pyFF Optimizations

GÉANT GN4-3 WP5 T2 Incubator topic Cycle III, 2020

Report by Team Alpha

Introduction

pyFF is widely used in our community to provide Discovery and Metadata Query services. This incubator activity investigated whether any significant memory optimizations can be made in pyFF.

When processing the eduGAIN metadata, pyFF memory usage balloons to the gigabytes, thereby inflicting some extra cost when running in procured VM-s like AWS. The startup/restart process speed, and service behavior while being started/restarted may also be improved. In particular, the service should never throw 5xx errors while in a normal startup/shutdown process.

The goal of this project is to optimize pyFF memory consumption and (re-)start behavior.

For the memory consumption, the underlying XML processing library may be swapped, or the memory-intensive part of the processing may be done on a short-lived cheap VM and the resulting in-memory representation serialized, transferred to the production instances and de-serialized.

For the in-(re)start behavior it must be established what is the right way of configuring pyFF so that it won't take queries while its internal database is still incomplete.

Experiments done

- Memory consumption charts
- guppy
- top with 1s refresh rate
- Externalized XML Processing

- SAX parsing
- visualization of the memory structure
- GC debug mode
- Meeting with Leif

Memory profiling

This is the bare import and code usage of using heapy to print heap information while running python code.

```
https://pkgcore.readthedocs.io/en/latest/dev-notes/heapy.html
from guppy import hpy
import code
hp=hpy()
...
# reset the heap counters
hp.setrelheap()
...

# just print the heap somewhere:
h = hp.heap()
log.debug(f"\nheapy: {h}")

# or possibly interrupt the code execution and inspect the hp object:
code.interact(local=dict(globals(), **locals()))
```

A typical dump in pyFF's mainloop then looks like this

```
Partition of a set of 117936 objects. Total size = 75634733 bytes.
Index Count  %    Size  % Cumulative % Kind (class / dict of class)
    0  1771  2 59385607 79 59385607 79 bytes
    1 25917 22 8466168 11 67851775 90 dict (no owner)
    2 25388 22 2640352  3 70492127 93 dict of pyff.samlmd.EntitySet
    3 23656 20 2232132  3 72724259 96 str
    4 25388 22 1218624  2 73942883 98 pyff.samlmd.EntitySet
    5  6795  6  380520  1 74323403 98 lxml.etree._Element
    6  2501  2  199024  0 74522427 99 tuple
```

```
    7   870  1  154828  0 74677255 99 types.CodeType
    8  1024  1  139264  0 74816519 99 function
    9    45  0  127872  0 74944391 99 dict of module
<196 more rows. Type e.g. '_.more' to view.>
```

Another way of profiling pyFF's memory usage is just following RES in top or htop for a long-running pyFF/gunicorn process, that has a 60s refresh interval. I normally use this pipeline

```
- when update:
 - load:
   - edugain.xml
- when request:
 - select:
 - pipe:
   - when accept application/samlmetadata+xml application/xml:
     - first
     - finalize:
         cacheDuration: PT12H
         validUntil: P10D
     - sign:
         key: cert/sign.key
         cert: cert/sign.crt
     - emit application/samlmetadata+xml
     - break
   - when accept application/json:
     - discojson
     - emit application/json
     - break
```

to feed the edugain feed that has been dowloaded using

```
$ curl http://mds.edugain.org/ -o edugain.xml
```

Un/Pickling etree.ElementTree object

Here we demonstrate that externally parsed etree.ElementTree objects can be pickled (serialized) to be consumed later in pyFF, without the need to parse.

```
from lxml import etree, objectify
import pickle
# Create pickled datafile
source = open("edugain.xml", "r", encoding="utf-8")
sink = open("edugain.pkl", "w")

t = objectify.parse(source)
p = pickle.dumps(t).decode('latin1')
sink.write(p)

# Read pickled object back in pyFF
def parse_xml
    return pickle.loads(io.encode('latin1'))
```

In metadata parser:

```
t = parse_xml(content) #Instead of parse_xml(unicode_stream(content))
```

Using un/pickling, pyFF's gunicorn starts out using ~800Mb of RES that slowly extends to a steady 1.2-1.5G.

xml.sax etree.ElementTree parser

This code uses the event based xml.sax parser to create an etree.ElementTree object for pyFF, inside pyFF. As of the moment of writing, pyFF refuses validate the result, but it produces correct metadata?
The parsing could be brought outside of pyFF to create a dictionary type of object to be read and parsed as a metadata representation to create the ElementTree object in pyFF instead of parsing XML.

```
https://docs.python.org/3/library/xml.sax.reader.html
import xml.sax
class XML(xml.sax.handler.ContentHandler):
  def __init__(self):
    self.current = etree.Element("root")
```

```
    self.nsmap = { 'xml': 'http://www.w3.org/XML/1998/namespace'}
    self.buffer = ''

  def startElement(self, name, attrs):
    attributes = {}
    for key, value in attrs.items():
        key = key.split(':')
        if len(key) == 2:
            if key[0] == 'xmlns':
                self.nsmap[key[-1]] = value
            else:
                    attributes[f"{{{ self.nsmap.get(key[0], key[0]) }}}{
key[-1] }"] = value
        elif value:
            attributes[key[-1]] = value

    name = name.split(':')
    if len(name) == 2:
        name = f"{{{ self.nsmap.get(name[0], name[0]) }}}{ name[-1] }"
    else:
        name = name[-1]
        self.current  =  etree.SubElement(self.current,  name,  attributes,
nsmap=self.nsmap)

  def endElement(self, name):
    self.current.text = self.buffer
    self.current.tail = "\n"
    self.current = self.current.getparent()
    self.buffer = ''

  def characters(self, data):
    d = data.strip()
    if d:
        self.buffer += d


def parse_xml(io, base_url=None):
    parser = xml.sax.make_parser()
    handler = XML()
    parser.setContentHandler(handler)
```

```
    parser.parse(io)
    return etree.ElementTree(handler.current[0])
```

Using xml.sax parser pyFF's gunicorn starts out using ~800Mb of RES that slowly extends to a steady 1.2-1.5G.

Run pyFF in a uwsgi server

```
#!/bin/sh

bin/uwsgi \
    --http 127.0.0.1:8080 \
    --module pyff.wsgi \
    --callable app \
    --enable-threads \
    --env PYFF_PIPELINE=edugain.yaml \
    --env PYFF_WORKER_POOL_SIZE=10 \
    --env PYFF_UPDATE_FREQUENCY=60 \
    --env PYFF_LOGGING=pyFFplus/examples/debug.ini
```

Long-run test reveals comparable memory usage as gunicorn, but there seem to be more knobs to play with.

One of the things we can do against boundless growth of uwsgi is the use of --reload-on-rss <limit>, this kills any worker that exceeds the RSS limit, but results in an empty metadata reply, which is unwanted behaviour. If however, we also supply --lazy, the app is loaded in the worker(s) and the (re)start of each worker then also triggers the reload of metadata. This could be a compromise if the VM is less cpu bound than memory?

Empty Metadata set while refreshing

It turns out pyFF returns an empty metadata set while refreshing, which is unwanted behaviour. The following code, inserted just before the final return in .api#process_handler inspects the validity of the Resource

metadata. Having a loadbalancer inspect pyFF and temporarily evicting the server from pool if it receives a 500 could create a stable service.

```
def process_handler():
...

# Only return request if md is valid?
valid = True
log.debug(f"Resource walk")
for child in request.registry.md.rm.walk():
    log.debug(f"Resource {child.url}")
    valid = valid and child.is_valid()

if len(request.registry.md.rm) == 0 or not valid:
    log.debug(f"Resource not valid")
     # 500: The server has either erred or is incapable of performing the requested operation.
    raise exc.exception_response(500)
else:
    log.debug(f"Resource valid")

return response
```

Performance-test branch

Incorporated the "store.py" changes in this branch https://github.com/IdentityPython/pyFF/compare/preformance-tests to see how that would change the memory consumption of pyFF, but it didn't change much. It ends up using ~1.8G of RES after several hours of continuously (60s) refreshing the edugain metadata feed.

The changes try to store entities as their serialized (tostring) version of the metadata, and re-parse it on demand. The idea being that we don't need to keep track of the whole parsed tree, but just the serialized entities.

Parked

https://tech.buzzfeed.com/finding-and-fixing-memory-leaks-in-python-413ce4266e7d

(never got around to go this deep into python debugging)

Size limitations

We created controlled mock metadata sets containing multitudes of edugain metadata (e.g. 5k, 10k, 20k and 100k entities) to see how pyFF would cope with that amount of entities and metadata.

The mock metadata is available here:
https://gitlab.geant.org/TI_Incubator/mockup-metadata

Conclusions

There is no real theoretical reason for the XML processing to be this way (apart from XML Dsig Verification). But it would require a total rewrite to make improvements (for pyFF at least: get rid of elementTree)
XML DSig is not helping since it requires c14n and this needs to be done recursively.

Overall, we did not manage to achieve any significant improvements, but we did confirm that the problem is real and we did create a test suite and test most of the popular SAML stacks.