ProxyStore: a Data Fabric for Parsl and FuncX

Greg Pauloski
14 September 2022
Decouple Object Communication from System Design

1. Compatibility
2. Productivity
3. Performance
Proxy Objects

- Transparently wrap **target** objects
- Acts like a wide-area reference
- Initialized with a **factory**
- Just-in-time **resolution**

```python
import numpy as np
from proxystore.proxy import Proxy

x = np.array([1, 2, 3])

# Proxy(Callable[[], T]) -> Proxy[T]
p = Proxy(lambda: x)

# A proxy is an instance of its wrapped object
assert isinstance(p, Proxy)
assert isinstance(p, np.ndarray)

# The proxy can do everything the numpy array can
assert np.array_equal(p, [1, 2, 3])
assert np.sum(p) == 6
y = x + p
assert np.array_equal(y, [2, 4, 6])
```
```python
import torch
from funcx.sdk.client import FuncXClient, FuncXExecutor
from proxystore.proxy import Proxy

def load_model() -> MyModel:
    state_dict = torch.load('/path/to/model')
    return MyModel().load_state_dict(state_dict)

def inference(model: MyModel) -> Result:
    ...

fx = FuncXExecutor(FuncXClient())

# Model will be lazily "resolved" once needed by inference()
# and no consumer-side code changes are needed
res = fx.submit(inference, Proxy(load_model), endpoint_id=...)```
```python
def compute(obj: MyData) -> Result:
    # Computation ...
    return Result(...)

def compute(
    obj: MyData | pathlib.Path | str,
) -> Result:
    obj = resolve(obj)
    # Computation ...
    return Result(...)

def resolve(
    obj: MyData | pathlib.Path | str,
) -> MyData:
    if isinstance(obj, str):
        obj = deserialize(redis.get(obj))
    elif isinstance(obj, pathlib.Path):
        with open(obj) as f:
            obj = deserialize(f.read())
    return obj
def compute(obj: MyData) -> Result:
    assert isinstance(obj, MyData)
    assert isinstance(obj, Proxy)
    # Computation ...
    return Result(...)

class FileFactory:
    def __init__(self, filepath: str) -> None: ...
    def __call__(self) -> MyData: ...

class RedisFactory:
    def __init__(self, key: str, address: str) -> None: ...
    def __call__(self) -> MyData: ...

compute(Proxy(FileFactory('/path/to/data')))

The proxy looks like MyData...
but only contains the code for how to become MyData.
Why ProxyStore?

**Store Interface**

- `.proxy()` method
- Factory implementations:
  - Shared file systems
  - Redis/KeyDB
  - Globus
  - Easy to add new ones

Proxies *look* like the object... *only* contain how to retrieve the object from the store that created them

```python
from proxystorestore import init_store

def compute(obj: MyData) -> Result:
    # Computation ...
    return Result(...)

# Stores registered globally
store = init_store(
    'redis', name='mystore1', hostname=..., port=...
)
store = init_store(
    'file', name='mystore2', data_dir=...
)
compute(store.proxy(obj))
```
Avoid unnecessary communication
Easy RDMA + Multi-site Workflows

```
Endpoint 1

Host A
Producer

Object
Store

1

NAT

2

obj

Cloud/Workflow
Engine

NAT

4

Endpoint 2

Host B
Consumer

Object
Store

3

obj

Apparent Data Path

True Data Path

Data Requests
```
Questions?

jgpauloski@uchicago.edu

github.com/proxystore

proxystore.rtfd.io