PyUNICORE

2023 UNICORE

Apr 12, 2024

USING PYUNICORE

UNICORE (**UN**iform **I**nterface to **CO**mputing **RE**sources) offers a ready-to-run system including client and server software. It makes distributed computing and data resources available in a seamless and secure way in intranets and the internet.

PyUNICORE is a Python library providing an API for UNICORE's REST API , making common tasks like file access, job submission and management, workflow submission and management more convenient, and integrating UNICORE features better with typical Python usage.

In addition, this library contains code for using UFTP (UNICORE FTP) for filesystem mounts with FUSE, a UFTP driver for PyFilesystem and a UNICORE implementation of a Dask Cluster

This project has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement Nos. 720270, 785907 and 945539 (Human Brain Project SGA 1, 2 and 3)

CHAPTER

ONE

INSTALLATION

Install from PyPI with

```
pip install -U pyunicore
```

Additional extra packages may be required for your use case:

- Using the UFTP fuse driver requires “fusepy”
- Using UFTP with pyfilesystem requires “fs”
- Creating JWT tokens signed with keys requires the “cryptography” package

You can install (one or more) extras with pip:

```
pip install -U pyunicore[crypto,fs,fuse]
```

1.1 Getting started

1.1.1 Creating a client for a UNICORE site

```
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials
import json

base_url = "https://localhost:8080/DEMO-SITE/rest/core"

# authenticate with username/password
credential = uc_credentials.UsernamePassword("demouser", "test123")

client = uc_client.Client(credential, base_url)
print(json.dumps(client.properties, indent = 2))
```

1.1.2 Running a job and read result files

```
my_job = {'Executable': 'date'}

job = client.new_job(job_description=my_job, inputs=[])
print(json.dumps(job.properties, indent = 2))

job.poll() # wait for job to finish

work_dir = job.working_dir
print(json.dumps(work_dir.properties, indent = 2))

stdout = work_dir.stat("/stdout")
print(json.dumps(stdout.properties, indent = 2))
content = stdout.raw().read()
print(content)
```

1.1.3 Connect to a Registry and list all registered services

```
registry_url = "https://localhost:8080/REGISTRY/rest/registries/default_registry"

# authenticate with username/password
credential = uc_credentials.UsernamePassword("demouser", "test123")

r = uc_client.Registry(credential, registry_url)
print(r.site_urls)
```

1.1.4 More examples

Further examples for using PyUNICORE can be found in the "integration-tests" folder in the source code repository.

1.2 Authentication

PyUNICORE supports all the authentication options available for UNICORE, so you can use the correct one for the server that you are trying to access.

1.2.1 Basic authentication options

The classes for the supported authentication options are in the *pyunicore.credentials* package.

Username and password

```
import pyunicore.credentials as uc_credentials

# authenticate with username/password
credential = uc_credentials.UsernamePassword("demouser", "test123")
```

This will encode the supplied username/password and add it as an HTTP header `Authorization: Basic ...` to outgoing calls.

Bearer token (OAuth/OIDC)

This will add the supplied token as an HTTP header `Authorization: Bearer ...` to outgoing calls.

```
import pyunicore.credentials as uc_credentials

# authenticate with Bearer token
token = "..."
credential = uc_credentials.OIDCToken(token)
```

Basic token

This will add the supplied value as a HTTP header `Authorization: Basic ...` to outgoing calls.

```
import pyunicore.credentials as uc_credentials

# authenticate with Bearer token
token = "..."
credential = uc_credentials.BasicToken(token)
```

JWT Token

This is a more complex option that creates a JWT token that is signed with a private key - for example this is usually an authentication option supported by the UFTP Authserver. In this case the user's UFTP / SSH key is used to sign.

The simplest way to create this credential is to use the *create_credential()* helper function.

```
import pyunicore.credentials as uc_credentials

# authenticate with SSH key
uftp_user  = "demouser"
identity_file = "~/.uftp/id_uftp"
credential = uc_credentials.create_credential(
                     username = uftp_user,
                     identity = identity_file)
```

The `JWTToken` credential can also be used for "trusted services", where a service uses its server certificate to sign the token. Of course this must be enabled / supported by the UNICORE server.

Anonymous access

If for some reason you explicitly want anonymous calls, i.e. NO authentication (which is treated differently from having invalid credentials!), you can use the `Anonymous` credential class:

```
import pyunicore.credentials as uc_credentials

# NO authentication
credential = uc_credentials.Anonymous()
```

This can be useful for simple health checks and the like.

1.2.2 User preferences (advanced feature)

If the user mapping at the UNICORE server gives you access to more than one remote user ID or primary group, you can select one using the user preferences feature of the UNICORE REST API.

The *access_info()* method shows the result of authentication and authorization.

```
import json
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials

credential = uc_credentials.UsernamePassword("demouser", "test123")
base_url = "https://localhost:8080/DEMO-SITE/rest/core"
client = uc_client.Client(credential, base_url)

print(json.dumps(client.access_info(), indent=2)
```

You can get access to the user preferences via the `Transport` object that every PyUNICORE resource has.

For example, to select a primary group (from the ones that are available)

```
client = uc_client.Client(credential, base_url)
client.transport.preferences = "group:myproject1"
```

Note that (of course) you cannot select a UID/group that is not available, trying that will cause a 403 error.

1.2.3 Creating an authentication token (advanced feature)

For some use cases (like automated workflows) you might want to not store your actual credentials (like passwords or private keys) for security reasons. For this purpose, you can use your (secret) credentials to have the UNICORE server issue a (long-lived) authentication token, that you can then use for your automation tasks without worrying that your secret credentials get compromised.

Note that you still should keep this token as secure as possible, since it would allow anybody who has the token to authenticate to UNICORE with the same permissions and authorization level as your real credentials.

You can access the token issue endpoint using the PyUNICORE client class:

```
client = uc_client.Client(credential, base_url)
my_auth_token = client.issue_auth_token(lifetime  = 3600,
                                        renewable = False,
                                        limited   = True)
```

and later use this token for authentication:

```
import pyunicore.credentials as uc_credentials

credential = uc_credential.create_token(token=my_auth_token)
client = uc_client.Client(credential, base_url)
```

The parameters are

- `lifetime` : token lifetime in seconds
- `renewable`: if True, the token can be used to issue a new token
- `limited` : if True, the token is only valid for the server that issued it. If False, the token is valid for all UNICORE servers that the issuing server trusts, i.e. usually those that are in the same UNICORE Registry

1.3 UFTP

UFTP (UNICORE FTP) is a fast file transfer toolkit, based on the standard FTP protocol, with an added authentication layer based on UNICORE.

To make a UFTP connection, a user first needs to authenticate to an authentication service, which will produce a one-time password, which is then used to connect to the actual UFTP file server.

UFTP support in PyUNICORE is based on the ftplib standard library.

1.3.1 Basic UFTP usage

Opening an FTP session involves authenticating to an authentication service using UNICORE credentials. Depending on the authentication service, different credentials might be accepted.

Here is a basic example using username/password.

```
import pyunicore.credentials as uc_credentials
import pyunicore.uftp as uc_uftp

# URL of the authentication service
auth_url = "https://localhost:9000/rest/auth/TEST"

# remote base directory that we want to access
base_directory = "/data"

# authenticate with username/password
credential = uc_credentials.UsernamePassword("demouser", "test123")

uftp_session = uc_uftp.UFTP().connect(credential, auth_url, base_directory)
```

The object returned by *connect()* is an *ftplib FTP* object.

1.3.2 Using UFTP for PyFilesystem

You can create a PyFilesystem *FS* object either directly in code, or implicitely via a URL.

The convenient way is via URL:

```
from fs import open_fs
fs_url = "uftp://demouser:test123@localhost:9000/rest/auth/TEST:/data"
uftp_fs = open_fs(fs_url)
```

The URL format is

```
uftp://[username]:[password]@[auth-server-url]:[base-directory]?[token=...][identity=...]
```

The FS driver supports three types of authentication

- Username/Password - give *username* and *password*
- SSH Key - give *username* and the *identity* parameter, where *identity* is the filename of a private key. Specify the *password* if needed to load the private key
- Bearer token - give the token value via the *token* parameter

1.3.3 Mounting remote filesystems via UFTP

PyUNICORE contains a FUSE driver based on fusepy, allowing you to mount a remote filesystem via UFTP. Mounting is a two step process,

- authenticate to an Auth server, giving you the UFTPD host/port and one-time password
- run the FUSE driver

The following code example gives you the basic idea:

```
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials
import pyunicore.uftp as uc_uftp
import pyunicore.uftpfuse as uc_fuse

_auth = "https://localhost:9000/rest/auth/TEST"
_base_dir = "/opt/shared-data"
_local_mount_dir = "/tmp/mount"

# authenticate
cred = uc_credentials.UsernamePassword("demouser", "test123")
_host, _port, _password  = uc_uftp.UFTP().authenticate(cred, _auth, _base_dir)

# run the fuse driver
fuse = uc_fuse.FUSE(
uc_fuse.UFTPDriver(_host, _port, _password), _local_mount_dir, foreground=False,␣
↪nothreads=True)
```

1.4 Dask integration

PyUNICORE provides the `UNICORECluster` class, which is an implementation of a Dask Cluster, allowing to run the Dask client on your local host (or in a Jupyter notebook in the Cloud), and have the Dask scheduler and workers running remotely on the HPC site.

Here is a basic usage example:

```
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials
import pyunicore.dask as uc_dask

# Create a UNICORE client for accessing the HPC cluster
base_url = "https://localhost:8080/DEMO-SITE/rest/core"
credential = uc_credentials.UsernamePassword("demouser", "test123")
submitter = uc_client.Client(credential, base_url)

# Create the UNICORECluster instance

uc_cluster = uc_dask.UNICORECluster(
   submitter,
   queue = "batch",
   project = "my-project",
   debug=True)

# Start two workers
uc_cluster.scale(2, wait_for_startup=True)

# Create a Dask client connected to the UNICORECluster

from dask.distributed import Client
dask_client = Client(uc_cluster, timeout=120)
```

That's it! Now Dask will run its computations using the scheduler and workers started via UNICORE on the HPC site.

1.4.1 Configuration

When creating the `UNICORECluster`, a number of parameters can be set via the constructor. All parameters except for the submitter to be used are OPTIONAL.

- *submitter*: this is either a Client object or an Allocation, which is used to submit new jobs
- *n_workers*: initial number of workers to launch
- *queue*: the batch queue to use
- *project*: the accounting project
- *threads*: worker option controlling the number of threads per worker
- *processes*: worker option controlling the number of worker processes per job (default: 1)
- *scheduler_job_desc*: base job description for launching the scheduler (default: None)
- *worker_job_desc*: base job description for launching a worker (default: None)
- *local_port*: which local port to use for the Dask client (default: 4322)

- *connect_dashboard*: if True, a second forwarding process will be lauched to allow a connection to the dashboard (default: False)
- *local_dashboard_port*: which local port to use for the dashboard (default: 4323)
- *debug*: if True, print some debug info (default: False)
- *connection_timeout*: timeout in seconds while setting up the port forwarding (default: 120)

1.4.2 Customizing the scheduler and workers

By default, the Dask extension will launch the Dask components using server-side applications called `dask-scheduler` and `dask-worker`, which need to be defined in the UNICORE IDB.

The job description will look like this:

```
{
      "ApplicationName": "dask-scheduler",
      "Arguments":  [
        "--port", "0",
        "--scheduler-file", "./dask.json"
      ],
      "Resources": {
              "Queue": "your_queue",
              "Project": "your_project"
      }
}
```

If you want to customize this, you can pass in a basic job description when creating the `UNICORECluster` object.

The job descriptions need not contain all command-line arguments, the `UNICORECluster` will add them as required. Also, the queue and project will be set if necessary.

For example

```
# Custom job to start scheduler

sched_jd = {
      "Executable" : "conda run -n dask dask-scheduler",
      "Resources": {
              "Runtime": "2h"
      },
      "Tags": ["dask", "testing"]
}

# Custom job to start worker

worker_jd = {
      "Executable" : "srun --tasks=1 conda run -n dask dask-worker",
      "Resources": {
              "Nodes": "2"
      }
}

# Create the UNICORECluster instance
uc_cluster = uc_dask.UNICORECluster(
```

(continues on next page)

(continued from previous page)

```
    submitter,
    queue = "batch",
    project = "my-project",
    scheduler_job_desc=sched_jd,
    worker_job_desc=worker_jd
    )
```

1.4.3 Scaling

To control the number of worker processes and threads, the UNICORECluster has the scale() method, as well as two properties that can be set from the constructor, or later at runtime

The scale() method controls how many workers (or worker jobs when using "jobs=…" as argument) are running.

```
# Start two workers
uc_cluster.scale(2, wait_for_startup=True)

# Or start two worker jobs with 4 workers per job
# and 128 threads per worker
uc_cluster.processes =   4
uc_cluster.threads   = 128
uc_cluster.scale(jobs=2)
```

1.4.4 The dashboard

By default a connection to the scheduler's dashboard is not possible. To allow connecting to the dashboard, set `connect_dashboard=True` when creating the `UNICORECluster`. The dashboard will then be available at `http://localhost:4323`, the port can be changed, if necessary.

1.4.5 Using an allocation

To speed up the startup and scaling process, it is possible to pre-allocate a multinode batch job (if the server side UNICORE supports this, i.e. runs UNICORE 9.1 and Slurm), and run the Dask components in this allocation.

```
import pyunicore.client as uc_client
import pyunicore.credentials as uc_credentials
import pyunicore.dask as uc_dask

# Create a UNICORE client for accessing the HPC cluster
base_url = "https://localhost:8080/DEMO-SITE/rest/core"
credential = uc_credentials.UsernamePassword("demouser", "test123")
submitter = uc_client.Client(credential, base_url)

# Allocate a 4-node job
allocation_jd = {
      "Job type": "ALLOCATE",

      "Resources": {
              "Runtime": "60m",
```

(continues on next page)

(continued from previous page)

```
                "Queue": "batch",
                "Project": "myproject"
        }
}

allocation = submitter.new_job(allocation_jd)
allocation.wait_until_available()

# Create the UNICORECluster instance using the allocation

uc_cluster = uc_dask.UNICORECluster(allocation, debug=True)
```

Note that in this case your custom scheduler / worker job descriptions MUST use `srun --tasks=1 ...` to make sure that exactly one scheduler / worker is started on one node.

Also make sure to not lauch more jobs than you have nodes - otherwise the new jobs will stay "QUEUED".

1.5 Port forwarding / tunneling

Opens a local server socket for clients to connect to, where traffic gets forwarded to a service on a HPC cluster login (or compute) node. This feature requires UNICORE 9.1.0 or later on the server side.

You can use this feature in two ways

- in your own applications via the `pyunicore.client.Job` class.
- you can also open a tunnel from the command line using the `pyunicore.forwarder` module

Here is an example for a command line tool invocation:

```
LOCAL_PORT=4322
JOB_URL=https://localhost:8080/DEMO-SITE/rest/core/jobs/some_job_id
REMOTE_PORT=8000
python3 -m pyunicore.forwarder --token <your_auth_token> \
   -L $LOCAL_PORT \
   $JOB_URL/forward-port?port=REMOTE_PORT \
```

Your application can now connect to `localhost:4322` but all traffic will be forwarded to port 8000 on the login node.

See

```
python3 -m pyunicore.forwarder --help
```

for all options.

1.6 License

```
BSD 3-Clause License

Copyright (c) Human Brain Project, Forschungszentrum Juelich GmbH
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the names of the copyright holders nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```