Implementation Of A Distributed Diskless Filesystem By Exploitation Of Popular Network Protocols

By Yuri Grigorian
Supervised by Prof. Dana Shapira

https://github.com/yurig93b/exploited-disks-final-project

Version	Date	Owner	Reason
1.0	17.08.2023	Yuri Grigorian	Release
0.1	19.04.2023	Yuri Grigorian	Initial Draft

Intro

In the early days of computer networking, numerous protocols were developed. However, some of these protocols were never maintained or upgraded to address their vulnerabilities. In this project/paper, our objective is to create a distributed file system that is storage efficient exploiting those protocols. To achieve this, we plan to identify an outdated protocol and use its structure for our benefit. Since the file system's performance is dependent on the network, it is vital to implement it in a way that optimizes network bandwidth usage and storage efficiency. One approach to maximizing storage efficiency is through data deduplication, which this project will implement using two methods: fixed block size data deduplication and variable block size data deduplication. The project will also compare the effectiveness of these deduplication methods and evaluate the implemented scheduling algorithms.

The objective of the project is to identify a network protocol that can be utilized to store payloads on distributed hosts, switches, routes, and other similar devices. The aim is to identify hosts that can be exploited to store the maximum amount of payload for the longest possible time using the identified protocol. Throughout the project, the file system server will not save any data locally, instead relying exclusively on responses received from exploited hosts at regular intervals to serve the requested information in the file system.

A Candidate Protocol - ICMP

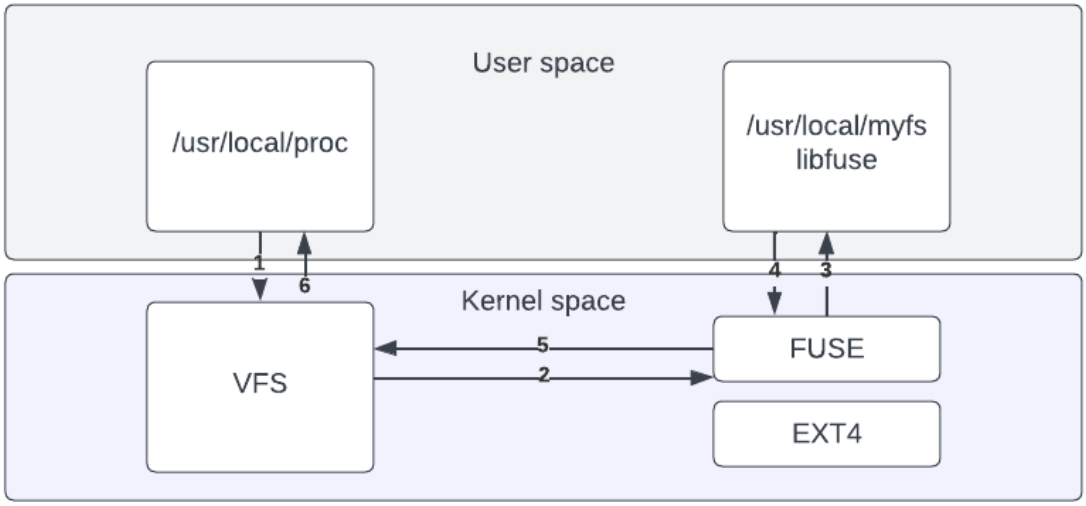
The Internet Control Message Protocol (ICMP) is utilized by network devices, including routers, as an error-reporting protocol. When network issues hinder the delivery of IP packets, ICMP generates error messages that are sent to the source IP address. These messages indicate that a gateway to the internet, such as a router, service, or host, cannot be accessed to deliver the packet. It's worth noting that any IP network device can handle ICMP messages by sending, receiving, or processing them.

The ICMP protocol encompasses various message types, and the Echo request type is particularly interesting for our project. Since the ICMP protocol doesn't have a limit on the number of requests that can be made, it's an excellent candidate for our purposes. It's possible to include a payload in a request to a supporting network entity and receive that payload back. By sending the payload, it will travel through all the switches, routers, and gateways on the path to the receiving party. Consequently, we must identify hosts that are as distant as feasible from the sending party to maximize the payload's lifespan in the global network.
Every ICMP request is able to store a payload of approximately 65,527 bytes.

File System Implementation Details

FUSE - Filesystem in userspace

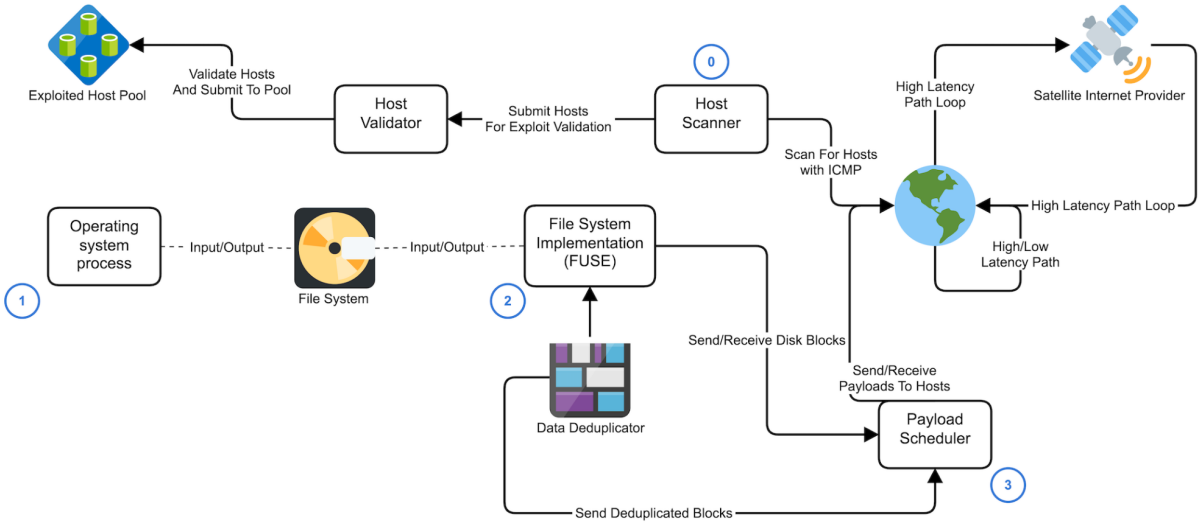
A UNIX system can employ several methods to implement a file system. One approach is to implement a kernel VFS to have the code in kernel space, or to use FUSE, which allows a file system to be implemented in user space utilizing various programming languages. Regardless of the implementation approach, the file system will be perceived by the operating system as a standard file system.



Process to filesystem file access flow.

File System Architecture

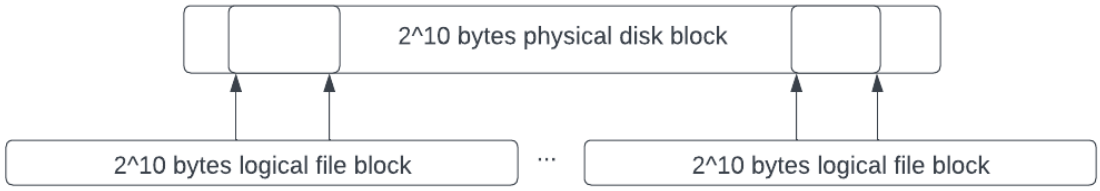
The filesystem will be implemented using FUSE in the Python language. It contains multiple components which are responsible for different functions of the system.



General architecture diagram.

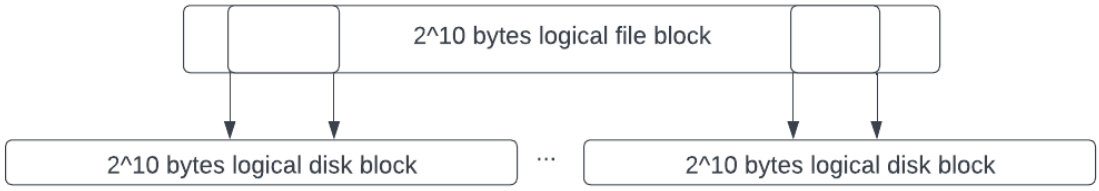
Physical Disk Blocks

The filesystem is based on disk blocks of 1024 as this size fits nicely within an ICMP packet payload limits. The disk blocks contain deduplicated data that is replicated across the different hosts across the globe. It is important to find as many hosts as possible with the “Host Scanner” component. With each change to a disk block, it will be propagated to the rest of the participating hosts. A single physical disk block may be referenced by multiple logical blocks at different offsets.



Logical File Blocks

A logical file block represents a chunk of a file of 1024 bytes. Logical blocks are created as the file grows to accommodate space needs. Upon allocation of a logical block the system will request the physical disk manager to allocate offsets for the logical file block. A logical file block may reference multiple offsets in different physical blocks in order to save as much space as possible at the expense of performance.



Inode object

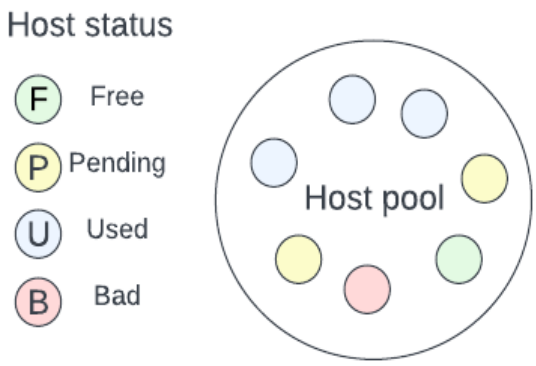
The Inode object represents a file or directory in the system. It filesystem metadata such as permissions. An Inode object references logical blocks that are allocated for it upon writing (lazy allocation).

Local file system

For the purposes of quicker testing, an additional file system was developed using the architecture presented earlier. This filesystem stores the data locally instead of propagating it to hosts worldwide and was used to test the different deduplication strategies only.

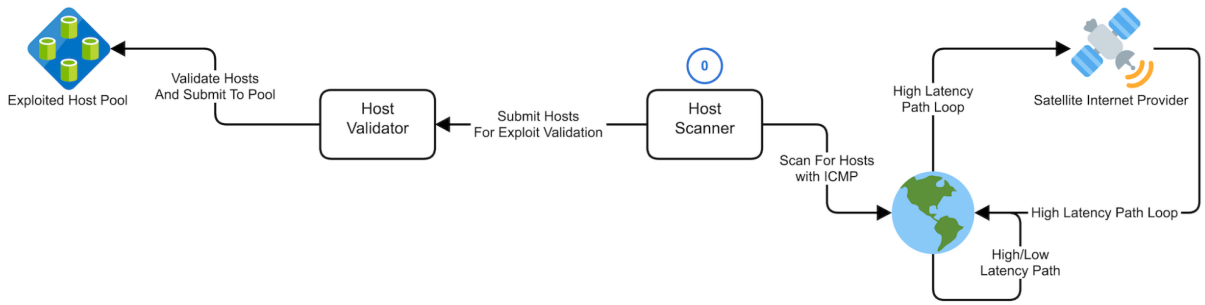
Host Pool

Using the "Host Scanner" component, a repository of both utilized and free hosts will be stored for the use of the file system. For every utilized host, a health record will be maintained. If a certain host exceeds a predetermined error threshold and becomes unstable, it will be substituted with a more robust one. The problematic host will be disregarded by the system and will only be reintegrated into the pool once it has been assessed and deemed stable again by the "Host health validator" component.



Host Scanner

Our goal, as previously stated, is to locate usable hosts for prolonged storage of the payload (disk blocks). This goal can only be accomplished by utilizing hosts that are situated at a great distance from our filesystem server. The greater the distance between our server and the targeted hosts, the longer it takes for packets to be transmitted to and from. Hosts that use satellite internet which is located in a Geostationary orbit are also ideal candidates due to their considerable distance from earth, which results in significant latency.
To achieve this, we will scan network CIDRs (Classless Inter-Domain Routing) in countries that are situated farthest from our testing site (Israel) on regular intervals. Hosts that respond to an ICMP Echo request will be submitted to the host pool with a "pending-validation" status. After the "Host health validator" component performs a series of successful health checks, the host will transition to a "free" state, indicating that it can now be utilized for storing payloads.



Host scanner scanning and host submission flow.

Host Health Validator

The maintenance of host status within the pool is handled by the host health validator, which determines whether a host is suitable for our needs and whether it remains healthy after host exploitation has started.
Upon adding a host to the pool with a "pending" status, the validator will conduct a series of health checks. It will calculate the average latency to the host and attempt to send it a specialized payload to detect the maximum payload size that the host can accommodate. The "max-payload-bytes" field will then be recorded on the host's health check record. If the host fails the health checks, it will be marked as "failed."

An example for a health record of a host:


```
{
  "host": "1.1.1.1"
  "avg-latency-ms": 9430,
  "failed-checks-last-hour": 1,
  "failure-threshold": 3,
  "max-payload-bytes": 32767
}
```

Payload Scheduler And Data Replicator

The payload scheduler plays a crucial role in ensuring that our data blocks or payloads are distributed globally and kept alive by scheduling payload packets to be sent to available hosts. Since we do not keep any data locally, we rely on the hosts to return our packets when requested by any process in the system. To achieve this, the file system implements two different scheduling strategies. The first strategy aims to maximize the availability and longevity of our payloads by replicating a data block across three hosts with similar and long latencies, in different countries, with overlapping timing. The second strategy prioritizes minimizing latency at the expense of availability and network bandwidth. It replicates the data block across three hosts, with two being slow hosts, and one being as close as possible to us, thus responding the quickest. It's worth noting that each host can store only one physical disk block in the current file system implementation, but it's generally possible to store more by sending additional packets to the host.

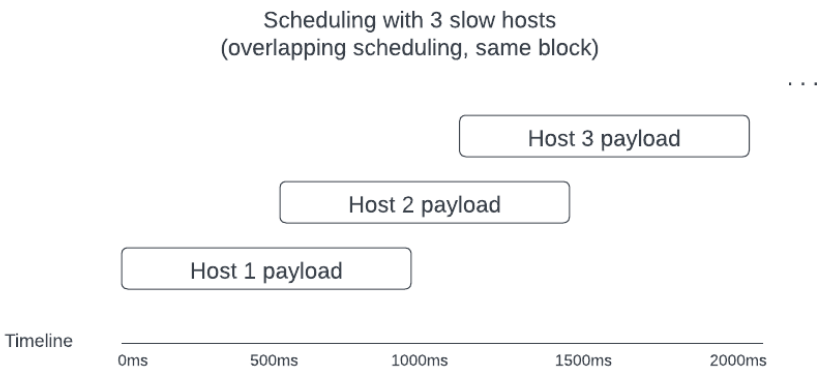


Illustration of overlapping timing scheduling of the first strategy.

Deduplication Implementation Details

Data Deduplicator

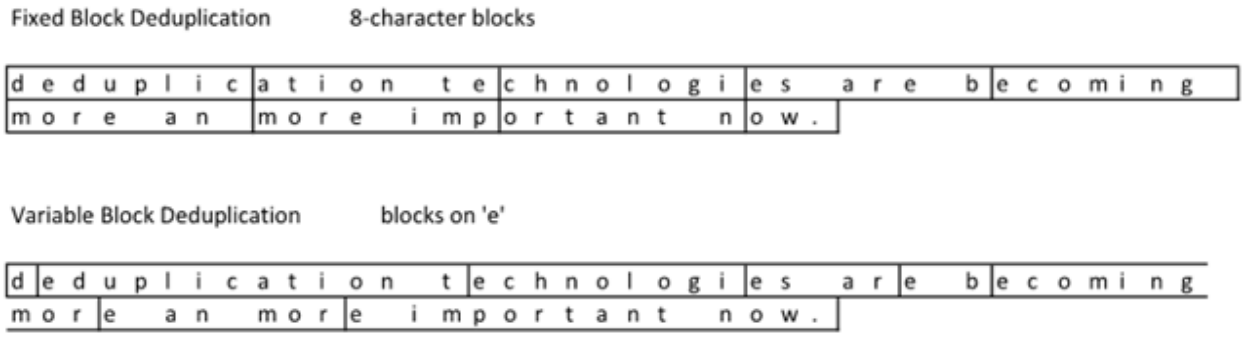
Since our filesystem is dependent on the amount of hosts we will be able to exploit and since this number can be pretty small, we have to employ a data deduplication mechanism. For this file system we will be doing **offline** deduplication using fixed and variable length blocks on regular intervals. The deduplication process will be started at regular intervals on logical blocks that were marked as “changed”. A logical block can be marked as “changed” when we either write a new file or update an existing one. The filesystem will implement a fixed chunk sized deduplication mechanism and a variable length chunks deduplication using a Rabin-Karp rolling hash for content based slicing.

Lifecycle of a logical block

A logical block is created whenever a file tries to write to an offset that this logical block should contain. A logical block of index 0 contains 0-1024 bytes from the beginning of the file. A logical block references one or many physical disk blocks at different offsets. The deduplication process runs on the logical blocks instead of the physical disk blocks. Once a logical block can be deduplicated as a whole (static deduplication) or only parts of it - the free offsets of the physical disk return to the free offsets pool and will be reused once a new logical block needs to be allocated.

logical_block_index = math.floor(offset / logical_block_index)

Static vs. Variable length deduplication



Difference between fixed block and variable block deduplication (To be changed in the future).

With static deduplication we set a fixed window size of X bytes and start hashing every window in the file. A window begins where a previous window ended. The problem with static deduplication is that it fails deduplicating blocks with shifted data. Thus, a text document with a letter inserted in the beginning can cause the whole file to not get deduplicated.

With variable length deduplication we aim to minimize the data shifting problem by slicing the file into variable sized chunks when a specific criteria is met. The implemented filesystem slices the file in a logical block when the N least significant bits of the generated **rolling hash** are all zeroes. It turns out that choosing different N values can cause different chunk sizes to be generated and we can use that for our benefit. A more in depth explanation follows below.

Rabin-Karp Rolling Hash

A rolling hash is a type of hash that can be rolled by adding or subtracting data from/to it in order to generate a new hash. A rolling hash is very efficient as it does not recalculate the whole hash but only the parts that were added and/or removed. A rolling hash can be used as a sliding window hash on a data array.

The Rabin–Karp string search algorithm is often explained using a rolling hash function that only uses multiplications and additions:

$$H = c_1 a^{k-1} + c_2 a^{k-2} + c_3 a^{k-3} + \ldots + c_k a^0,$$

Where **a** is a constant, and **ck** are the input characters. In order to avoid manipulating huge H values, all math is done modulo n. The choice of a and n is critical to get good hashing. In particular, the modulus n is usually a big prime number.

Using probability to target an approximated chunk length

A good hashing function is a function that has uniform distribution on its values. Assuming this is the case with our chosen hash function, we can use probability to target specific chunk sizes. The implemented file system aims to get chunk sizes of **~32 bytes** and will produce a chunk when the N least significant bits of the generated rolling hash value are all zeroes. Assuming our hash function returns a number of 32 bits let's look at the **N** least significant bits of the returned value. For example purposes let's set **N=5**:

Hash value: 0010 0000 111**0 0000**

The probability of the last N bits to be exactly zero are 2^{-N} since there is an equal chance to get 0 or 1 in each of the bits. As this is a geometric distribution the number of tries needed until a first success is 2^{N}. Since a rolling hash sliding is a new experiment, a set of N zeroes will be met approximately every 2^{N} slides. Thus we can use the value of N to approximate the sizes of the generated variable chunks.

It is worth mentioning that the all zeroes pattern can be replaced with any other pattern as the distribution is uniform.

Deduplication time complexity

Total hash generation time: O(n) where n=len(data)
Total hash lookup time: O(1*m) where m is the number of chunks generated.

Overall: O(n+m)

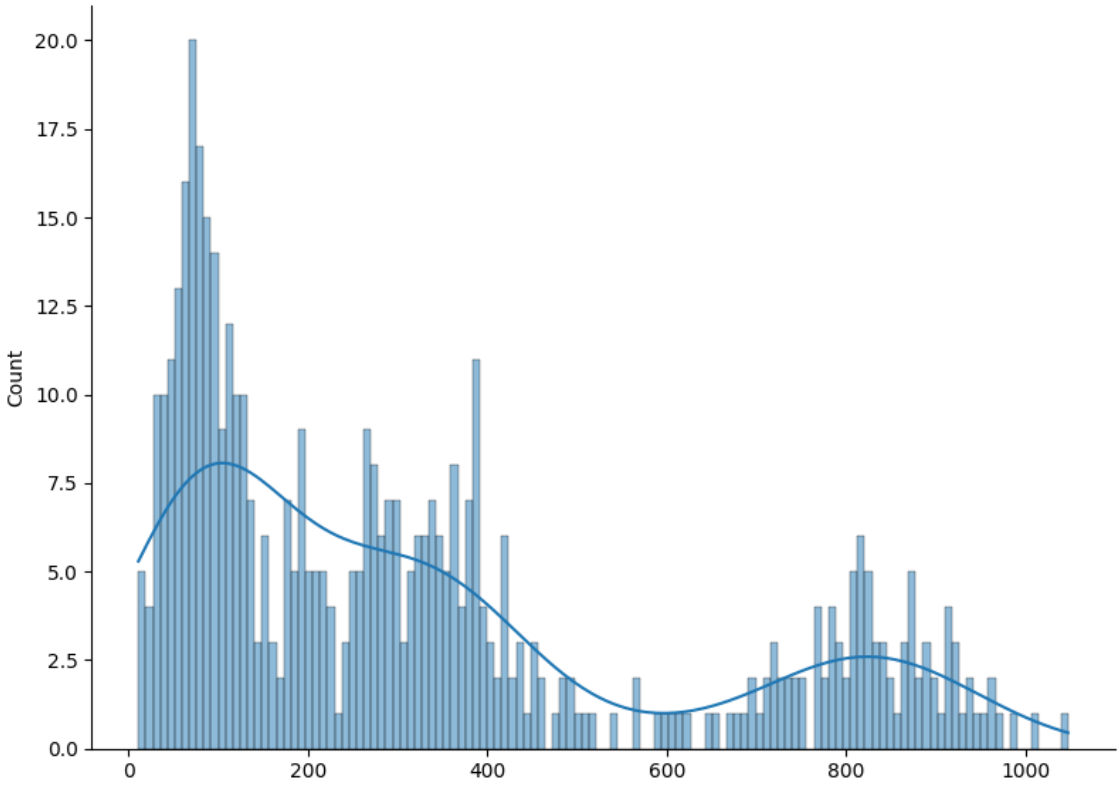
Results

Dataset definition for tests

For the tests the dataset created was:

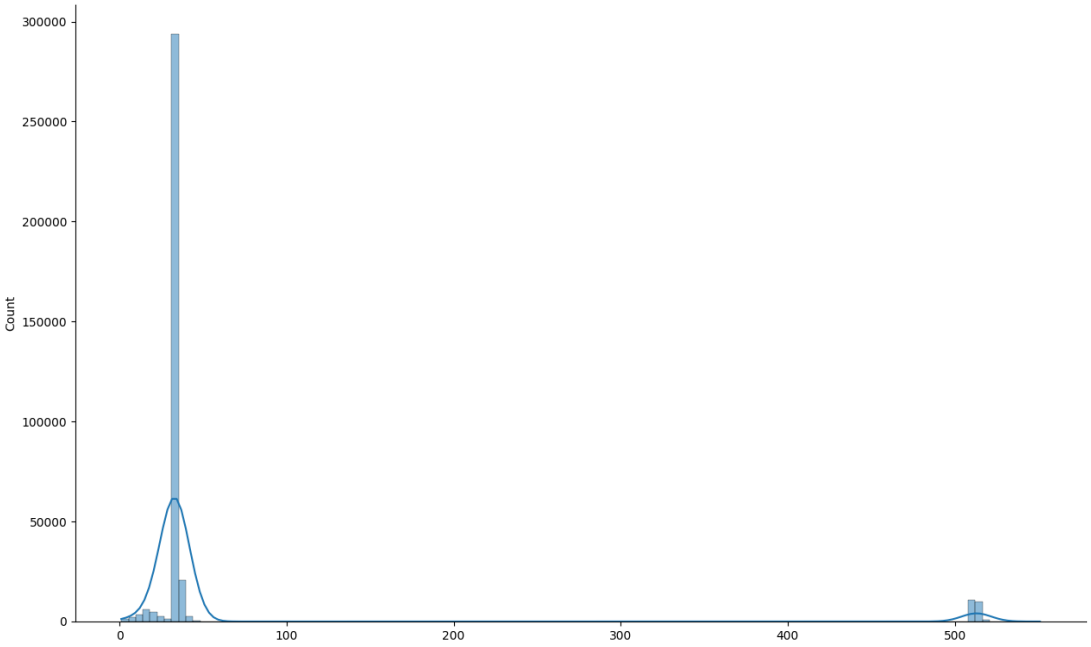
a) A PDF file consisting of presentations of one of the courses in Ariel University **(21.4 MB)**.
b) An Alice in Wonderland text file **(150KB).**
c) A shifted alice in wonderland text file **(150KB)**.

Distribution of latencies for scanned hosts



Min: 11.142, Max: 1047.20, Mean: 325.41

Distribution of chunk sizes for variable sized deduplication



Min: 1, Max: 551, Mean: 60.39

The targeted chunk sizes were ~32-64 bytes and the distribution shows that the majority of chunks generated were in that region. It is worth mentioning that the bigger chunks (>500) were generated because the initial window for the rolling hash is 512 bytes.

Data deduplication strategy comparison

The test results showed that the variable sized deduplication performed significantly better than the static sized deduplication on the dataset used.

File system status with no deduplication:

```
Filesystem 512-blocks     Used Available Capacity   iused    ifree %iused  Mounted on
MyFS         83886080 83886080  83843734    51% 18446744073667629749 41921867  100%   /private/tmp/t5
```

File system status after static deduplication:

```
Filesystem 512-blocks     Used Available Capacity   iused    ifree %iused  Mounted on
MyFS         83886080 83886080  83844138    51% 18446744073667629547 41922069  100%   /private/tmp/t5
```

File system status after variable deduplication:

```
Filesystem 512-blocks     Used Available Capacity   iused    ifree %iused  Mounted on
MyFS         83886080 83886080  83845228    51% 18446744073667629002 41922614  100%   /private/tmp/t5
```

Saved blocks in static deduplication: **404**
Saved blocks in variable deduplication: **1494** (x3.69 better)

Summary

The project showed how simple unprotected network protocols can be used for our own benefit for data storage purposes. The deduplication methods implemented in this project allow the file system to be optimal with regards to space usage and the results support the benefits claimed of variable deduplication over static deduplication.

Bibliography

[1]

https://www.researchgate.net/publication/327062086_Dynamic_determination_of_variable_sizes_of_chunks_in_a_deduplication_system

[2] - https://en.wikipedia.org/wiki/Rolling_hash

[3] - https://www.netapp.com/media/22774-Variable_length-dedup_Giridhar.pdf