

RAID-on-Cloud: A Cloud-based Replication NAS Backed by Multiple Providers

1. Project Description

Keywords: Public Cloud Services, Replication, Vendor Lock-in, Object Storage

In this project, we will be designing a NAS (network-attached storage) backed by three public cloud storage services. These cheap, scalable, and highly reliable cloud storage services are available for both internal use of the cloud and external access of client-side applications. The variety and convenience of cloud storage give users huge leverage when developing web-connected applications.

From the perspective of customers, a single cloud provider may not be trustworthy enough for providing the best quality of service and cost-efficiency. A single-platform application suffers the “**vendor lock-in**” problem, blocking the customers from migrating to better cloud providers. More and more customers have adopted a multi-platform approach, in order to obtain better services.

Figure 1 describes **RAID-on-Cloud**, a cloud-based replication NAS which we will implement in this project. The client-end will be implemented in Python, as a CLI (command-line interface) running in a [Singularity Container](#) backed by TAMU’s High-Performance Research Center (HPRC). Through the Python CLI, users can access files without knowing how data are actually stored and distributed across cloud platforms.

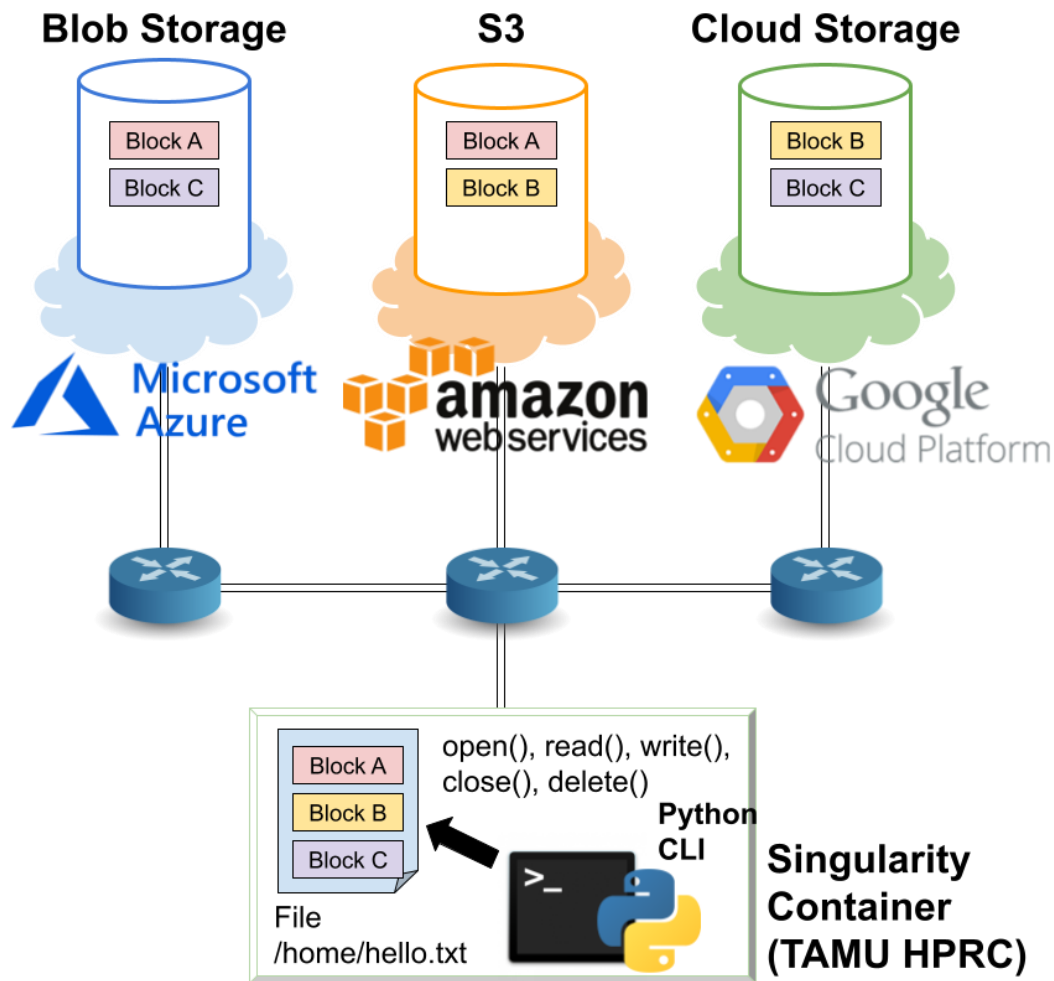


Figure 1: Architecture of **RAID-on-Cloud**

The files stored in RAID-on-Cloud will be replicated across three cloud providers, AWS S3, Azure Blob Storage, and Google Cloud Storage, in a **RAID-5 style** (without parity bits). The minimum RAID-5 setup requires at least three storage media, with each block replicated to at least two locations. The file blocks must be stored as objects in the cloud storage services, and the storage sizes in all three cloud storage services must be equally distributed.

2. Learning Goal

In this project, you will learn:

- how to develop a client-side application and access multiple cloud storage services via APIs and SDKs; and
- how to authenticate with public cloud platforms; and
- how to balance the utilization of cloud resources across platforms; and
- how to implement the traditional, POSIX-style filesystem functions, such as `read()` and `write()`, as wrappers of object storage APIs, and to implement a RAID-like replication system.

For more advanced projects, you can:

- implement the NAS balancing feature to adjust the ratio of storage sizes across the cloud platforms. For example, you can adjust the ratio between AWS S3, Azure Blob, and Google Cloud Storage from 1:1:1 to 3:2:1, so that your customers can obtain better cost-efficiency.
- implement the open-to-close consistency of the NAS filesystem, using the versioning read and write features of the object storage.

The rest of this document will describe each step to finish the project:

- Step 1: Set Up the Singularity Container
- Step 2: Trying the CLI and Configuring for AWS, Azure, and GCP
- Step 3: Developing the NAS

3. Step 1: Set Up the Singularity Container


In this project, you will be using the Singularity Container hosted by TAMU's HPRC Center to develop the client-side program. **You are free to use your own Linux environment** to develop the project, but you must ensure that your finished program will properly run in the Singularity Container for the purpose of grading.

To obtain a Singularity Container, you will need to apply for a basic account in HPRC Center. The whole application will take roughly **one working day** to process. To apply for the account, access the following URL: <https://hprc.tamu.edu/apply/>


1. HPRC Portal Login

- <https://portal.hprc.tamu.edu/>
- Once on the VPN, use the link above to access the portal:

High Performance Research Computing
A Resource for Research and Discovery


TEXAS A&M
 UNIVERSITY


TAMU HPRC OnDemand Homepage



[Ada OnDemand Portal](#)

↑

Use this link



[Terra OnDemand Portal](#)

[OnDemand Portal User Guide](#)

- Log in with your NetID, and you will be greeted with this page:

TAMU HPRC OnDemand (Ada) Files Jobs Clusters Interactive Apps

OnDemand provides an integrated, single access point for all of your HPC resources.

Message of the Day

**** Job Scheduling Stopped on HPRC Clusters, February 15 ****

Job scheduling has been stopped on all clusters to help conserve power and reduce the heat load in the TAMU data centers during the state of emergency in Texas.

IMPORTANT POLICY INFORMATION

2. File Access

- The first thing to note on the HPRC portal is the “Files” tab. This tab allows you to upload and download files to the HPRC container. For the sake of this course, the home directory will be enough.

TAMU HPRC OnDemand (Ada) Files Jobs Clusters Interactive Apps

Home Directory
/scratch/user/arearnest7

OnDemand provides an integrated, single access point for all of your HPC

- When accessed, you will see this page. From here, you are able to perform basic file manager operations for the sake of uploading and downloading code for ease of use.

Go To... Open in Terminal New File New Dir Upload Show Dotfiles Show Owner/Mode

Home Directory

- csce678-s21-project1-master
- devstack
- intel

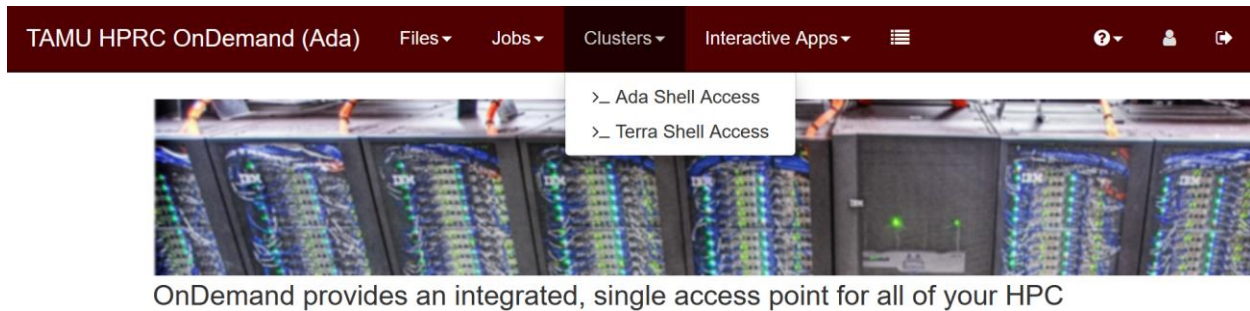
/home/arearnest7/csce678-s21-project1-master/

View Edit Rename Download Copy Paste (Un)Select All Delete

name	size	modified date
..	<dir>	
tests	<dir>	02/19/2021
678-project1.job	1.12kb	02/19/2021
LICENSE.md	1.04kb	02/19/2021
NAS.py	6.32kb	02/19/2021
basic_defs.py	815b	02/19/2021

3. Ada Terminal

- Now that you have access to the portal, you can access the interactive user shell for the HPRC container from the “Clusters” tab.



- Select the “Ada Shell Access” to gain access to the shell.
- From here, use your NetID password and use your 2FA to access the shell as shown.

```
*****
This computer system and the data herein are available only for authorized
purposes by authorized users: use for any other purpose is prohibited and may
result in administrative/disciplinary actions or criminal prosecution against
the user. Usage may be subject to security testing and monitoring to ensure
compliance with the policies of Texas A&M University, Texas A&M University
System, and the State of Texas. There is no expectation of privacy on this
system except as otherwise provided by applicable privacy laws. Users should
refer to Texas A&M University Standard Administrative Procedure 29.01.03.M0.02,
Rules for Responsible Computing, for guidance on the appropriate use of Texas
A&M University information resources.
*****

Password:
Duo two-factor login for arearnest7

Enter a passcode or select one of the following options:

1. Duo Push to XXX-XXX-
2. Phone call to XXX-XXX-
3. SMS passcodes to XXX-XXX-

Passcode or option (1-3): 1
Success. Logging you in...
```

- From here, you have access to the shell, which has basic Linux shell permissions and tools, including Python 2.6.8 (i.e. python), pip, and git.


```

Last login: Fri Feb 19 09:44:30 2021 from login7-eth1
=====
|      Texas A&M University High Performance Research Computing      |
|-----|
| Website:           https://hprc.tamu.edu |
| Consulting:        help@hprc.tamu.edu (preferred) or (979) 845-0219 |
| Ada Documentation: https://hprc.tamu.edu/wiki/Ada |
| Terra Documentation: https://hprc.tamu.edu/wiki/Terra |
| Grace Documentation: https://hprc.tamu.edu/wiki/Grace |
| YouTube Channel:   https://www.youtube.com/texasamhprc |
|-----|
|
| *****
|      == IMPORTANT POLICY INFORMATION ==
| *****
| - Unauthorized use of HPRC resources is prohibited and subject to
|   criminal prosecution.
| - Use of HPRC resources in violation of United States export control
|   laws and regulations is prohibited. Current HPRC staff members are
|   US citizens and legal residents.
| - Sharing HPRC account and password information is in violation of
|   Texas State law. Any shared accounts will be DISABLED.
| - Authorized users must also adhere to ALL policies at:
|   https://hprc.tamu.edu/policies/
| *****
|
| !! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!
|
| Please restrict usage to 8 CORES across ALL login nodes.
|   Users found in violation of this policy will be SUSPENDED.
|
| **** Job Scheduling Stopped on HPRC Clusters, February 15 ****
|
| UPDATE: (2/19/2021 5:31pm): Job scheduling has resumed at reduced levels
| with 40% of the compute nodes respectively for Ada and Terra until TAMU
| IT approves further increases after monitoring impact on the data center.
| Ada currently has limited cooling in the data center.
|
| Job scheduling has been stopped on all clusters to help conserve power
| and reduce the heat load in the TAMU data centers during the state of
| emergency in Texas.
|
| To see these messages again, run the motd command.
|
| Your current disk quotas are:
| Disk      Disk Usage      Limit      File Usage      Limit
| /home      20.88M          10G          480             10000
| /scratch    0                  1T             1             250000
| /tiered     0                  10T             1             50000
| Type 'showquota' to view these quotas again.
| [arearnest7@ada8 ~]$

```

Note:

To properly run the right version of Python in the HPRC container, you must load an extra modules. Run the following command after you logged in:

```
$ module load Python/2.7.18-GCCcore-10.2.0
```

From this point, all pip installations and python functionality should work as intended.

4. Step 2: Trying the CLI and Configuring for AWS, Azure, and GCP

Once you have access to the Singularity Container, the first thing you need to do is to clone the project 1 repository which contains the basic framework for RAID-on-Cloud. Type the following commands into the terminal:

```

$ git clone https://github.com/chiaache/csce678-s21-project1.git
$ cd ~/csce678-s21-project1
$ git submodule update --init

```

To develop the framework, multiple Python packages need to be installed. Since singularity containers do not allow installing packages as root, you will need to install in the local directory. To help us simplify grading, please install the required packages in “csce678-s21-project1/lib”.

```
$ cd ~/csce678-s21-project1
$ pip install --target=$PWD/lib --upgrade boto3 \
    google-cloud-storage \
    azure-storage-blob
```

If you find pip unavailable in your singularity container, you can run the following commands to install pip locally first:

```
$ cd ~/csce678-s21-project1
$ wget https://bootstrap.pypa.io/2.6/get-pip.py
$ python get-pip.py --user
$ ~/.local/bin/pip install --target=$PWD/lib boto3 --upgrade \
    google-cloud-storage azure-storage-blob
```

Note:

We are no longer recommending the installation of a local pip. If you are still using it, most likely you are still running Python 2.6. Please follow the steps above to load the module of Python 2.7.

TA's note:

If you encounter an error with the installation of google-cloud-storage warning about “no module named google.cloud”, this is an unintended side effect of the pip installation leaving out 2 `__init__.py` files in the correct directories. If you have this issue, please follow the following directions.

- Add a file named `__init__.py` in `csce678-s21-project1/lib/google/` and `csce678-s21-project1/lib/google/cloud/`:

```
$ cd ~/csce678-s21-project1
$ echo "__path__ = __import__('pkgutil').extend_path(__path__, __name__)" \
    > lib/google/__init__.py
$ cp lib/google/__init__.py lib/google/cloud/__init__.py
```

Doing the steps should resolve the unknown module error. If you encounter an error similar to this or still experience this error after these instructions, please let us know.

Once the environment is set up, you may try to run the Python CLI program. The Python CLI program can be run with the following command:

Function not implemented.

The reason for this warning message is that your system is not yet implemented and not yet connected to AWS, Azure, and GCP. However, you can still test with a local filesystem-based implementation, without access to the cloud:

```
cd ~/csce678-s21-project1
PYTHONPATH=$PWD/lib python NAS.py --local
*****
**      Welcome to the RAID-on-Cloud NAS!      **
**                                              **
** Type one of the following cmds:             **
**      o  / open   <filename>                  **
**      r  / read   <fd> <len> <offset>          **
**      rb / readb  <fd> <len> <offset>          **
**      w  / write  <fd> <offset>                **
**          (input from the next line)           **
**      wb / writeb <fd> <offset>                **
**          (input from the next line)           **
**      c  / close  <fd>                        **
**      d  / delete <filename>                  **
**      q  / quit                                     **
*****
NAS> o hello.txt
Opened file descriptor 0 for hello.txt
NAS> w 0 0
Enter the content to write (To break, type <ENTER> and then <Control-
D>):
hellohello
NAS> r 0 10 0
hellohello<eof>
NAS> rb 0 10 0
0000  68 65 6C 6C 6F 68 65 6C 6C 6F  hellohello
NAS> c 0
File descriptor 0 closed.
NAS> d hello.txt
File hello.txt deleted.
NAS> q
Goodbye!!!
```

Here's a list of commands you can use in the CLI:

- **open** (abbreviated as **o**):
Opening a file based on the filename given as the argument. There is no need to implement any directory or filesystem hierarchy. For example, 'hello.txt' and '/hello.txt' will be treated as different files since they have different names. Once the file is successfully opened, the CLI prints the file descriptor on the screen.
- **read** (abbreviated as **r**):
Reading the file descriptor up to the given number of bytes, at the given offset. Once the file is read successfully, the CLI will print the output directly on the screen as UTF-8 strings.
- **readb** (abbreviated as **rb**):
Reading the file descriptor up to the given number of bytes, at the given offset. Once the file is read successfully, the CLI will print the hexdump of the output on the screen.
- **write** (abbreviated as **w**):
Reading the string from the screen and write to the file descriptor at the given offset. The string will be read until the CLI detects a line break followed by a Control-D.
- **writb** (abbreviated as **wb**):
Reading the hexadecimal representation from the screen and write to the file descriptor at the given offset. The string will be read until the CLI detects a line break followed by a Control-D.
- **close** (abbreviated as **c**):
Close the given file descriptor.
- **delete** (abbreviated as **d**):
Delete the file based on the filename given.
- **quit** (abbreviated as **q**):
Quit the CLI.

Configuring the Authentication for AWS, Azure, and GCP:

To access the cloud storage from the CLI, you need to provide the authentication credentials for the SDKs. To enter the credentials, open `cloud.py` to fill in the information. The following instructions will walk you through the configuration of each platform.

- **Configuration for AWS:**

The authentication information for AWS will be given as parameters in the class `AWS_S3`. Copy and paste the following information into the Python script:

- **access_key_id**: 'AKIAZ3WFZEEF2ARUKYJL'
- **access_secret_key**: 'gYPzw1DMdRChVpzw7eoQn4EXW/jF1Ks/j8CzS7em'
- **bucket_name**: Your bucket will be 'csce678-s21-p1-<Your UIN>'. For example, if your UIN is 123456789, then your bucket will be 'csce678-s21-p1-123456789'.

- **Configuration for Azure:**

The authentication information for Azure will be given as parameters in the class `Azure_Blob`. Copy and paste the following information into the Python script:

- **key:**
`'u04DPr/UGGADYc127vrXG31AZ7cMP7LC+4Y3NKuR3nL8jLkp0xwG9NRzfCtDHG2n4xX4adldrHfFmHrT3afA=='`
- **conn_str:**
`'DefaultEndpointsProtocol=https;AccountName=csce678s21;AccountKey=u04DPr/UGGADYc127vrXG31AZ7cMP7LC+4Y3NKuR3nL8jLkp0xwG9NRzfCtDHG2n4xX4adldrHfFmHrT3afA==;EndpointSuffix=core.windows.net'`
- **account_name:** `'csce678s21'`
- **container_name:** Your bucket will be `'csce678-s21-p1-<Your UIN>'`. For example, if your UIN is 123456789, then your bucket will be `'csce678-s21-p1-123456789'`.

- **Configuration for GCP:**

The authentication information for GCP will be given both as a JSON file in the local directory and parameters in the class `Google_Cloud_Storage`. First, please download and store this credential file into the directory of your repository:

```
$ wget --no-check-certificate  
'https://docs.google.com/uc?export=download&id=1G7SBEGJBJZZ4Tsie_X45f7Kf  
kroabHGp' -O gcp-credential.json
```

And then copy and paste the following information into the script:

- **credential_file:** `gcp-credential.json`
- **bucket_name:** Your bucket will be `'csce678-s21-p1-<Your UIN>'`. For example, if your UIN is 123456789, then your bucket will be `'csce678-s21-p1-123456789'`.

5. Step 3: Developing the NAS

The next step is to implement the APIs for each cloud to handle the block operations. The block operations are defined in the class `cloud_storage` (`basic_defs.py`) and should be implemented in the classes `AWS_S3`, `Azure_Blob`, and `Google_Cloud_Storage`. (**hint: Do not directly implement the functions in `cloud_storage`!**)

See the definition of `cloud_storage`:

```

class cloud_storage:
    block_size = 4096          # System configuration -- DO NOT CHANGE!

    def list_blocks(self):
        raise NotImplementedError

    def read_block(self, offset):
        raise NotImplementedError

    def write_block(self, block, offset):
        raise NotImplementedError

    def delete_block(self, offset):
        raise NotImplementedError

```

Our system assumes all blocks are **4096 bytes**. To store a file, you must first split the file into blocks, and then replicate the blocks to the cloud. If the size of a file is not multiple of 4096 bytes, the last block may be less than 4096 bytes. If a cloud storage service(e.g., AWS S3) is chosen to store a specific block, you must **put** the block as a single **object** to the cloud. Similarly, to read each block, you must **get** the block as a single object from the cloud. If you need to update the block, you must get the block from the cloud, make the update in place, and then put the block back to the cloud.

As a result, you must learn how to get and put objects in AWS S3, Azure Blob Storage, and Google Cloud Storage, using their provided SDKs. For each class that extends `cloud_storage`, you must implement the following four functions:

- `list_blocks()`: List all the blocks as an array of offsets that are currently stored inside the cloud storage.
- `read_block(offset)`: Read the block at the given offset, and return the block as a byte array.
- `write_block(block, offset)`: Write the block as a byte array into the given offset.
- `delete_block(offset)`: Delete the block as the given offset.

Using these four functions would be sufficient to implement the filesystem APIs supported by the RAID-on-Cloud NAS. First, you must determine, for each file and each of its offsets, **which two cloud storage services will be used** to replicate the data, and **at which offset** the block will be stored in each cloud storage service. You can design a mapping function as follows:

(filename, offset) →
[(cloud storage 1, block offset 1), (cloud storage 2, block offset 2)]

(Example added on 3/8):

For example, suppose you are reading a file called hello.txt at the offset 5000. First, you should align the file offset to the nearest block offset. As a result, offset 5000 will be part of the block from the 4096th byte to the 8192nd byte. Then, using a hashing function you can determine which two cloud storages should be used to store the block, at which offset respectively. Again, for instance, the result of your hashing function **H** is $H(\text{'hello.txt'}, 4096) = 10000$. You can divide the hash value (i.e., 10000) by 3 to determine which cloud storage should be chosen and the block offset (hints: use the remainder to decide the cloud storage!).

(Note):

Using the hash function will inevitably result in collision; in other words, with certain probability, two different offsets in different filenames may be mapped to the same cloud storages and the same block offsets. In this project, we DO NOT require you to solve the collision problem as long as you are using a collision-resistant hash function like SHA or MD5.

(Note):

Note that the block offsets used to store the data in the cloud storages (not the file offsets of the blocks) DO NOT have to be the actual block offsets. Since the cloud storages we used as backends are object storages instead of block storages, the cloud storages are not providing a continuous space to store your data. Instead, you can simply use block offsets as the keys to store the blocks as values in the object storages.

Based on this mapping, you should implement all the functions defined in the basic class **NAS**, as part of the extended class named **RAID_on_Cloud** (hint: **Do not directly implement the functions in NAS!**):

```
class NAS:
    def open(self, filename):
        raise NotImplementedError

    def read(self, fd, len, offset):
        raise NotImplementedError

    def write(self, fd, data, offset):
        raise NotImplementedError

    def close(self, fd):
        raise NotImplementedError

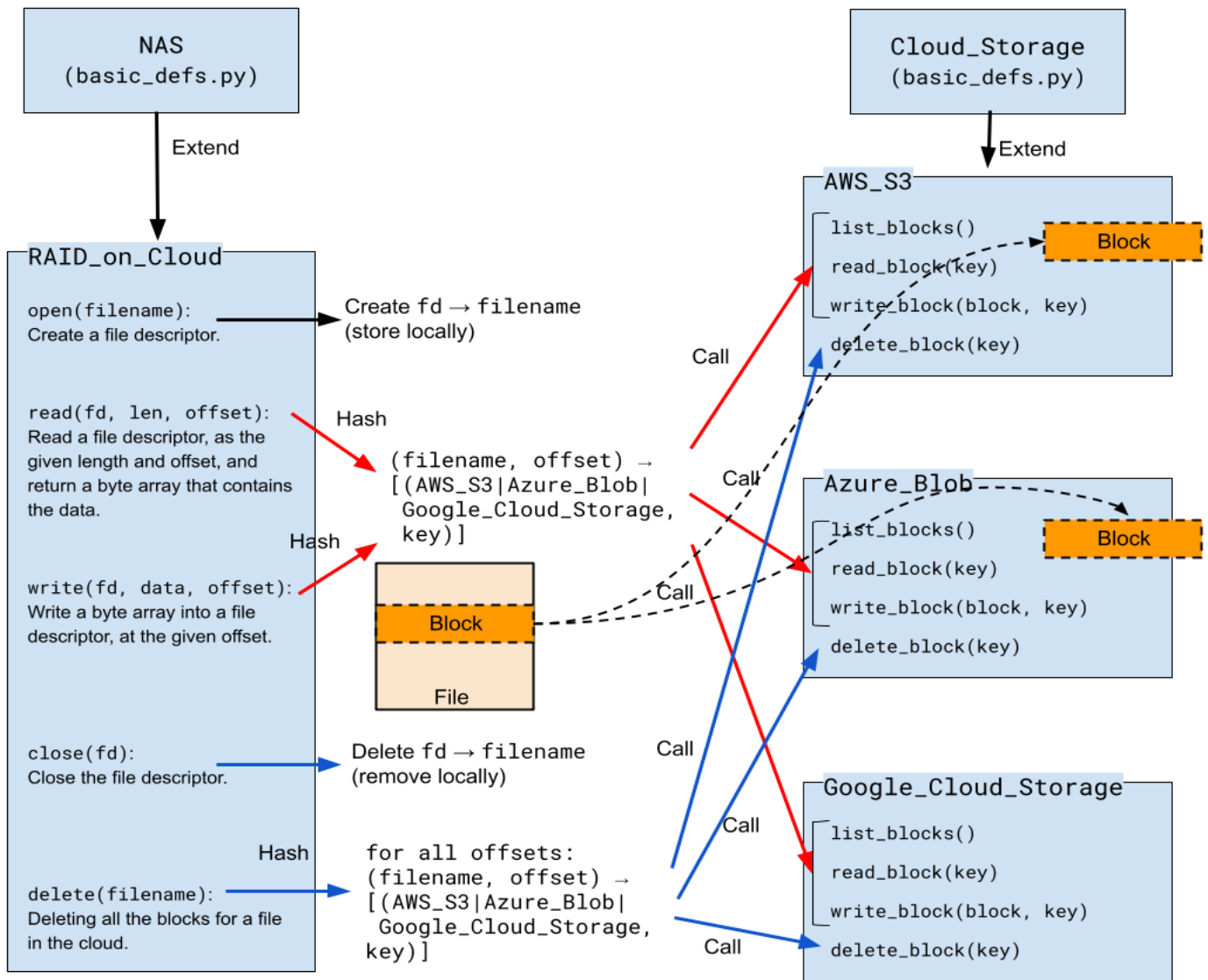
    def delete(self, filename):
        raise NotImplementedError

    def get_storage_sizes(self):
        return [len(b.list_blocks()) for b in self.backends]
```


In NAS, besides `get_storage_sizes` which need not be implemented, all the other functions must be implemented in `RAID_on_Cloud`, including:

- `open(filename)`: Create a file descriptor to represent the file. Because RAID-on-Cloud NAS does not store metadata in the cloud, `open(filename)` does not distinguish whether the file has been previously created and written. `open(filename)` should always succeed and return a file descriptor. All files are opened as readable and writable.
- `read(fd, len, offset)`: Read the data of the opened file descriptor, as the given length and offset, and return a byte array that contains the data. If the file does not exist in the cloud, or the offset has exceeded the end of the file, return a byte array with 0 byte.
- `write(fd, data, offset)`: Write the data store in a byte array into the opened file descriptor, at the given offset. No return value is needed. The function should always succeed. If the file is previously written and the newly written offset and length have overlapped with the original file size, the overlapped data will be overwritten. You must implement in-place updating to handle this corner case.
- `close(fd)`: Simply close the file and deallocate the file descriptor. This function should always return successfully as long as the
 - given file descriptor is valid.
- `delete(filename)`: Deleting all the blocks for a specific file, including all the data which are replicated to the cloud.

Here is an overview of the intended architecture:



• Testing the framework

To help you test your implementation, we have provided a testing framework with plenty of test cases. To run the test framework, first ensure your local Git repository `csce678-s21-project1` has a `tests` directory that is a Git submodule and is up-to-date:

```
$ cd ~/csce678-s21-project1
$ git pull
$ git submodule update --init
```

Once you have the latest test cases, you can run the whole test suite by running the following commands:

```
$ ./run-tests.sh
```

If your framework is not yet fully implemented, or you only want to test a specific component of the system, you can use the following command to run the specific test cases:COOr:

```
$ ./run-tests.sh tests.test_2_NAS.test_1_RAID_on_Cloud
```

Note:

Another way of running the individual test cases is to use the following commands:

```
$ PYTHONPATH=$PWD/lib:$PWD python tests/test_1_cloud_storage.py -v
$ PYTHONPATH=$PWD/lib:$PWD python tests/test_1_cloud_storage.py -v test1_AWS_S3
$ PYTHONPATH=$PWD/lib:$PWD python tests/test_2_NAS.py -v
```