronize(asa_nonce, 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_nonce, objective)
#
# Instructs GRASP to stop listening for synchronization
# requests for the given objective.
#
# return zero if successful
# return errorcode if failure
#################################################################
```

```
#############################################################
# flood(asa_nonce, 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.
#
# Each objective is tagged with an asa_locator or with None.
#
# 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_nonce, 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. Therefore we check 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.
#############################################################

#############################################################
# expire_flood(asa_nonce, 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
#############################################################
```

Implementation Notes

This is my first real Python program, so it was a voyage of discovery. Ignoring comments and docstrings, there are about 1700 lines of code. Suggestions for improvement will be very welcome. From here, I assume you are familiar with Python and have a Python 3.4 environment available. There is a lot of threading and considerable use of sockets. I suggest studying the relevant Python modules first: `threading`, `queue`, `socket` and `ipaddress`. I didn't use `socketserver`.

The module is called `grasp`. There is also a dummy module called `acp`. They are not yet real Python packages. Some test modules are described below.

At the moment everything is in a directory called `graspy` found at https://www.cs.auckland.ac.nz/~brian/graspy/. It will hopefully be packaged and moved to GitHub at some point. 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 nonces 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. Other global variables:

`_tls_required`	True if no ACP
`_secure`	True if either ACP or TLS is working
`_rapid_supported`	True if rapid mode allowed
`_mcq`	FIFO queue for incoming multicasts
`_drq`	FIFO queue for pending discovery responses
`_my_locator`	this node's first global 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 Discovery/Flood relaying
`test_mode`	True iff module is running in test mode

`listen_self`	True iff listening to own LL multicasts for testing. WARNING: `listen_self` is set interactively when `grasp.py` initialises, for testing with only one 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`	True in some tests to force a pretend divert message

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 synchronisation 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 a clumsy 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.

Utility functions:

`tprint(*whatever)` - This does exactly what Python 3 `print()` does except that it's thread safe and precedes the output with the thread's name and number.

`ttprint(*whatever)` - This does exactly what `tprint()` does but only if `test_mode` is `True`. Used for detailed diagnostics during debugging & testing.

`init_bubble_text("Caption")` - This initialises pretty-printing; `tprint()` will generate a speech bubble window, if the Tkinter package is available.

`dump_all()` - Diagnostic print of the main GRASP data structures for debugging.

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.

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

Pending IANA assignment, the code uses an experimental address from RFC 4727:
ALL_GRASP_NEIGHBOR_6 = ff02::114
The resulting traffic should be harmless unless it collides with another person's tests!

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/ .)

Firstly make a local copy of the `graspy` directory/folder containing at least all the `.py` files, and make that your working directory. Because of the socket usage in GRASP, you need Administrator or `sudo su` privilege.

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

The `grasptests` 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).

FYI, 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 `grasp.py` and `acp.py`. The test suite needs to `import grasp`. Run the test suite; when GRASP is imported it will perform its initialisations (with a modest amount of FYI printout - example on next page). Then the test code can do what it needs to do, starting with `grasp.register_asa(...)` and `grasp.register_obj(...)`.

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

To get diagnostic printouts from inside GRASP, set `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 ff02::114, 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::114 when GRASP nodes send an MLD listener report (join) for that address. Windows 7 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 two different 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, and 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 `grasptests`, I would like to hear details of any Python exceptions or other issues.

Another sample ASA is unromantically named "pfxm1.py". It models the IPv6 prefix management use case (draft-ietf-anima-prefix-management). Multiple instances can interoperate.

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

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
```

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.):