Parallelized Distributed Storage System

1st Arjun Pherwani
PM, Programmer
UCF CS Dept.
Orlando, USA
arjunp@knights.ucf.edu

2nd Cameron Cuff
Programmer
UCF CS Dept.
Orlando, USA
ctcuff@knights.ucf.edu

2nd Joseph Giordano
Programmer
UCF CS Dept.
Orlando, USA
joseph.giordano@knights.ucf.edu

2nd Dylan Skelly
Programmer
UCF CS Dept.
Orlando, USA
dskelly@knights.ucf.edu

***Abstract*—This document is the implementation details and design of the distributed storage system worked on by our team.**

***Index Terms*—distributed storage, encryption, parallelization**

I. Introduction

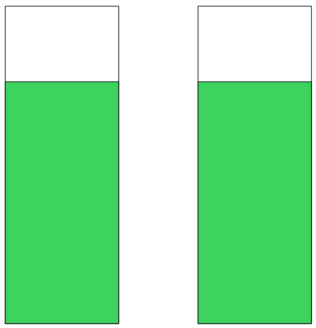
A. The Problem

When thinking of storage we often think of it in unbroken files and folders. All files that are associated with each other are stored next to each other in folders. This methodology works quite well for humans because this is how we stored information prior to the advent of computing, however, this methodology breaks when we try to extend it to the cloud.

To give the reader an example, let's take a hypothetical storage provider, Storage Corp. This company has some existing storage infrastructure which is the servers they have running on premises. For simplicity sake lets say their infrastructure is limited to 2 drives each being 1 TB in size.

Now, they have 3 large files to store. Each file is 600 GB in size. Since the three files will collectively be 1.8 TB in space and we have 2 TB of total space, on the surface this is easy to solve. Let's try to do that right now. The first file is 600 GB, and goes to the first drives. Since 1 TB is greater than 600 GB we can easily store that and have room to spare. 400 GB of room to be precise.

Fig. 1. Two drives that still have capacity but cannot be used in the traditional way.



Now onto the second large file. This file is 600 GB large also, but the first drive only has 400 GB of capacity. No problem, we can store that in the second disk. This disk has ample space. After storing the second file, we now have 400 GB of capacity each in our two drives. This sums up to 800 GB which is more than enough to store a 600 GB large file. However there is a slight hiccup. We don't have enough space in each of the drives to individually hold the third file (as shown in Fig. 1).

B. The Solution

We plan to solve this problem for our project. Now to be clear, this is a solved problem. It's quite easy to solve in fact. Storage Corp should just take the large file, break it into smaller chunks and store those. This way there is no wasted space. Our project was inspired by MongoDB's GridFS system. What they do is break larger files into smaller chunks which are easy to store and then just assemble those at runtime later and return the complete file to the user.

While this is great, we wanted to explore this problem ourselves. For one, because we couldn't find the source code to GridFS around we figured what they did must be a proprietary solution that we cannot see. Because we were interested in the mechanics of how they made this work we decided to come up with our own solution from scratch to do this. Below are the implementation details of our solution.

II. Implementation

At the most basic level our project will be a command line tool. The idea is that the user points it in the direction of a file (for the scope of this project we are restricting the file type to .txt) and the tool breaks the file apart into different chunks, encrypts those chunks and stores them in an s3 bucket. In doing so, it creates a record of the location of the chunks, how to decrypt those chunks and the order in which the chunks should be placed to access them.

Of course, this by itself is not parallel or multithreaded in any way whatsoever. Neither is the above paragraph detail enough to really explain what we are doing so in the below subsections we will be going into more detail on that.

A. Chunking

The process of "chunking" involves reading a file into memory and splitting it into smaller pieces that are easier to manage. The size of each chunk is determined by the user. As an example, if we define a chunks size C (such that $C \in \mathbb{N}$), a file that is N bytes will be split into $\lfloor \frac{N}{C} \rfloor$ chunks. It should

be noted that the last chunk may be significantly smaller than the others if $\frac{N}{C}$ yields a noticeable remainder.

Because each chunk is encrypted (a topic mentioned in a later section), every chunk isn't guaranteed to be exactly of size C. This is due to the fact that the encryption method used is not a one-to-one character mapping. Encrypting one singe character may instead yield two characters.

After the encryption step, each chunk is uploaded to an AWS bucket. Chunks are given a random name generated with a UUID library. Each chunk is prefixed with the order in which it was split. This makes it easier to piece chunks back together in a later process. As a final step, a JSON file is saved to user's computer that contains instructions on hho to piece together each encrypted chunk. This JSON file (aptly named *keystore.json*) holds the master encryption key along with the nonce keys used to encrypt each chunk.

The benefit of file chunking is the potential space saved from breaking the file into pieces. If an application uses multiple databases and one database has a small amount of storage available, a file that exceeds that limit won't be able to fit in that space. However, we'd now have unused and potentially wasted space. When we take a large file and split it into many smaller pieces, we can utilize this space across many databases.

B. Encryption / Decryption

As mentioned in an earlier section, each file chunk is encrypted with the ChaCha20-Poly1305 cipher. ChaCha20-Poly1305 is an AEAD (Authenticated Encryption with Additional Data) cipher that combines the ChaCha20 stream cipher with the Poly1305 message authentication code.

ChaCha20 is a stream cipher that takes a secret key and some sort of nonce value (a value that cannot be used more than once for the same key) and generates a stream of deterministic random bits called the *keystream*. The primary purpose of the keystream is to XOR it with plaintext to produce ciphertext. Because the plaintext doesn't need to be known in advance to generate the stream, this approach allows the ChaCha20 algorithm to be very efficient and easily parallelizable. Additionally, this works fell when implemented with file splitting. This means that smaller blocks of text can be encrypted in parallel because the ChaCha20 cipher doesn't need to know what it's encrypting.

Published in 2004, Poly1305 is a cryptographic message authentication code (MAC) that can be used with and encrypted or unencrypted message to generate a keyed authentication token. This token is used to guarantee the integrity and validity of a given message. Poly1305 is extremely high speed and can take advantage of additional hardware to reduce latency for long message. Similar to ChaCha20, this means Poly1305 can easily be parallelizable.

The general process for file encryption is as follows:

1) Generate a master encryption key using a UUID generation function. This master key must be 32 bytes.
2) For every file chunk, generate a random nonce key using a UUID generation function. This nonce key must be 24 bytes and, along with the master encryption key, will be used to encrypt this chunk using the ChaCha20-Poly1305 algorithm.
3) Create a JSON file (named *keystore.json*) that contains the encryption key and nonce keys used to encrypt each file chunk.

The general process for file encryption is as follows:

1) Load the keystore JSON file into memory and read the master encryption key
2) Take each file chunk and sort them alphanumerically from 0 to N, where N is the total number of chunks.
3) For every encrypted file chunk, use the master encryption key and associated nonce key to decrypt the file chunk into a plain text string.
4) Once all chunks have been decrypted, combine each decrypted string into one large string to form the original plaintext.

C. Parallelization

We have identified potential tasks that we can hand off to other threads which would parallelize the code. It goes without saying that the more work that can be done in parallel the faster our code will run. Rust has great support for concurrency and we hope to leverage this to achieve the maximum speedup possible. x Because of the nature of the project, theoretically, once a chunk has been created everything about it can be handed off to a thread to take care of. This can go right from encryption, to sending the item off to the bucket. On this flip side, during the retrieval process, we can potentially parallelize the retrieval from the S3 bucket all the way to decryption. At the moment, our code is completely sequential and we are working on parallelizing the aforementioned tasks.

The parallelized storage and retrieval algorithm is described below.

Algorithm 1 Parallelized Bucket Storage

procedure BUCKETWRITE(i)
    $c \leftarrow Creds$
    if b.empty() **then**
        $b \leftarrow Bucket$
    else
        $Threads$[i] $\leftarrow b$

D. Storage and Retrieval

In order to simulate the experience of storing and retrieving from a file store we decided to use AWS S3 buckets. There are a couple of reasons for this, the first and most important being that this project was intended to be an into how a file can be split apart into smaller and more manageable fragments (chunks) and how we could parallelize that. By that logic it

didn't seem worth it to try to figure out a complicated way of storing it.

On the other hand we also wanted something that was non-trivial. By this we mean that we did not want to just paste the files in local storage.

For those reasons we decided we wanted to go with some cloud storage option. But there were still other options like Azure Blob Storage. Unfortunately we could not use blob storage because it does not have an officially supported Rust SDK. S3 on the other hand seemed to have promising support which is why we picked AWS S3 buckets. Additionally, S3 allows for large PUT operations, all the way up to 5GB, making it capable of testing sizeable file inputs, and storing up to 50TB in one S3 instance. Another benefit is the ability to pay as we go, with assurance that large tests will not be interrupted even in during significant spikes in network traffic. The reliability and availability of S3 storage makes it a perfect platform for facilitating our storage system and developing to scale.

The rust-s3 library simplifies interactions with Amazon S3 buckets with built in asynchronous API features, which can be deactivated if needed. Full suppport for operations including put, get, and delete allows for cleaner Rust code. In terms of file security, the methods presign_get and presign_put allows us to upload files to a bucket, and selectively distribute the link to team members without concern for files being visible as they are stored at a presigned URL.

One of the rust-s3 library's default features is Tokio. The Tokio asynchronous runtime for Rust provides flexible, thread-safe, and performant APIs for storing and retrieving objects in our buckets. GET and PUT functions for objects and object streams with Tokio methods are tokio::io::AsyncWriteExt and tokio::io::AsyncReadExt. The Tokio runtime harnesses a work-stealing scheduler, a multithreaded scheduler in which a processor has an assigned thread queue with their own sequential instructions. However, what distinguishes the work-stealing scheduler is the fact that threads can spawn new tasks that can be parallelized with the code currently being executed. As the new tasks are placed on the processor's queue. When the queue is empty, a processor examines other queues in an effort to adopt extra tasks, effectively dealing with idle processors by balancing the excess workload spawned by other threads. This is where the strategy gets its name, by *stealing* and distributing work until it is all dealt with by highly parallelized task execution. This can be particularly useful in our application, as chunking for multiple files may imply uneven workloads by forking more or less threads across different processors, especially when working with larger files, or scaling the application in the future.

E. Why Rust?

We picked Rust primarily because we all wanted learn it and now seemed like a great time to try it out. On a language level Rust has an excellent dependency management solution, Cargo. Another thing we like about Rust is that it's strongly typed but it infers the types from assignment. Finally, we wanted our code to be as performant as possible.

All that being said because Rust does a lot of unique things with it's concept of ownership we constantly find ourselves wrestling with the language to get any work done. Tools like IntelliSense and TabNine do help speed up our development time.

F. Testing

We believe that to write code of high quality testing is a requirement. This is why we are in the process of writing tests for our code. While this isn't a requirement and neither is it a priority for us we still believe we should do it and are doing it.

III. Appendix

A. Team Management and Organization

We are a fairly small team of busy people. So while the team size makes it easy to organize, the schedules do not.

A notable point is that our team is not using the Agile methodology because all the members in our team have a strong distaste for it. Instead we are using a pseudo-waterfall methodology where some of our members get together for a short period of time and get as much work done as possible.

We believe that this makes our work easier because there is less overhead of having to maintain the standard Agile artifacts like Kanban boards and have regularly scheduled stand-ups.

We are using GitHub for version control and Discord for communication.

B. Current Progress

At the moment we are completely done with a sequential version of the command line app. This means that we can and take a file, break it into several chunks, encrypt those chunks, write the encryption key and other relevant information to a shared document and send these an AWS S3 bucket.

Over the next few weeks our efforts are going to be focused on parallelizing the work we have already done.

IV. Challenges Faced

Rust has been for the large part the single biggest challenge we have faced. This is primarily because all of us are new to the language and we constantly find ourselves wrestling with the language to get any kind of progress. One of our team members has described the experience as "learning to program all over again".

Another issue we have found is the abundance of out of date documentation. This is probably because Rust is a very new language and so it's ecosystem is rapidly evolving.

Acknowledgment

Thanks to Professor Dechev for allowing us to do this project. We realize that it is a bit far removed from the usual flavor of projects submitted for this class.

Thanks also to the Rust community of Discord, StackOverflow and other mediums for either answering our questions

or having answered other people's questions that were very similar to our own.

A debt of gratitude is also owed to all the developers who contributed to the open source Rust crates that we used in our project.

REFERENCES

[1] A. Langley, W.-T. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, The ChaCha Stream Cipher for Transport Layer Security, 24-Jan-2014. [Online]. Available: https://tools.ietf.org/id/draft-mavrogiannopoulos-chacha-tls-01.html.

[2] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," Document search and retrieval page, May-2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7539.

[3] https://tokio.rs/.

[4] https://crates.io/crates/rust-s3.

[5] https://aws.amazon.com/s3/faqs/.