PyOrthanc

A Python client for the Orthanc REST API

Gabriel Couture[1] Yannick Lemaréchal[1] Philippe Després[1]

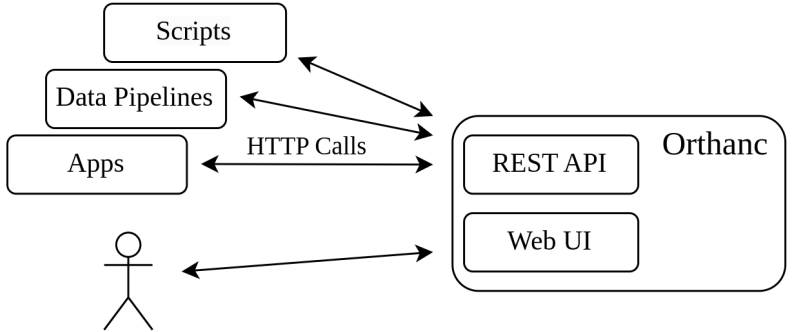
[1]Centre de recherche de l'Institut universitaire de cardiologie et de pneumologie de Québec-Université Laval

Orthanc Conference 2023, 1 October 2023

Hello, today I will present the PyOrthanc library, which is a Python client that interacts with the Orthanc REST API.

Introduction

- Orthanc is used around the world in both clinical and research contexts
- Orthanc stands out by its REST API
  - Allows users to **programmatically** interact with Orthanc
  - Can be difficult or simply tedious



Introduction
Orthanc is used around the world in both clinical and research contexts
Orthanc stands out by its REST API
Allows users to programmatically interact with Orthanc
Can be difficult or simply tedious
Scripts
Data Pipelines
Apps
HTTP Calls
REST API
Orthanc
Web UI

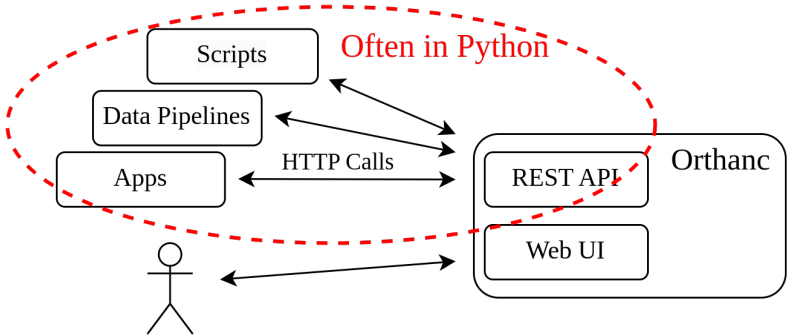
Orthanc is used around the world in both clinical and research contexts.

Orthanc stands out among the open-source DICOM servers with its REST API, which has many interesting functionalities and allows users to interact programmatically with it.

However using the REST API may be difficult for people without a programming background or it can involve a lot of boilerplate.

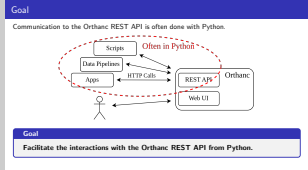
Goal

Communication to the Orthanc REST API is often done with Python.



Goal

Facilitate the interactions with the Orthanc REST API from Python.

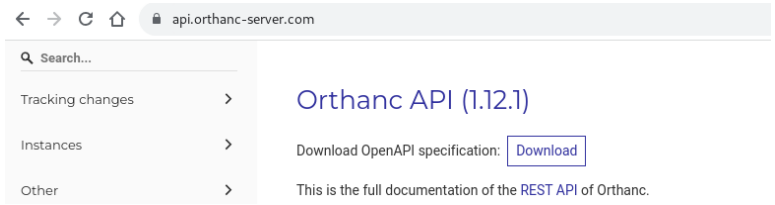


A very common way to interact with Orthanc is from Python. When I started PyOrthanc, there was not really any complete Python library to interact with the REST API.

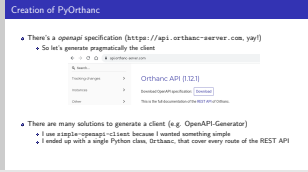
I decided to create one and my goal was to facilitate the interaction with the Orthanc REST API from Python, for both users who are not used to REST APIs and experienced users who simply want a straightforward way to do things.

Creation of PyOrthanc

- There's a *openapi* specification (`https://api.orthanc-server.com`, yay!)
  - So let's generate pragmatically the client



- There are many solutions to generate a client (e.g. OpenAPI-Generator)
  - I use `simple-openapi-client` because I wanted something simple
  - I ended up with a single Python class, `Orthanc`, that cover every route of the REST API



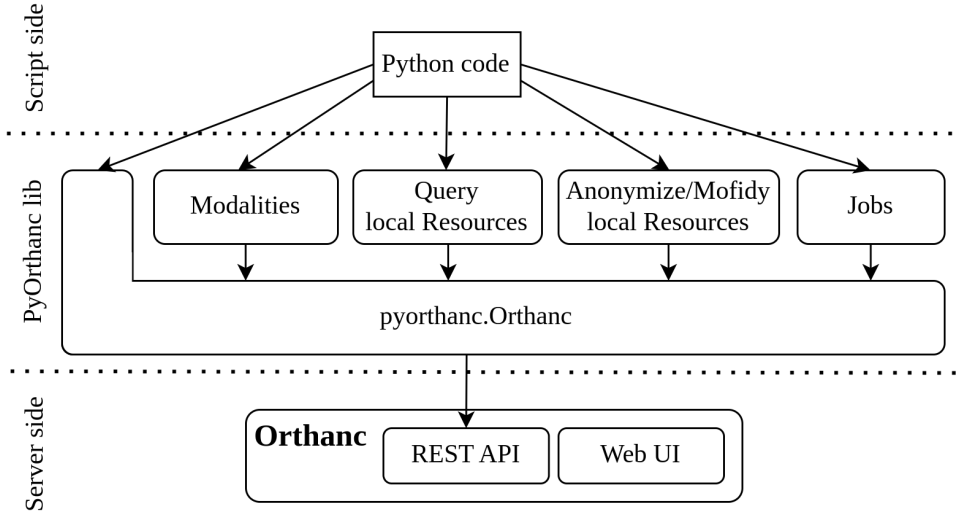
In the first version, everything was done by hand. However, at some point, the Orthanc team released an openAPI specification with which I was able to programmatically generate a Client.

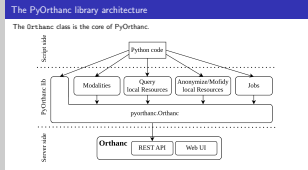
There are many solutions to do that. I use the simple-openapi-client which simply generates a Python class.

I ended up with the Python class Orthanc, which cover every route of the REST API.

The PyOrthanc library architecture

The `Orthanc` class is the core of PyOrthanc.





This is the basic architecture of the library. You can see that the pyorthanc.Orthanc class is the core of the library.

Pretty much all utility functions or classes are built onto it.

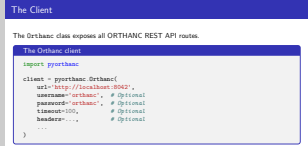
The Client

The `Orthanc` class exposes all ORTHANC REST API routes.

The Orthanc client

```python
import pyorthanc

client = pyorthanc.Orthanc(
    url='http://localhost:8042',
    username='orthanc',  # Optional
    password='orthanc',  # Optional
    timeout=100,         # Optional
    headers=...,         # Optional
    ...
)
```

Creating a client instance is as simple as this.

You have to provide the URL, username and password, and if you like additional HTTP-related parameters such as the timeout.

The Client

Then we can use the client as we want

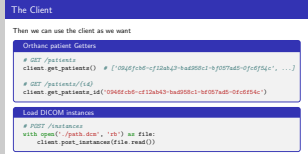
Orthanc patient Getters

```python
# GET /patients
client.get_patients()  # ['0946fcb6-cf12ab43-bad958c1-bf057ad5-0fc6f54c', ...]

# GET /patients/{id}
client.get_patients_id('0946fcb6-cf12ab43-bad958c1-bf057ad5-0fc6f54c')
```

Load DICOM instances

```python
# POST /instances
with open('./path.dcm', 'rb') as file:
    client.post_instances(file.read())
```

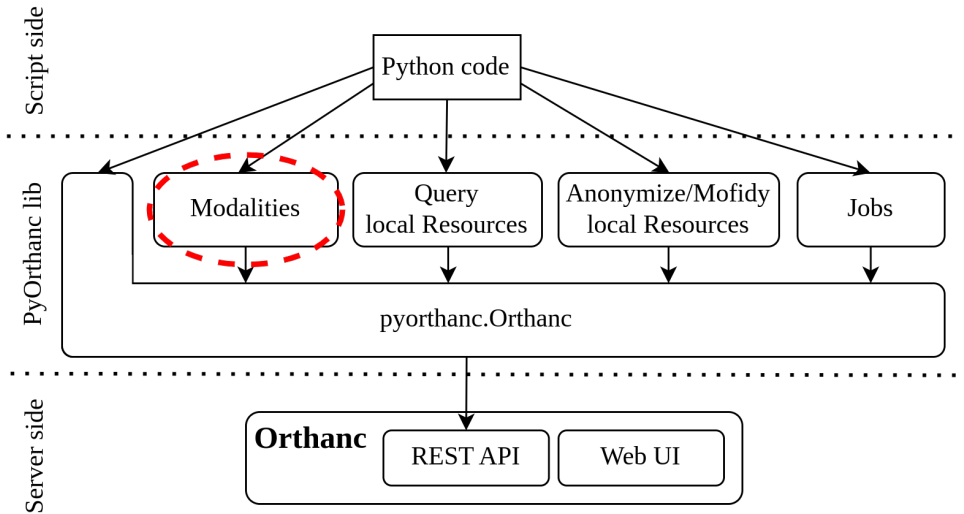



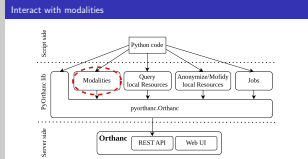
This is a very basic showcase of few methods.

As you can see, the client methods that start with "get" make an HTTP GET call, and it is the same thing with the post, delete and put methods.

Here I retrieve the Orthanc Patient IDs and then I retrieve the information of a specific patient with HTTP GET calls. I also load a new instance into Orthanc with HTTP POST call to /instances.

Interact with modalities



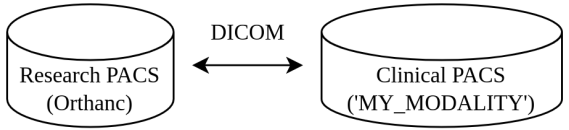


A very common use case for us when using Orthanc is to interact with other DICOM servers. So it was one of the first things that we implemented.

Modalities

Find the modalities

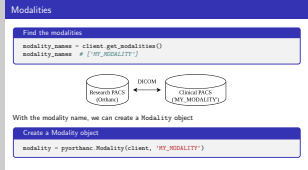
```
modality_names = client.get_modalities()
modality_names  # ['MY_MODALITY']
```



With the modality name, we can create a `Modality` object

Create a Modality object

```
modality = pyorthanc.Modality(client, 'MY_MODALITY')
```

You can retrieve the connected modalities with the client, which are essentially other DICOM servers.

For example, my research PACS which is an Orthanc server is connected to the MY MODALITY, which is a clinical PACS.

With the Modality name, we can create a Modality object that facilitates the interaction between Orthanc and the modality.

C-ECHO and C-FIND

Common workflow to interact with a modality

C-ECHO

```
modality.echo()  # True if success, False if not.
```

C-FIND

```
modality.query(
    {'Level': 'Study', 'Query': {'StudyDescription': 'Something*'}}
)
```

C-MOVE

```
modality.move('<query_id>', {'TargetAet': 'MY_PACS'})
```

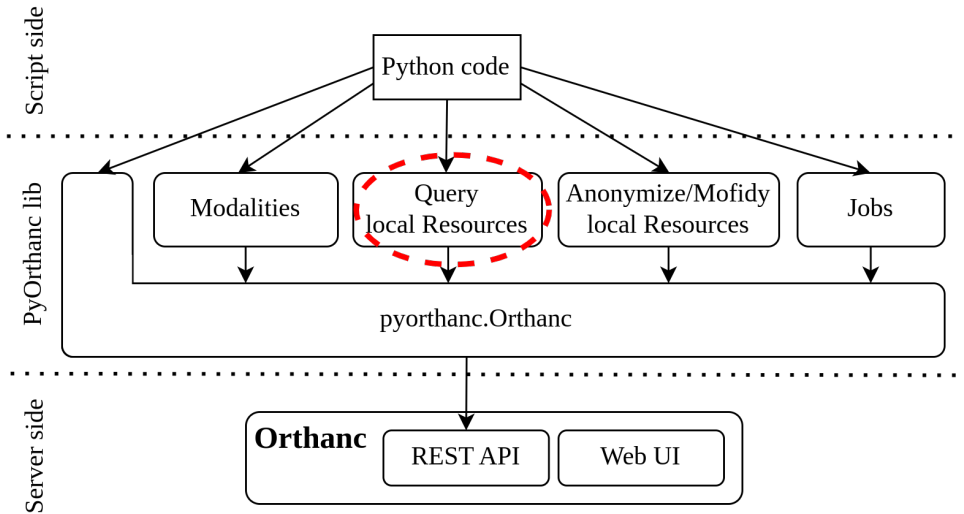

The modality object allows you to perform the common DICOM operations, such as C-Echo, C-Find and C-Move.

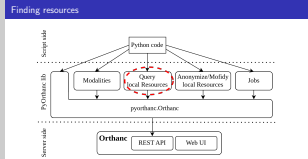
The C-Echo will return True or False depending on the success of the connection between Orthanc and the modality.

The C-Find will ask Orthanc to make a query on the Modality. Then Orthanc will have the query result.

The C-Move will ask Orthanc to ask the Modality to send a copy of the data that fits the query to a target modality, which could be the Orthanc server itself or another PACS.

Finding resources





We also add some utility functions and classes to query the local data on Orthanc.

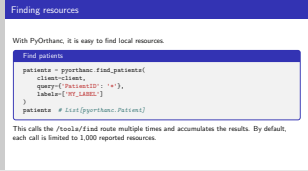
Finding resources

With PyOrthanc, it is easy to find local resources.

Find patients

```
patients = pyorthanc.find_patients(
    client=client,
    query={'PatientID': '*'},
    labels=['MY_LABEL']
)
patients  # List[pyorthanc.Patient]
```

This calls the `/tools/find` route multiple times and accumulates the results. By default, each call is limited to 1,000 reported resources.



For example, you can easily find patients with the find_patients function.

You simply have to provide a query and/or a list of labels that fit the patients that you are looking for.

A great thing about this function is that, by default, it calls Orthanc many times to accumulate the results. It limits each call to 1,000 reported resources. This is very handy for large Orthanc servers. If you don't want to do that, you can always use a lower-level function called query_orthanc.

Finding resources

Each level of resources can be queried.

Find specific resources

```
patients = pyorthanc.find_patients(...)
studies = pyorthanc.find_studies(...)
series = pyorthanc.find_series(...)
instances = pyorthanc.find_instances(...)
```


Finding resources
Each level of resources can be queried.
Find specific resources
patients = pyorthanc.find_patients(...)
studies = pyorthanc.find_studies(...)
series = pyorthanc.find_series(...)
instances = pyorthanc.find_instances(...)

In the same spirit, every resource level has its own function.

Access resource data

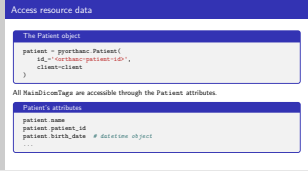
The Patient object

```
patient = pyorthanc.Patient(
    id_='<orthanc-patient-id>',
    client=client
)
```

All `MainDicomTags` are accessible through the `Patient` attributes.

Patient's attributes

```
patient.name
patient.patient_id
patient.birth_date  # datetime object
...
```

Once you have your resources in pyorthanc objects, it is very easy to retrieve their metadata.

For example, the patient object makes every MainDicomTags accessible as attributes that return Python objects, such as a datetime object for the birth_date.

Each access to the attributes will make an HTTP call to Orthanc. There is also a way to query Orthanc a single time when creating a patient and keep a local version of the data, with the risk that it will not be up to date.

Access resource data

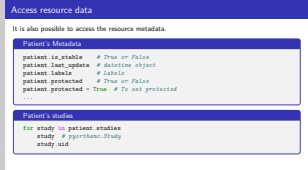
It is also possible to access the resource metadata.

Patient's Metadata

```
patient.is_stable     # True or False
patient.last_update   # datetime object
patient.labels        # Labels
patient.protected     # True or False
patient.protected = True  # To set protected
...
```

Patient's studies

```
for study in patient.studies
    study  # pyorthanc.Study
    study.uid
```

Metadata not related to DICOM are also accessible through attributes, such as the stability or update status, the labels or if the patient is protected.

The patient studies are also easily accessible.

Note that there are resource classes for every resource level, which are the patient, study, series and instance.

Patient zip and pydicom Dataset

To download the patient zip file that contains all patient's DICOM files.

Download zip

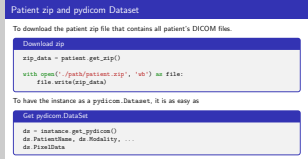
```
zip_data = patient.get_zip()

with open('./path/patient.zip', 'wb') as file:
    file.write(zip_data)
```

To have the instance as a `pydicom.Dataset`, it is as easy as

Get pydicom.DataSet

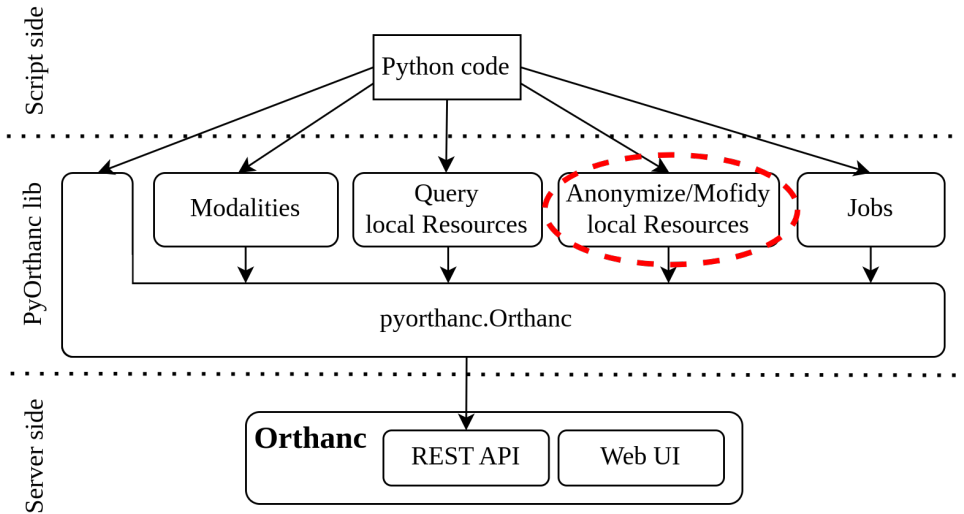
```
ds = instance.get_pydicom()
ds.PatientName, ds.Modality, ...
ds.PixelData
```

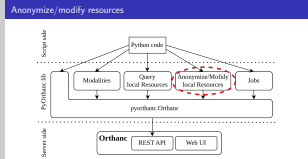



The resource objects also have handy methods. For example, to retrieve all the data from one patient, you can retrieve it as a zip file, and then save it locally.

Another interesting example is with the Instance object, with which you can retrieve and serialize a pydicom Dataset in a single method call.

Anonymize/modify resources





PyOrthanc allows you to use the Orthanc functionalities to anonymize and modify data, which are very interesting functionalities of Orthanc.

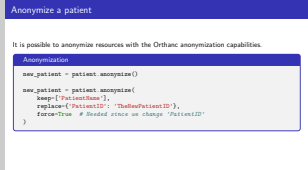
Anonymize a patient

It is possible to anonymize resources with the Orthanc anonymization capabilities.

Anonymization

```
new_patient = patient.anonymize()

new_patient = patient.anonymize(
    keep=['PatientName'],
    replace={'PatientID': 'TheNewPatientID'},
    force=True  # Needed since we change 'PatientID'
)
```

For example, here we can create a new anonymized patient with a single call that will return a new patient. The method also gives control over the anonymization options.

Modify a patient

We can modify resources as well.

Modification

```python
patient.modify(replace={'PatientName': 'new-name'})
patient.name  # 'new-name'

# This creates a new patient since we change the PatientID
new_patient = patient.modify(
    replace={'PatientID': 'TheNewPatientID'},
    force=True  # Needed since we change 'PatientID'
)
```


It is pretty much the same thing for the Modify functionality.

Here I modify a patient in place, and I also modify a PatientID, which creates a new patient instance.

Other Resources

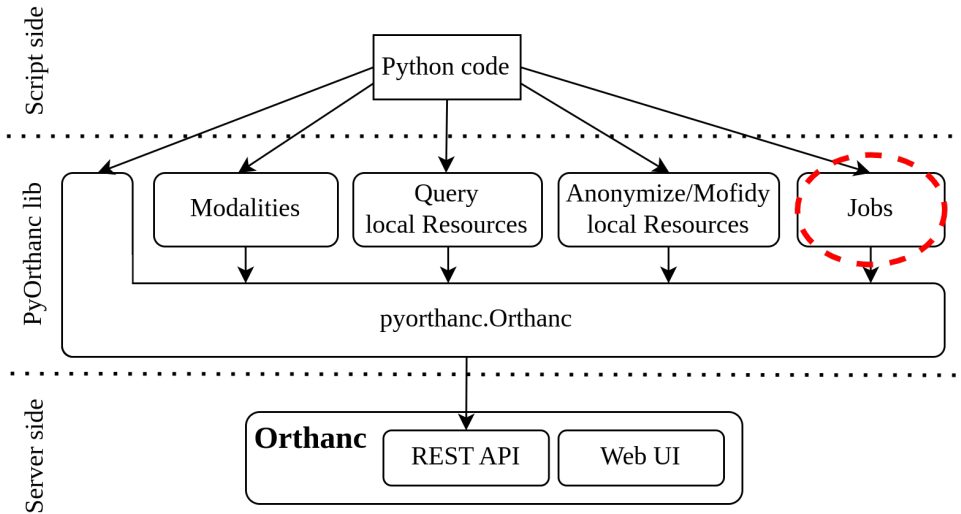
Similar methods are available for the other resources.

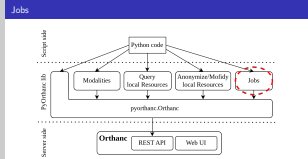
Resources methods

```
study.anonymize(), series.anonymize(), instance.anonymize()
study.modify(), series.modify(), instance.modify()
```


Note that the other resource levels has the same methods.

Jobs





Another important aspect of Orthanc is the jobs. Many long processes are handled with this.

PyOrthanc also provides a utility class to deal with that.

Jobs

The Job objects

```
job = pyorthanc.Job('<job_id>', client)

job.state  # 'Failure', 'Running', 'Pending', ...
```

Wait until completion

```
job.wait_until_completion()
```

Job info

```
job.creation_time, job.progress
job.completion_time, effective_runtime, ...
```


For jobs in Orthanc, you can create a Job object with the Job ID.

The Job object is quite handy to follow the state of a job. For example, you could start a couple of jobs and follow their progress with the state attribute in a while loop. You could also just wait for the completion of the job with the wait_until_completion method, which blocks the Python interpreter until the job is note running or pending.

The Job object also has a couple of interesting attributes, such as the completion time or the effective run time.

Anonymize and modify as Job

We can anonymize and modify as job.

Anonymize as job

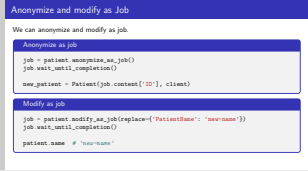
```
job = patient.anonymize_as_job()
job.wait_until_completion()

new_patient = Patient(job.content['ID'], client)
```

Modify as job

```
job = patient.modify_as_job(replace={'PatientName': 'new-name'})
job.wait_until_completion()

patient.name  # 'new-name'
```

A common way to use jobs is when anonymizing or modifying large patients.

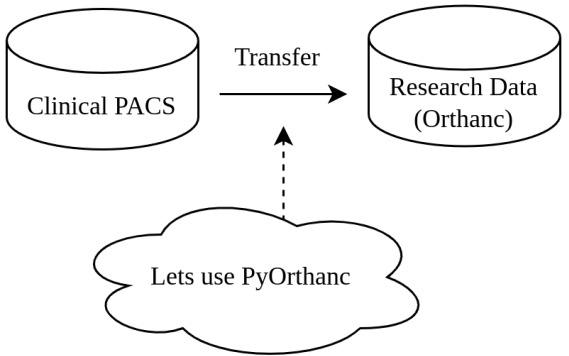
It is usually a good idea to run those processes in a job which can be launched with the anonymize_as_job or modify_as_job methods.

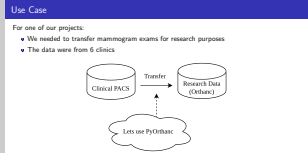
Those methods return job objects which will contain the new patient ID once done.

Use Case

For one of our projects:

- We needed to transfer mammogram exams for research purposes
- The data were from 6 clinics





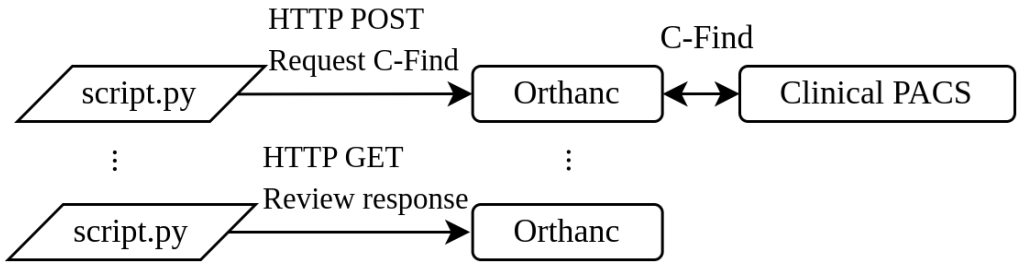
I will briefly present a use case where we use PyOrthanc with success.

We had a project where we needed to transfer mammogram exams from a clinical environment to a research environment.

We actually needed to transfer data from 6 clinics to our research PACS, which was an Orthanc server.

Finding the desired data

From PyOrthanc, it is easy to request a C-Find on a target PACS



Finding the desired data
From PyOrthanc, it is easy to request a C-Find on a target PACS

Our workflow was to use the Modality object from PyOrthanc to make C-Find requests.

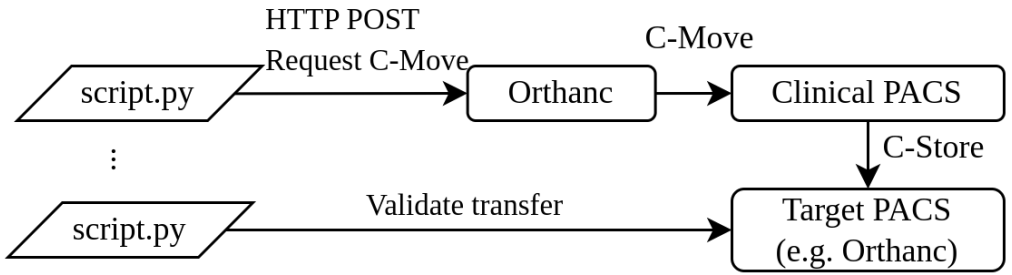
We built our queries from a list of participants to the research project.

As you can see, from Python we can ask an Orthanc Server to perform the C-Find requests to another PACS.

We can then review the results.

Move the data to the desired Orthanc

With the C-Find query response, we then request a C-Move



- Transferred 45,000+ mammogram exams from 6 clinics
- 500-800 exams per night
- PyOrthanc was a big help
- Source code: `https://github.com/ylemarechal/dicom-transfer`

Move the data to the desired Orthanc
With the C-Find query response, we then request a C-Move
HTTP POST
Request C-Move
C-Move
script.py
Orthanc
Clinical PACS
C-Store
script.py
Validate transfer
Target PACS
(e.g. Orthanc)
Transferred 45,000+ mammogram exams from 6 clinics
500-800 exams per night
PyOrthanc was a big help
Source code: https://github.com/ylemarechal/dicom-transfer

With the C-Find results, we then performed C-Move to transfer the data on our research PACS, which was another Orthanc server.

We were able to transfer more than 45,000 exams from 6 clinics at more than 500 per night.

PyOrthanc made this project quite straightforward.

Wrapping up

PyOrthanc facilitates the interaction with Orthanc from Python.

- Documentation: `https://gacou54.github.io/pyorthanc/`
- Github page: `https://github.com/gacou54/pyorthanc`
- PyPi: `https://pypi.org/project/pyorthanc/`
- Citation: `https://doi.org/10.5281/zenodo.3387552`

```
pip install pyorthanc
```


In conclusion, we find that PyOrthanc was a handy library that facilitates the interaction with Orthanc from Python. We hope that you find it useful too.

The documentation, GitHub, PyPi and citation links are all here.

You can install it with a simple pip install pyorthanc.

Future Development

Future development:

- Add an Orthanc SDK mock when developing with the Python plugin
  - This will provide **autocomplete**/**linting**

- Improve documentation
- Improve the `Modality` class

And finally, for future development, we hope to implement an Orthanc SDK mock for the Python plugin. This would ease the development with the Python plugin with nice autocomplete and linting.

We also want to improve the documentation and the Modality class.

Thank you