Pyff+ Optimizations and mock metadata

Mihály Héder

pyFF+ original aims

- **Context:**
  - When processing the eduGAIN metadata, pyFF's
    - memory usage balloons to the gigabytes,
    - hereby inflicting some extra cost when running
    - same with other SAML stacks

pyFF+ original aims

- **Original Goals**
  - An investigation of memory consumption
  - Finding potential memory hot spots
  - If there are hot spots, solutions are planned and implemented
  - pyFF is split into multiple modules to externalize the metadata processing
  - New implementation is committed to the official repository

TRUST & IDENTITY INCUBATOR

pyFF+ hacking experiments

GÉANT

Experiments made

- Memory profiling
  - **heapy way**: 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
  - **top/htop way:**
    - following RES in top or htop for a long-running pyFF/gunicorn process, that has a 60s refresh interval
- all while loading edugain.xml

Experiments made

- Un/Pickling etree.ElementTree object
- Idea: process once, then distribute
  - externally parsed etree.ElementTree objects can be pickled (serialized) to be consumed later in pyFF, without the need to parse.

Experiments made

- Rewriting to SAX parsing from DOM
- The idea is to switch from recursive processing to sequential

- This code uses the event based xml.sax parser to create an etree.ElementTree object for pyFF, inside pyFF.
- 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.

```
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])
```

Experiments made

- Run pyFF in a uwsgi server
  - 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?

```
#!/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
```

Experiments made

- 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
```

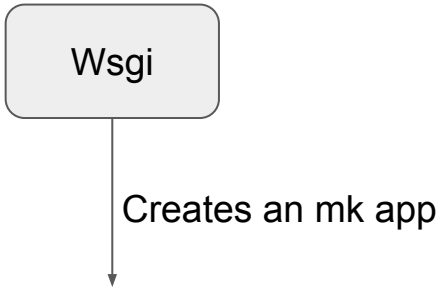
Experiments made

Unpacking pyFF+ resource loading model

Wsgi

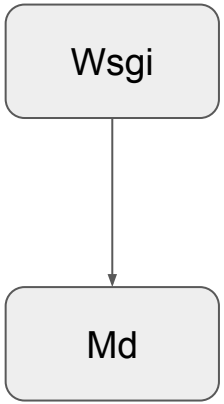
Experiments made

Unpacking pyFF+ resource loading model



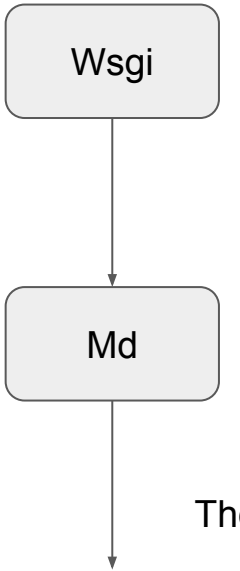
Experiments made

Unpacking pyFF+ resource loading model



Experiments made

Unpacking pyFF+ resource loading model

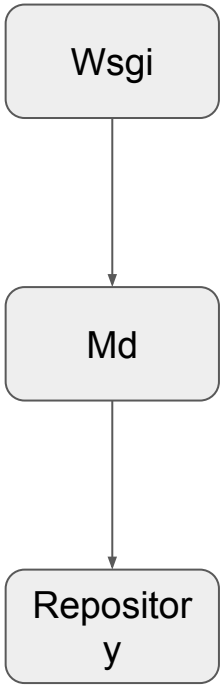


Then a repository is created

Check /src/pyff/md.py line 34

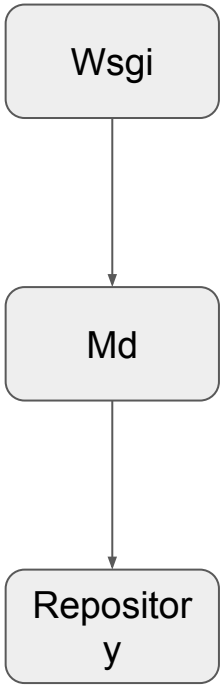
Experiments made

Unpacking pyFF+ resource loading model



Experiments made

Unpacking pyFF+ resource loading model

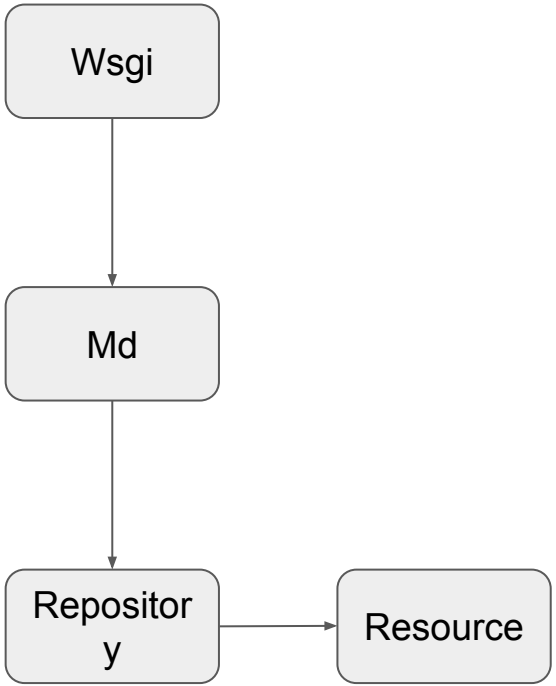


At this point a resource is created and a watcher

Check src/pyff/repo.py line 13

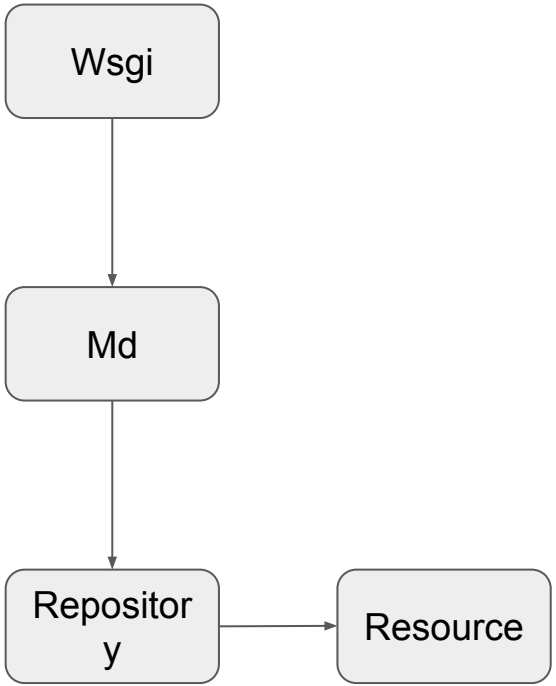
Experiments made

Unpacking pyFF+ resource loading model



Experiments made

Unpacking pyFF+ resource loading model

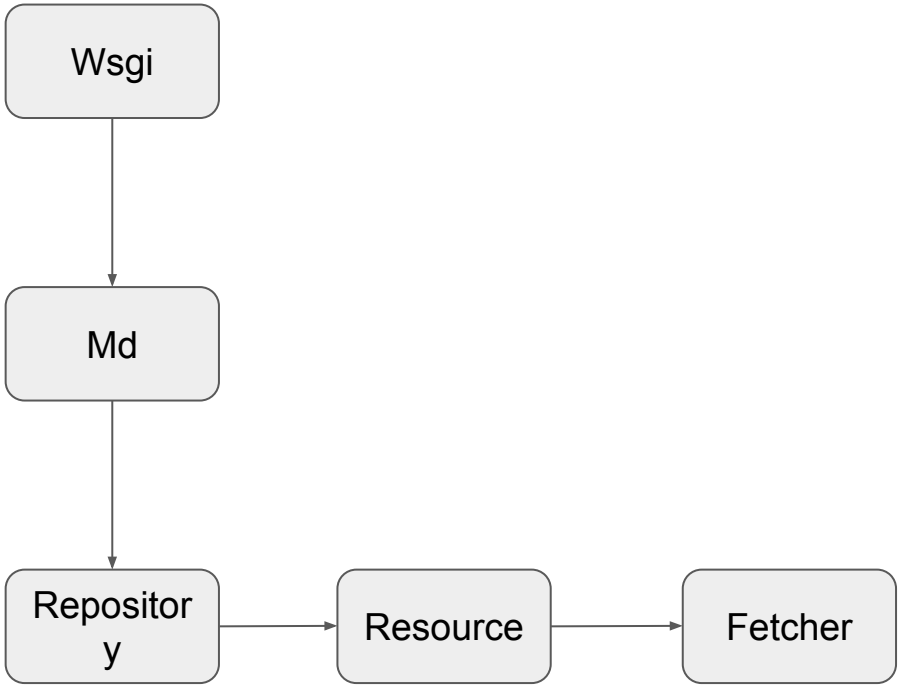


Next is the creation of the Fetcher

src/pyff/resource.py line 18, 33

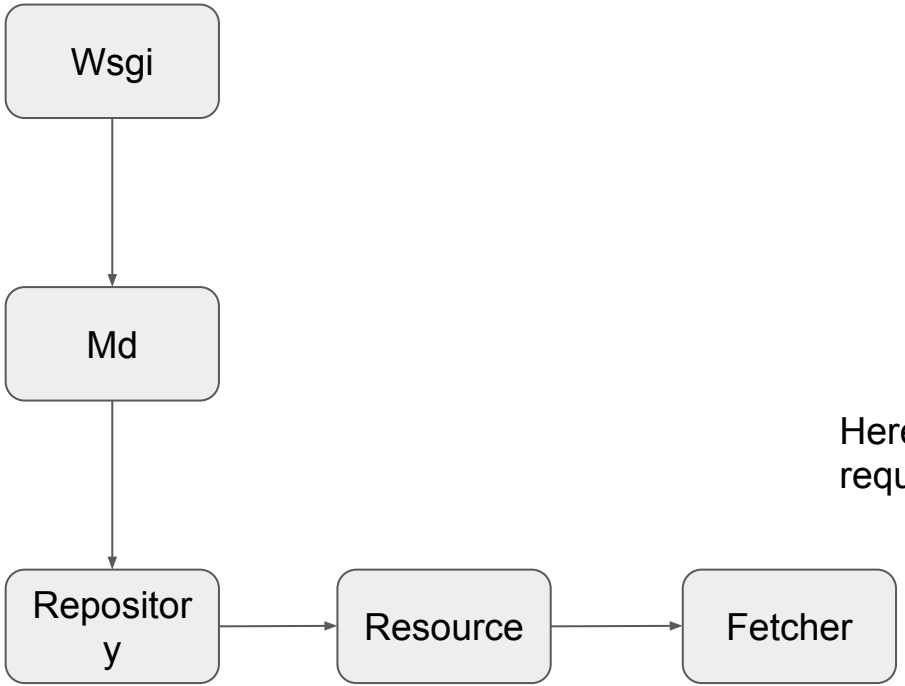
Experiments made

Unpacking pyFF+ resource loading model



Experiments made

Unpacking pyFF+ resource loading model

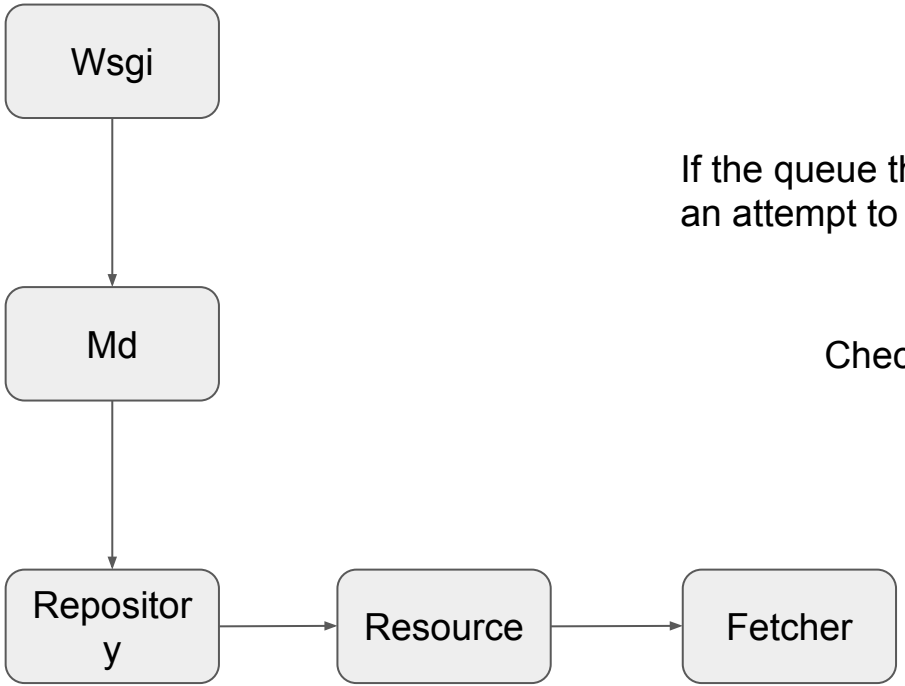


Here is the creation of the important request,response

Check init of the class Fetch line 21 at src/pyff/fetch.py

Experiments made

Unpacking pyFF+ resource loading model

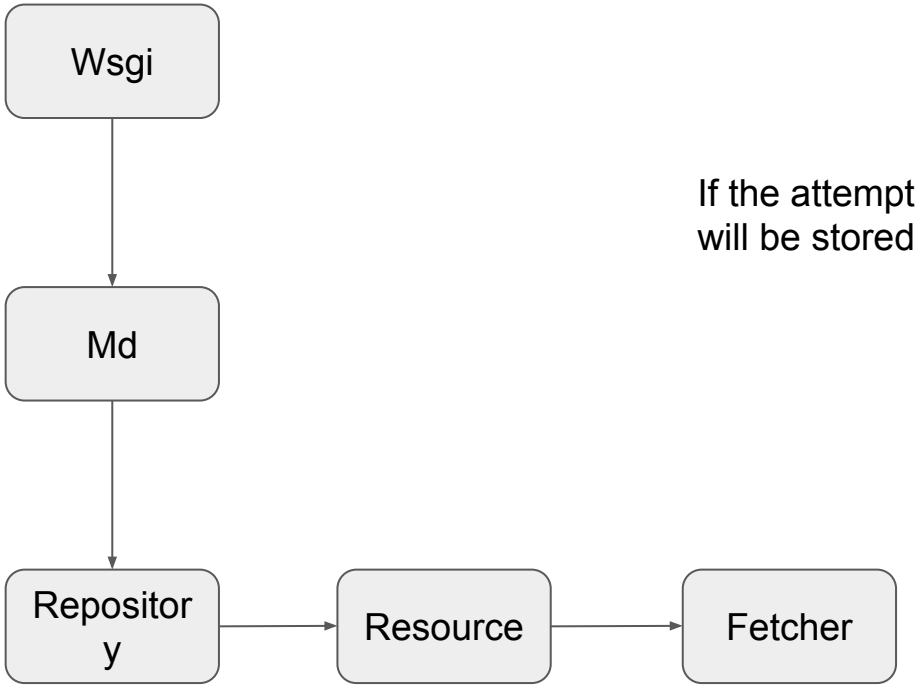


If the queue that is being tracked is not empty, there will be an attempt to fetch the url from the queue

Check line 36 of fetch.py

Experiments made

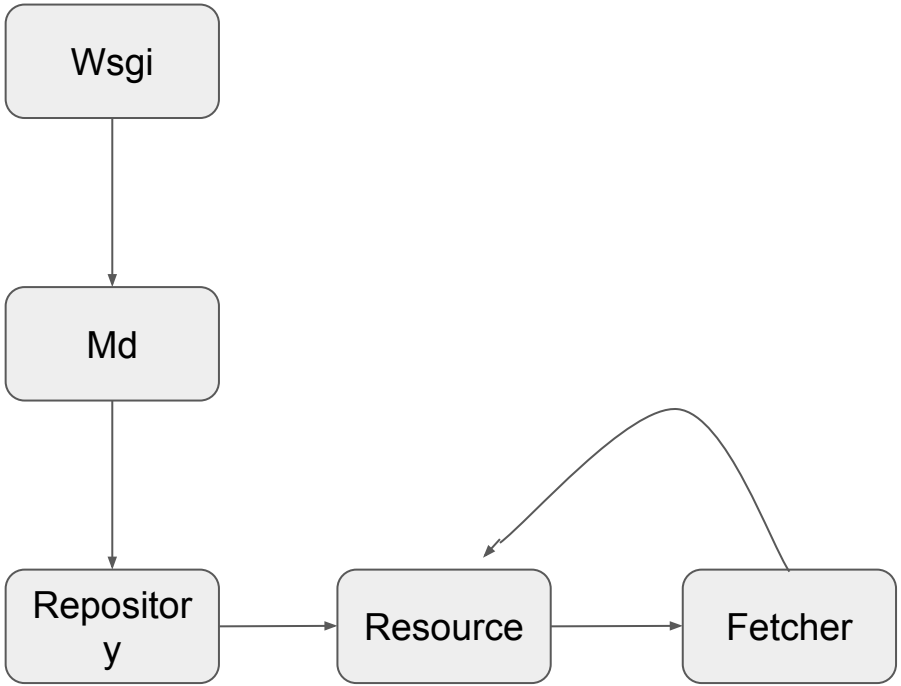
Unpacking pyFF+ resource loading model



If the attempt is successful the fetched data will be stored in the Resource

Experiments made

Unpacking pyFF+ resource loading model



Other actions taken

- Interview with Leif
  - central theme: python XML processing is a lackluster
  - Ways forward
    - pyff (for batch only)+thiss.io usage (eduTEAMs way)
    - reimplementation (GO?)

TRUST & IDENTITY INCUBATOR

Mock Metadata Experiments

GÉANT

Mock metadata

- Idea:
  - generate big metadata files that resemble eduGAIN XML but larger
  - sign
- Details
  - 10,15,30,50,100k entities
  - 300k names, 300k email, 300k domains, 300k certificates generated
  - We do idp/sps (no AA)

Alternative outcome: Mock metadata test suite

Test set	Entities	XML flesize (MB)	Size as compare to eduGAIN
10k	10.000	48	1.38
15k	15.000	74.1	2.07
30k	30.000	148	3.76
50k	50.000	242	6.9
100k	100.000	484	13.8

pyFF

Mock Size	Memory (% of 4 Gb)	Time (s)	Errors, Anomalies, Notes
10k	12	9	
15k	22	12	
30k	40	18	
50k	98	30	
100k	-	-	The process exits code=killed. Out of memory

Shibboleth SP

Mock Size	Memory (% of 4 Gb)	Time (s)	Errors, Anomalies, Notes
10k	9	4	
15k	22	7	
30k	44	12	
50k	72	17	
100k	-	-	“Cannot allocate memory”

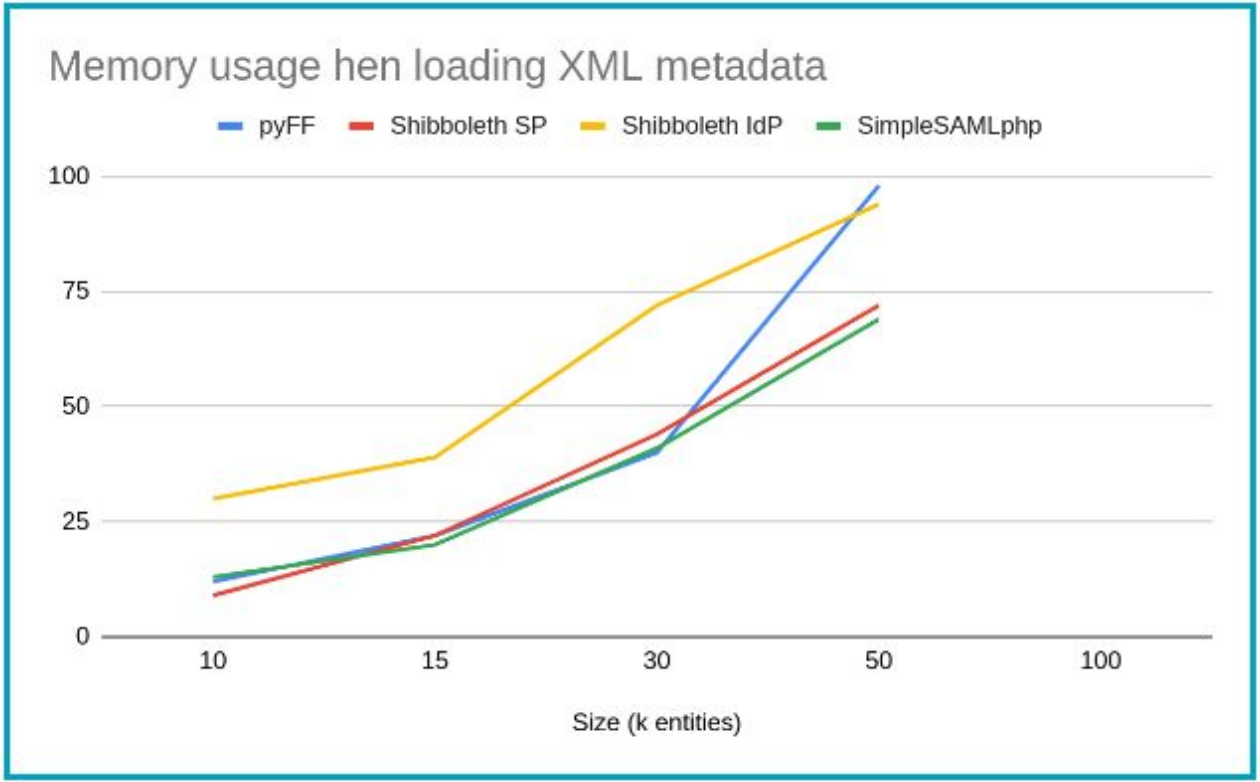
Shibboleth IdP

Mock Size	Memory (% of 4 Gb)	Time (s)	Errors, Anomalies, Notes
10k	30	18	
15k	39	25	
30k	72	39	
50k	94	66	Increased Java heapsize
100k	-	-	"OpenJDK 64-Bit Server VM warning: INFO: os::commit_memory(0x00000007ee900000, 115343360, 0) failed; error='Not enough space' (errno=12)"

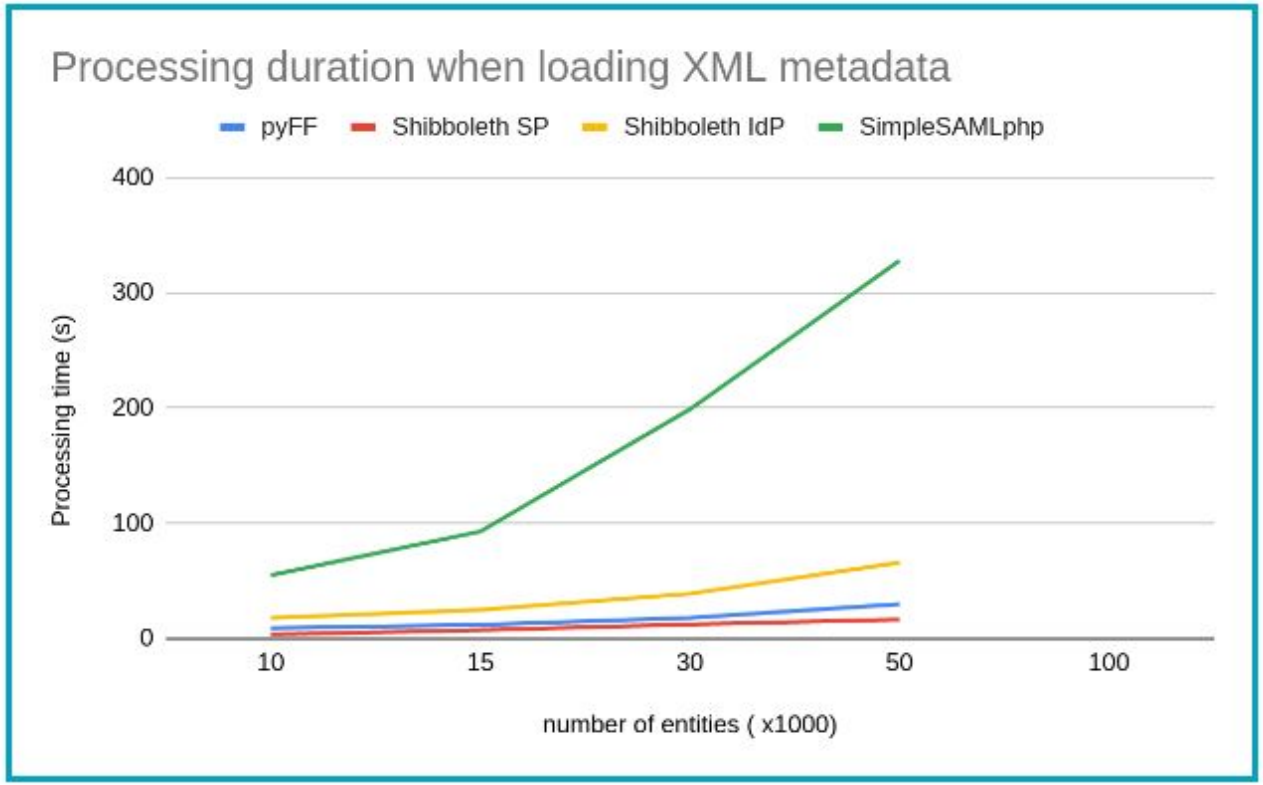
simpleSAMLphp

Mock Size	Memory (% of 4 Gb)	Time (s)	Errors, Anomalies, Notes
10k	13	55	
15k	20	93	
30k	41	99	
50k	69	328	
100k	-	-	mmap() failed: [12] Cannot allocate memory mmap() failed: [12] Cannot allocate memory PHP Fatal error: Out of memory (allocated 2141122560) (tried to allocate 8192 bytes) in /var/simplesamlphp/lib/SimpleSAML/Metadata/SAMLParser.php on line 166

Comparison chart: RAM



Comparison chart: Time



Mock XML conclusion & possible road ahead

- Things are not great
- future investigations: what is the issue
  - no. of entities?
  - no. of XML elements?
  - some combination?
- mock XML should resemble the real thing even more
  - entity attributes
  - extensions
  - etc.
- Report is being wrapped up

Overall conclusion

- No real theoretical reason for the XML processing to be this way (apart from 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
  - part of the XML
  - because of the InfoSet
- Needs to be on the radar of everyone

Thank you!

Questions?