RACS: Extended Version in Java

Gary Zibrat
gdz4

Abstract

Cloud storage is becoming increasingly popular and cheap. It is convenient for companies to simply store their data online so that they don't have to buy a large amount of storage, set it up, and pay people to maintain it. Companies such as Netflix store as much as 3 petabytes of information in the cloud (typical home computers can store up to 1-2 terabytes of information and 1 petabyte is 1024 terabytes). When storing such a volume of data the client is susceptible to being locked in with a provider due to the high costs of transferring data out of the cloud and relatively low cost of actually storing the data. This report discusses the feasibility of storing files evenly over multiple providers in a system called Redundant Array of Cloud Storage (RACS). First I will discuss RACS' evolution and then I will explain the current version of RACS and finally compare the performance of two versions of RACS.

1. Introduction

Relying on a single provider can be risky. Customers can experience vendor lock-in where the client becomes bound to the provider due to cost reasons. Figure 1 reveals that the cost to transfer data out of the cloud is significantly more than storage itself. Users typically also have to pay for each request they issue, but these prices are normally negligible. Most cloud providers don't charge for in data transfer. While providers have 99.99+% availability for data, the .01% of the time when data is inaccessible are normally large chunks consisting of multiple hours/days. So 99.99% availability isn't the data becoming unavailable for a few minutes every week, but rather large windows of unavailability. This could halt business for a whole day for companies that rely on the cloud. Some providers, such as Microsoft, realize this and allow users to upload to two separate places for slightly less than double the cost of a single upload. RACS aims to mitigate this effect while keeping costs down by evenly distributing your data among multiple providers. Cloud providers also lessen their prices when you store more data to prevent users from distributing files, but the price drops are only around %20.

2. History

The original RACS set out with the following goals:

Tolerating Outages: When a provider is unavailable, a user should still be able to get the data

	Storage	Transfer out	Put Request	Get Request
Microsoft	$0.024 per GB/month	$0.080 per GB	$0.000036 per 1,000 transactions	$0.000036 per 1,000 transactions
Amazon	$0.0290 per GB/month	$.080 per GB	$0.005 per 1,000 requests	$0.0004 per 1,000 requests
Google	$0.026 per GB/month (flat rate)	$0.080 per GB	$0.01 per 1,000 requests	$0.001 per 1,000 requests

Figure 1: Shows the prices of cloud storage for three popular and cheap providers.

using RACS.

Tolerating Data Loss: It is rare for providers to lose data but it does happen. Using the same techniques as handling Outages, RACS can handle some data loss.

Adapting to Price Changes: Since data is spread out evenly over multiple providers, if one provider lowers prices users of RACS benefit.

RACS was designed to a have a simple interface that mimicked Amazon's cloud storage interface. There were a few functions that are self-explanatory: GET – takes in a file name and returns the data associated with the file, PUT – takes in a file name and a file and uploads it the cloud providers, DELETE – takes in a file name and deletes the file on the cloud. It also has some non-familiar functions: LOCATE – finds which cloud provider RACS decided to put your data in, PUTAT – allows the user to put the data in a specific provider. Racs-EV adds a new function called PUTS which allows users to upload multiple files in a single transaction.

```
public abstract class Repository {

  boolean put(String key, String Bucket,
    InputStream data, int size) throws Exception;

  InputStream get(String key,
    String Bucket) throws Exception;

  boolean remove(String key,
    String bucket) throws Exception;

  Iterator<String> getKeys(String bucket)
    throws Exception;
```

Figure 2: In order to add a new repository only this simply interface must be extended.

The original RACS accomplished the goals by splitting up a file into smaller fixed sized chunks and sending them evenly to the cloud providers. Using erasure coding on the chunks allows RACS to generate an additional chunk that can be used to recover one other missing chunk of the real file.

Some users of RACS may wish to do cloud computation, or use computers in the same data center as the storage to do operations and computation on the files stored in the cloud. This is beneficial because transfer data from the storage computers to the computation computers is normally free as long as they are both provided by the same

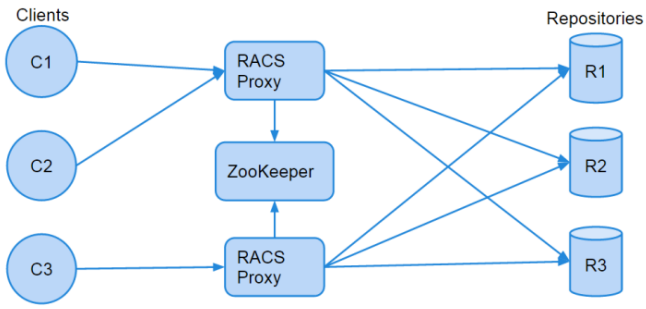


Figure 3: Shows the basic setup of RACS. Clients send data to RACS, and then RACS uses a program called ZooKeeper to synchronize access between RACS servers and finally uploads the data to the repositories

provider. This is a problem for the original RACS since each file is spread up in little chunks all over the cloud. Doing a computation on one file requires accessing most of the other providers in order to reconstruct the file.

2.1 RACS-EV

RACS-EV was designed to solve the problem of reconstruction of files for cloud computation. Instead of files being split up into little pieces whole files are stored in a single provider. To allow for erasure coding, files are group by size and the erasure coding is computed on whole files. These groups of files are called object groups. Each object group has exactly 1 file in each provider and one of those files is a erasure coding file.

This solution introduces a whole new set of problems. Before a file was split up and put into every repository and this time a file is in only one repository. So now there needs to be a way to track which repository a file is in. RACS stores this mapping in object groups. Furthermore a file also belongs to an object group so there needs a mapping from file to object group. These two mappings are stored in ZooKeeper. In order to a PUT operation, RACS needs to do a few addition actions. First, the file needs to find a free object group to join (making one if needed). Then the object group's erasure coding needs to be updated which requires downloading the erasure file from the cloud first and re-uploading it afterwards. The file may have already been uploaded in the past, so now some old stale mappings exist which need to be cleaned up. The old file has to be downloaded and then deleted from the cloud and the

old erasure file needs to be downloaded to computer a new erasure and then re-uploaded. On top of all these things that need to be updated, the RACS server could lose connection at any second and leave data in an inconsistent state. For example, RACS could update the object group to include a new file, but then lose connection during the actual uploading of either the erasure file or the file itself.

RACS has typically used distributed locks to help deal with inconsistency issues, but locks don't help all that much. A ZooKeeper lock can be lost at any point during execution meaning that any changes being made could be conflicting with another RACS server that has now obtained the lock.

Python, while a powerful simple language, isn't naturally suited for RACS. Python doesn't allow parts of the same program to run concurrently which inhibits the ability to compute multiple erasure files at the same time. Python also generally runs CPU bound (computation tasks) tasks slower than most languages. (Of course, python can be augmented with native C libraries, but then again so can other faster languages)

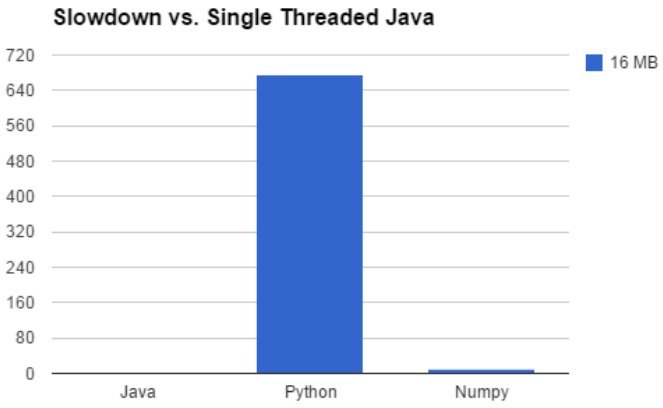


Figure 4: Shows the slowdown of Python vs. Java in computing the bitwise Xor (an operation used in erasure coding) of two 16 MB files. Python is nearly 700x slower. Numpy, a third party extension for Python, gets only a 10x slow down. With larger file sizes, Python quickly runs out of memory.

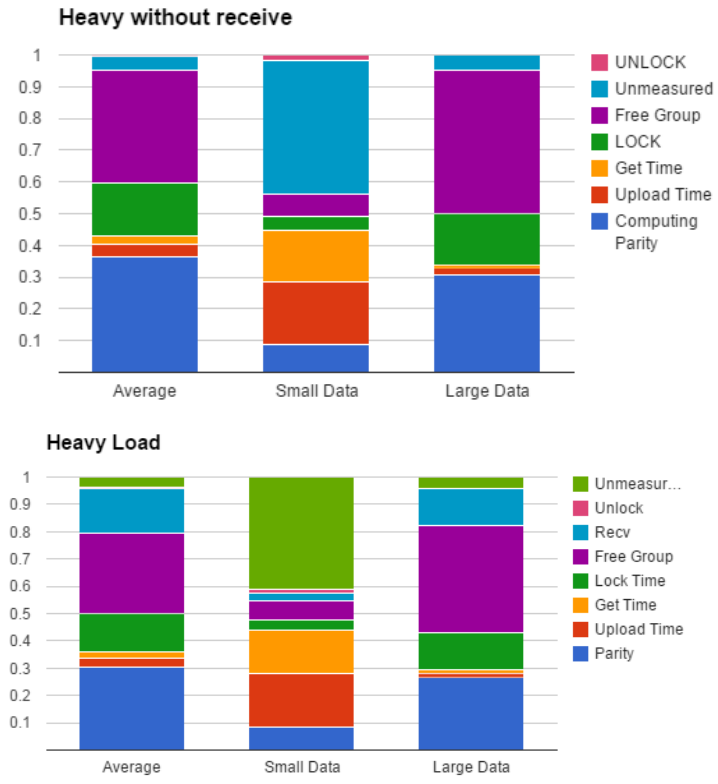


Figure 5 (bottom): This is RACS being run on small EC2 instance at the highest load it could handle before running out of memory. A large chunk of time turned out to be waiting fully for the file to be received before actually doing any work (RECV).
Racs used to do this:
data = socket.recv()
efile = socket2.recv()
compute_erasure(data, efile)
It is more efficient to compute the data as it becomes available:
while(more data)
data = socket.recv(4096) # read in chunks
efile = socket.recv(4096)
erasure_file += compute_erasure(data, efile)
(Various chunk sizes could be experimented with)

Figure 5 (top): Since I believed the receive time could be masked, the top graph shows where the large chunks of time were going. Computing the parity (erasure file) and trying to find an object group with a free slot ate up huge amounts of time, so these are the areas I targeted with RACS-Java.
The unmeasured time for small data, is mostly that of system time (context switch, page faults).

2.2 RACS-JAVA

RACS Java sets out to fix a few things with RACS-EV relating mostly to correctness and speed.

Locks:

Locks can be lost at any time during execution. In order to update meta-data relating to a key a versioning system must be used. The system is detailed in Figure 6. Anytime we wish to update the object group for the key to object group mapping we must use this versioning scheme to have coherent data.

Objectgroup Freelist:

Finding a free group (group with empty slots available for new files) in the previous RACS took a while under heavy loads due to contention of locks. RACS servers all tried to access the same free groups in the same order. RACS Java takes a much more liberal approach. The order in the groups are accessed is random. Unlike RACS Python, when getting the lock on a group tryLock is used instead of

Fault Handling

Eight different things must be updated in a single put operation. Crashing in between any step could result in an unrecoverable error or a very hard to find error. The following is the order in which RACS-Java does updates.

1. Data on cloud
2. Objectgroup
3. Parity File
4. Object group freelist (keeps track of groups with available space)
5. Key to object group mapping
6. Previous key objectgroup (remove key from group)
7. Previous key data on cloud (remove it)
8. Previous parity file

In order explain this process in more depth the steps of a put will be detailed:

1. The Lock is obtained on the file name
2. A free group is found and locked
3. The provider is chosen for the file
4. The intent to write to the specified repo and object group is put into ZooKeeper along with the parity file's unique id and the file's unique id. (each instance of the same parity file and file name have unique IDs)
5. The file is uploaded to the cloud with a unique key. If anything goes wrong, no data will be overwritten and worse there will be an orphaned file (file on the cloud without a ZooKeeper mapping)
6. New erasure is computed and uploaded with a unique name.
7. The updated object group is put into Zookeeper using the versioning system.
8. The object group is placed back on the free list.
9. Atomically (all or nothing) – The intent to write is deleted. The intent to delete the old key is written. (A backup thread sees this and simply takes care of deleting later. Since the intent is on Zookeeper another less busy RACS server could do the cleanup.
10. Release the lock.

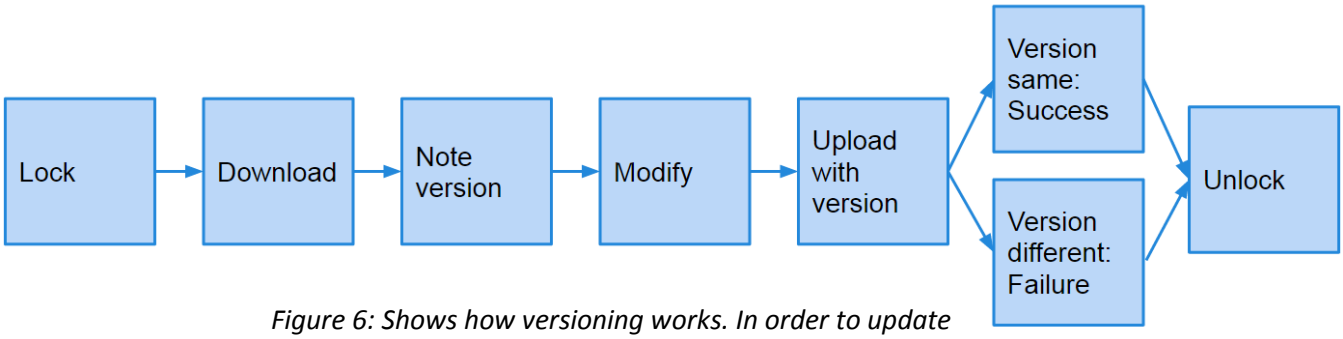


Figure 6: Shows how versioning works. In order to update mappings relating to files in Zookeeper, we must obtain the lock on the mapping, download the mapping, record the version, modify the mapping, upload the mapping with the version, if the version is the same, we still have the lock, otherwise someone else may have the lock.

Failures can be handled at each step with relative ease. Fail after:

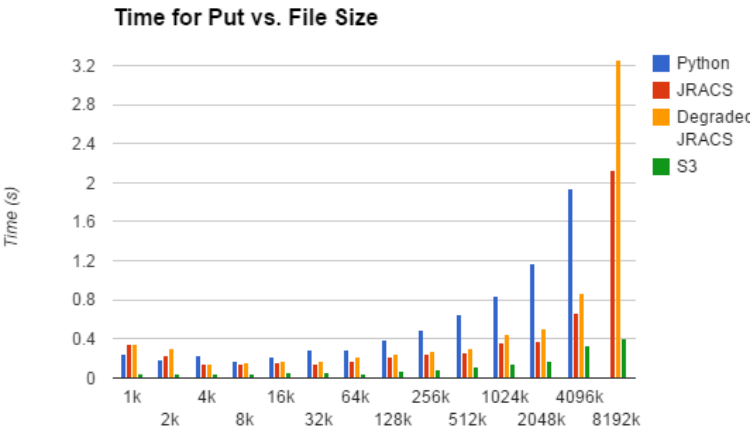
1. The lock is lost and nothing could go wrong.
2. The lock is lost and nothing could go wrong since the group remains on the free list.
3. This is all local so nothing can go wrong.
4. The intent is written to Zookeeper (with a unique key so no intents can be overwritten).
5. If a failure happens hear or later the cleanup thread will simply lock on the key and group abd check the erasure file to see if it has mapping for the file name (erasure files duplicate the object mapping and keep track of the file's length). If it does the erasure is recomputed and uploaded (or deleted if the object group has a different erasure file unique ID). Then the cleaner must check to see if the object group in the intent file has the mapping to the file name (remember each instance of a file name has a unique ID, so a file with the same name can't be mistaken for the failed file). Then the object group is updated (using the versioning system). Lastly, since the erasure file has been updated we can safely remove the file from the cloud (if we removed the file first, we couldn't remove the changes made to the erasure file. After all of this is complete the original intent to write is deleted. If the cleaner fails, the intent is still there for another cleaner on another RACS server to investigate.
6. See 5
7. See 5
8. See 5
9. See 5
10. Releasing the lock can fail due to the versioning system

Caching Erasure Files

RACS JAVA caches parity files temporarily so that they don't have to be fetched from cloud storage as often. The current system of caching holds free (don't have to be completely free) groups after they have been used for roughly 5 seconds. If the cached group is selected for use a lock is obtained (using tryLock – if it fails you can assume the local cached group is invalid) and then the version of the cached object group is compared to the version of the global copy. If they are the same the object group is useable. The cache right now has a 10% hit ratio while yielding small speedups (5%) in the computation of cache groups vs. non cached groups. I suspect this speed up would be greater if the providers used in the benchmark were a little more spread out over the world/country since fetching a parity file would take much longer. (see benchmark setup)

Get:

Get work in a similar fashion to RACS-EV. A get is first tried without a lock and then with a lock. If both of those fail then a degraded get is performed. This means that the erasure file is fetched along with all other files in the object group.



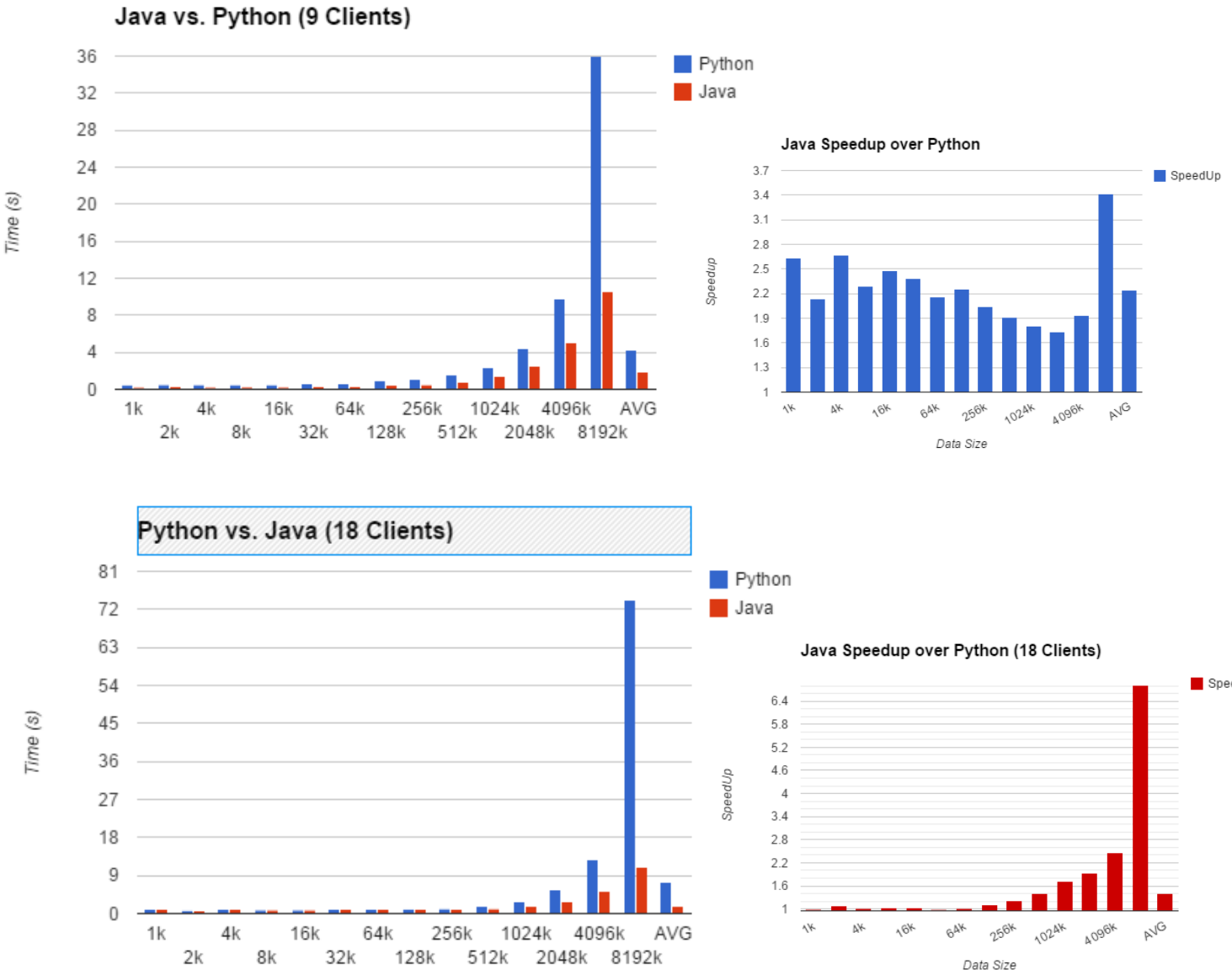
The chart above shows that degraded gets are slower than regular gets. This test repeats the regular GET benchmark but with a deleted repository (so roughly 2/3 of the GET operations are still non degraded) This is expected since more data has to be fetched. The graph probably doesn't show how much slower degraded gets would be in practice since once again fetches from the cloud are fast since the cloud storage and the computation computers running RACS servers are all in the same geographical area and the amount of RACS servers was only 3. So there were at most 2 things to fetch.

PUTS

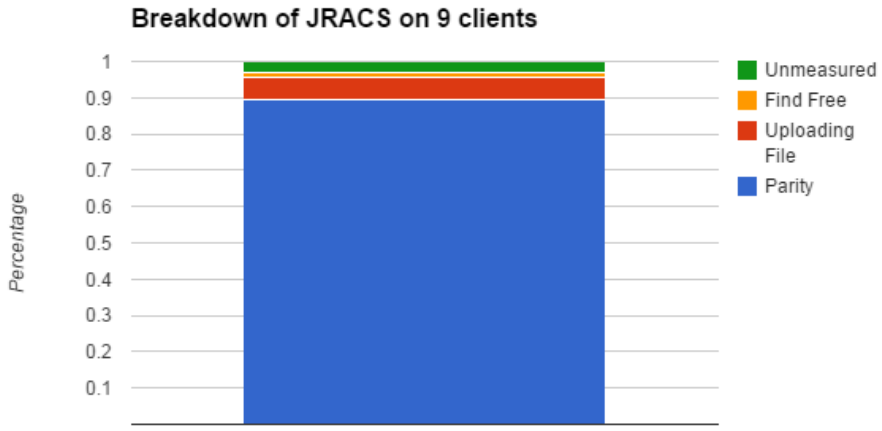
PUTS is extremely similar to put, but it allows multiple files to be uploaded in the same transaction. Because of this, the erasure file is guaranteed to be generated completely locally (no fetches or uploads in between in each additional file). The files intents are

atomically deleted so the files either all fail or all succeed.

3. Results



This figures show that JRacs can obtain a nice speedup over the python version especially for large files



Finding Free is the time spent looking for a group for the file. This now takes nearly no time compared to the overall time.

Uploading File is the time spent actually uploading the file.

Parity includes calling recv on the socket (these parts are sort of inseparable now), computing the parity, and uploading the parity. This is the dominating factor which is should be. This shows that the Java version is being more efficient with the other parts of the program (non-parity parts).

Set up:

3 client computers in CSUG running Intel i7-4770 with 4 cores and 2 threads per core at 3.40 GHZ and 12 GB Ram. Each client computer can then run multiple threads. In the cases of the benchmark each ran an equal number of threads.

3 virtual machines running on us-east EC2 on m1.large instances. (2 cores. 2 threads/core at 1.7 GHZ, 8 GB RAM)
3 Repositories on S3 us-east1.

The client threads each send 20 files of each size with some overlapping file names to create conflicts. The clients send one request at a time and wait for a response before going on.

Conclusion

Java RACS provides a nice speedup over the python version mainly due to its ability to compute the parity faster and actually run multithreaded. My goal in this project was to increase the performance and safety of RACS and I believe I have accomplished that goal. Other discussions include how cost effective RACS actually is. Transfer fees are the killing cost and RACS requires many more transfers than direct cloud usage. A PUT could potentially incur a charge for downloading the erasure file from another provider. On top of this a GET could also become two transfer fees since providers normally charge for data leaving the computation computer as well as the cloud storage. First the GET has to download the file from another provider and then send it out. The other future topic for RACS is getting the computation side of it working. The ability to send code to a cloud computer that would only operate on files in the same provider could provide to be quite fast and cost effective.

Other groups have done things similar to RACS. But they don't focus on the reliability, cost, and potential cloud computation part of RACS. They normally focus on performance, which I think RACS has drifted away from since the file striping that the original RACS did probably could provide more performance. One such idea is commercial product:
https://www.multcloud.com/