BRSKI demonstration ASAs

Brian Carpenter

November 2017

This document is a quick report on the Python BRSKI demonstration modules using GRASP. If you don't know what BRSKI is, please read https://tools.ietf.org/html/draft-ietf-anima-bootstrapping-keyinfra

Specifically, the latest code supports the intention of draft-ietf-anima-bootstrapping-keyinfra-09, with a small correction to the registrar objective format.

If you don't know what GRASP is, please read https://tools.ietf.org/html/draft-ietf-anima-grasp

There are three modules in this demo. The latest published versions are at https://github.com/becarpenter/graspy

They are coded in Python 3.4 and will fail with Python 2.

They are intended only to show how BRSKI registrars, proxies, and joining nodes (pledges) could use GRASP for discovery. They don't include code for the guts of BRSKI, but they contain some pretend logic (and output messages) where the BRSKI code would fit. Also some choices are made at random, including simulated failures, so results *will* vary.

Note - these versions use the latest GRASP API with integer error codes. INCOMPATIBLE WITH PYTHON GRASP RELEASES BEFORE 2017!

The code, like all my GRASP prototype code, is under the Simplified BSD open source license.

Reggie.py

This is demo code for a BRSKI registrar. The registrar is found by autonomic nodes using GRASP discovery and synchronization. The corresponding GRASP objective is named "AN_join_registrar" and has a null value field.

The objective is flooded out to all interested nodes (i.e. those running a BRSKI proxy) with one or more locators, as allowed by the M_FLOOD message. Each locator is a GRASP locator option in the usual format:

    [O_IPv6_LOCATOR, address, protocol, port]

Each BRSKI method might need its own port at the registrar side. That's why there is a locator associated with each method. The protocol indicates the BRSKI method supported.

See the Python code for further details.

Procksy.py and Pledji.py

These are two demo modules of a BRSKI join proxy and joining node (pledge), using an on-link flooding model to announce the proxy to the pledges.

The proxy collects all available instances of "AN_join_registrar" by use of the GRASP *get_flood()* functions Then it chooses which instances to proxy for, and announces itself accordingly to all on-link neghbours using GRASP flooding. The flood is limited to one hop (i.e. loop_count=1) and the GRASP API forces the use of a link-local address. (For GRASP experts, this is known as poor man's DULL.)

The corresponding GRASP objective is named "AN_proxy" and has an empty value field. GRASP *flood()* allows the proxy to associate a locator with each flooded objective. Note that the proxy only announces a link-local address to the pledges, and its own port number for each method. Otherwise, the locators are as described for "AN_join_registrar".

The pledge collects all available instances of "AN_ proxy" by use of the GRASP API *get_flood()* function, and chooses which one it wants to use. Since the proxy is on link and there will often be only one proxy per link, we expect the choice to be based on the preferred BRSKI method.

See the Python code for further details.

Running the code

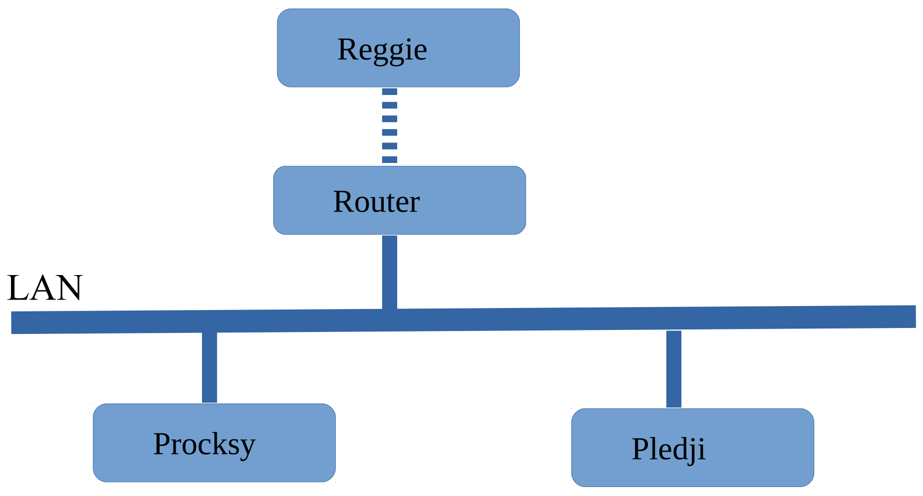
For background and basic instructions for running Python GRASP see **graspy.pdf**, which should be in the same folder as this file, as well as the necessary support modules (**grasp.py** and **acp.py**).

Run all three modules. Don't forget you need to be Administrator (Windows) or su (Linux/MacOS). Double click the files on Windows; on Linux or MacOS do

```
>python3 Reggie.py
```

etc.

Running on a single machine, type "y" whenever it asks "Listen to own multicasts?" On two machines connected to the same LAN, start Reggie and Pledji on one machine, Procksy on another, and type "n" to the multicast questions. That will create three separate GRASP instances across the two machines. If you have three machines, there could be a router between Reggie and Procksy, but Procksy and Pledji must be on the same LAN. Like this, for example:



In any configuration, start the modules in any order; they will eventually get together*. The outputs should be self-explanatory.

You should be able to run any number of registrars, proxies and pledges simultaneously. However, one registrar per network, and one proxy per LAN segment, is sufficient.

Then read the code, in order to really understand what's happening.

* If an unexpected event happens, like one of the machines going to sleep and waking up, or if Reggie dies and has to be restarted, the prototype usually recovers, although it can take several minutes for a stale discovery result or cached flood to expire. But there may well be cases where recovery fails; it's only a prototype.