Lessons learned maintaining the oVirt Python SDK

Juan Hernández

The oVirt project.
Objective

Share some lessons that we learned while developing and maintaining the Python SDK of the oVirt project, and how we used what we learned to improve things with version 4 of the SDK.
Introduction
What is oVirt?

- A virtualization platform.
- Manages virtual machines, storage and networks.
- Easy to use web interface, for humans.
- Comprehensive REST API, for other systems.
What is the oVirt Python SDK?

A Python library that simplifies access to the oVirt API, hiding the details of the HTTP protocol and the XML format.
Without the SDK:

Send this:

```
GET /ovirt-engine/api/vms?search=name%3Dmyvm%2A&max=10
Version: 4
Accept: text/xml
Authorization: Bearer nTVBigh2keLtgChmCiL
```

Receive this:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: 1023

<vms>
  <vm id="123" href="/ovirt-engine/api/vms/123">
    <name>myvm</name>
    <memory>1073741824</memory>
  </vm>
  ...
</vms>
```
With the SDK:

```python
import ovirtsdk4 as sdk

# Create the connection to the server:
connection = sdk.Connection(
    url='https://engine40.example.com/ovirt-engine/api',
    username='admin@internal',
    password='...',
    ca_file='ca.pem'
)

# Get the service that manages the collection of virtual machines:
vms_service = connection.system_service().vms_service()

# Send the request and process the response:
vms = vms_service.list(search='name=myvm*', max=10)

# Print the details:
for vm in vms:
    print("%s: %d" % (vm.name, vm.memory))

# Close the connection:
connection.close()
```
The lessons
The lessons

Lesson 1: Need an easy to use specification.
Lesson 2: Don't expose code that you can't control.
Lesson 3: Simple code performs better than smart code.
Lesson 4: Latency kills performance.
Lesson 1:
Need an easy to use specification.
Lesson 1: What was the situation?

- In version 3 of the API we used two separate descriptions of the API: the XML schema and the RSDL.

- The XML schema defined the types.

- The RSDL defined the operations.

- The type classes were generated from the XML schema using `generateDS.py`.

- The code implementing the operations code (the brokers) was generated from the RSDL.

- Releases of the SDK were strongly tied to releases of the server.
Lesson 1: Why did this need to be improved?

- The code generator was complex and maintaining it was expensive.
- Many parts of the code generator were repeated for the existing Java SDK (and the Ruby SDK, and maybe, in the future, the Go SDK).
Lesson 1: What was changed?

- In version 4 of the server we moved the specification of the API to a separate *model* project.

- The same specification defines both the types and the services, using the same language.

- The specification of the API has its own independent release cycle.

- The dependency between the server and the SDKs was removed. Both the server and the SDKs depend on the specification instead.

- We introduced a new *metamodel* project that simplifies usage of the specification and contains the parts of the code generator that can be used by multiple SDKs.
Lesson 2:
Don't expose code that you can't control.
Lesson 2: What was the situation?

- Version 3 of the SDK exposed two main kinds of classes: params and brokers.
- The **params** classes were generated from the XML schema using `generateDS.py`.
- The code generated by `generateDS.py` was modified by the code generator.
- The **broker** classes extended the param classes, and added the operations, based on the RSDL.
Lesson 2: Why did this need to be improved?

- We couldn't easily upgrade the version of `generateDS.py` because it can break backwards compatibility, or just make our modifications fail.

- Our XML schema, when processed by `generateDS.py` resulted in code that is less Pythonic than it could be.

- The code generated by `generateDS.py` didn't support Python 3.

- The use of `generateDS.py` implies the use of XML, making it difficult to use JSON, should we want to use it in the future.
Lesson 2: What was changed?

- In version 4 of the SDK we replaced `generateDS.py` with a code generator that we control and that is driven by the specification of the API.
- The generated code supports Python 2 and Python 3 out of the box.
- The generated code is more Pythonic.
- The code that implements the types and the code that converts from/to XML is independent.
Lesson 3:
Simple code performs better than smart code.
Lesson 3: What was the situation?

- Version 3 of the SDK used `generateDS.py` to parse XML, which in turn used a DOM based approach.

- Version 3 of the SDK used smart code to be able to use the classes generated by `generateDS.py` as the base for `broker` classes.

- Python `inspect.getmembers`, `__dict__` and `__name__` were heavily used.
Lesson 3: Why did this need to be improved?

- The complete XML parsing solution performed bad for large documents, in terms of memory and specially in terms of CPU usage.
Lesson 3: What was changed?

- In version 4 of the SDK we replaced `generateDS.py` and the DOM based approach with `libxml2` and streaming XML parsing.

- New `XmlReader` and `XmlWriter` classes were implemented on top of `libxml2`, in a module written in C.

- The type classes are simple, and completely independent of the XML reading/writing code.

- The XML reading classes are also simple.

- The XML writing classes are even simpler.

- The complete XML reading and writing solution performs better.
Lesson 3: How better?

Read 4000 virtual machines with version 3 of the SDK:

```bash
$ /bin/time ./v3-read-vms.py
765.81 user 0.21 system 12:47.75 elapsed 99%CPU (0 avgtext+0 avgdata 528008 maxresident)k
0 inputs+0 outputs (0 major+140566 minor) pagefaults 0 swaps
```

With version 4 of the SDK:

```bash
$ /bin/time ./v4-read-vms.py
4.48 user 0.14 system 0:04.64 elapsed 99%CPU (0 avgtext+0 avgdata 268604 maxresident)k
0 inputs+0 outputs (0 major+108110 minor) pagefaults 0 swaps
```

That is approx 170 times faster.

Since the issue was detected, version 3 of the SDK has also been simplified and performs much better now. But version 4 still consumes half the time and half the amount of memory.
Lesson 4: Latency kills performance.
Lesson 4: What was the situation?

- The typical usage of the SDK was completely **sequential**.

- Taking advantage of multiple connections was only possible using multiple threads and/or multiple processes.
Lesson 4: Why did this need to be improved?

- In environments with high network latency performance of sequential access to the API was unacceptable.
Lesson 4: What was changed?

- In version 4 of the SDK we introduced support for multiple connections, HTTP pipelining and asynchronous requests.

- The application performs one or multiple requests specifying the \texttt{wait=False} parameter (the default is True):

  ```python
  vm_id = ..
  vm_service = vms_service.vm_service(vm_id)
  vm_future = vm_service.get(wait=False)
  ```

- The request is sent to the underlying library (libcurl, via pycurl), but nothing is sent through the network yet.

- The application receives a \textit{future} object instead of the requested object.
Lesson 4: What was changed? (II)

- The application may do other things, typically send other related requests, for related objects:

```python
nics_service = vm_service.nics_service()
nics_future = nics_service.list(wait=False)
```

- Eventually the application waits for the requested objects, calling the `wait` method of the `Future` class:

```python
vm = vm_future.wait()
nics = nics_future.wait()
```

- The `wait` method of the `Future` class calls the `wait` method of the `Connection` class. This waits till response for that request, and maybe for others, is received.
Lesson 4: What was changed? (III)

- No need for multiple processes or threads, as all processing happens when calling the `wait` method.
Lesson 4: How better?

In an environment with 200 ms of latency retrieve 100 virtual machines and their NICs sequentially:

```bash
$ /bin/time ./v4-vms-with-nics.py
20.84 user 2.65 system 0:25.11 elapsed 93% CPU (0 avgtext+0 avgdata 37748 maxresident)k
0 inputs+0 outputs (0 major+8000 minor) pagefaults 0 swaps
```

Same, but enabling multiple connections and pipelining:

```bash
$ /bin/time ./v4-vms-with-nics-async.py
3.22 user 0.46 system 0:05.47 elapsed 67% CPU (0 avgtext+0 avgdata 40880 maxresident)k
0 inputs+0 outputs (0 major+9188 minor) pagefaults 0 swaps
```
Thanks!