UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Gurret: Decentralized data management using subscription-based file attribute propagation

—

Sivert Johansen

INF-3990 Master’s thesis in Computer Science - May 2022

Abstract

Research institutions and funding agencies are increasingly adopting open-data science, where data is freely available or available under some data sharing policy. In addition to making publication efforts easier, open data science also promotes collaborative work using data from various sources around the world.

While the research datasets are often static and immutable, the *metadata* of a file can be ever-changing. For researchers who frequently work with metadata, accessing the latest version may be essential. However, this is not trivial in a distributed environment where multiple people access the same file. We hypothesize that the publisher-subscriber model is a useful abstraction to achieve this system.

To this, we present Gurret: a distributed system for open science that uses a publisher-subscriber based substrate to propagate metadata updates to client machines. Gurret offers a transparent system infrastructure that lets users subscribe to metadata, configure update frequencies, and define custom metadata to create data policies. Additionally, Gurret tracks information flow inside a filesystem container to prevent data leakage and policy violations. Our evaluations show that Gurret has minimal overhead for small to medium-sized files and that Gurret can support hundreds of custom metadata without losing transparency.

Contents

Abstract i

List of Figures vii

List of Tables ix

List of definitions xi

1 Introduction 1
- 1.1 Thesis Statement 3
- 1.2 Scope and Limitations 4
- 1.3 Context 4
- 1.4 Methodology 4
- 1.5 Outline 5

2 Background 7
- 2.1 Filesystem 7
- 2.2 XFS 8
- 2.3 Btrfs 9
- 2.4 F2FS 10
- 2.5 ext4 11
  - 2.5.1 ext4 Structure 11
  - 2.5.2 Extent Trees 13
  - 2.5.3 Inline Data 15
  - 2.5.4 Directories 15
  - 2.5.5 Journaling 15
- 2.6 FUSE 16
- 2.7 Access Control 17
- 2.8 Discretionary Access Control 17
  - 2.8.1 Access-Control List 17
  - 2.8.2 Capability-Based Access Control 18
- 2.9 Mandatory Access Control 18
  - 2.9.1 Lattice 19
  - 2.9.2 The Bell-LaPadula Model 20

2.9.3 The Biba Model 20
2.9.4 Access-Control Limitations 21
2.10 Information Flow Control 21
2.10.1 Information Flow 21
2.10.2 Flow Control 22
2.11 Summary 23

3 The Gurret System 25
3.1 Architecture 25
3.2 Filesystem Container 26
3.3 Filesystem Structure 27
3.4 Storing Metadata 28
3.5 Gurret Daemon 28
3.6 Message Structure 29
3.7 Summary 29

4 Metadata 31
4.1 Metadata Interface 31
4.2 Standard Metadata 32
4.3 Custom Metadata 32
4.3.1 Meta-Code Requirements 33
4.4 Creating Data Management Policies 35
4.5 Metadata Update 36
4.5.1 Receiving Updates 36
4.5.2 Sending Updates 36
4.6 Summary 37

5 Metadata Subscriptions 39
5.1 Subscription Language 40
5.2 Metadata-Field Querying 41
5.3 Sending Subscription Messages to the Backend 42
5.3.1 MetadataUpdate 42
5.3.2 GetSubscriptions 42
5.3.3 SetSubscriptions 43
5.4 How End-Users Subscribe to Metadata 44
5.5 Metadata Consistency Model 44
5.6 Summary 45

6 Data-Sharing and Notification Engine 47
6.1 Data-sharing service 47
6.2 Notification Engine 48
6.2.1 Socket Server 48
6.2.2 Metadata Store 48
6.2.3 Query Engine 49

6.3 Summary 52

7 Taint Tracking 53
7.1 Taint Tracking 54
7.2 Taint Tracking Rule 54
7.3 Implementing the Taint Tracking Rule 56
7.4 Tracking Data Lineage 58
7.5 Forest of Trees 58
7.6 Intercepting Information Flow Related Syscalls 60
7.7 Storing the Forest of Trees and Broker 62
7.8 Taint Tracking Weaknesses 62
7.9 Implementing an Information Flow Control 62
7.10 Labels 62
7.11 Access-Control Rules 63
7.12 File Removal Policy 63
7.13 Information Flow Control System Use Cases 64
7.14 Summary 64

8 Evaluation 65
8.1 Hardware Specification 65
8.1.1 Benchmarking Method 66
8.2 FUSE Overhead 66
8.2.1 Setup 66
8.2.2 Results 66
8.3 Metadata Benchmarks 68
8.3.1 Setup 68
8.3.2 Checking Metadata Result 69
8.3.3 Updating Metadata Result 70
8.4 Filesystem Benchmark 70
8.4.1 Benchmarking Setup 71
8.4.2 File Benchmark Result 71
8.4.3 Extended Attributes Result 72
8.5 Filesystem Benchmark Result 72
8.6 File Subscription Benchmarks 74
8.6.1 Benchmarking Setup 74
8.6.2 Benchmarking Result 75
8.7 Summary 75

9 Conclusion 77
9.1 Related Work 77
9.1.1 Distributed Data Management 77
9.1.2 Filesystems 78
9.1.3 Publisher-Subscriber Model 79
9.1.4 Information Flow Control Systems 79

9.2 Future Work 79
9.2.1 Filesystem Container 79
9.2.2 Filesystem 80
9.2.3 False-Positive Information Flows 80
9.2.4 Gurret Daemon 80
9.2.5 Distinguishing between Data Management Metadata . 81
9.2.6 Distributed Data-Sharing and Notification Service . . 81
9.3 Concluding Remarks 82

Bibliography **83**

List of Figures

2.1 Output from dumpe2fs. A lot of lines are omitted for clarity. . 12
2.2 Overview of the inode block pointers. Tripley indirect pointer is omitted. 14
2.3 The FUSE architecture. 16
2.4 Linear lattice with the values 1 - 4. 19
2.5 A more complicated lattice. 19
2.6 no-read-up, no-write-down rules. 20
2.7 no-read-down, no-write-up rules. 21

3.1 Overview of the Gurret system. 26
3.2 Creating and mounting a loop device containg ext4. 27
3.3 The pre-arranged filesystem structure. 27
3.4 Example metadata for a file. 28
3.5 The structure of a message. 29

4.1 Access metadata implementation. 33
4.2 Custom Metadata hierarchy. 34
4.3 Implementation of the FiveDays data policy 35
4.4 Algorithm for receiving metadata updates. 36
4.5 Algorithm for receiving metadata updates. 37
4.6 Pseudo-code for detecting metadata updates. 37

5.1 The Document structure of subscription messages. 40
5.2 Example message of the metadata update type. 42
5.3 Typical GetSubscriptions message. 43
5.4 Typical GetSubscriptions response message. 43
5.5 Basic SetSubscriptions message. 44

6.1 SetSubscriptions with three users 50
6.2 Hierarchy of async tasks idling 51
6.3 Hierarchy of async tasks on update 52

7.1 Architecture for handling flow transactions. 56
7.2 Lookup-table for data management metadata. 57

7.3 Forest of trees structure. 59
7.4 Simplified structure of the forest of trees. 59

8.1 FUSE overead reading. 67
8.2 FUSE overead writing. 67
8.3 FUSE overhead writing, logarithmic y scale. 68
8.4 Metadata checking time. 69
8.5 Custom metadata scaling with check store. 70
8.6 Metadata update time. 71
8.7 Filesystem writing benchmark. 72
8.8 Filesystem reading benchmark. 73
8.9 Filesystem extended attributes writing benchmark. 73
8.10 Filesystem extended attributes reading benchmark. 74
8.11 Time to check metadata for files that subscribes to *n* metadata. 75

List of Tables

2.1 Disk layout of ext4. 11

5.1 Avalable comparison operators. 42

6.1 Table representing Figure 6.1 51

List of definitions

7.1

Let $\longrightarrow$ be an explicit flow between two files

. . . . 54

7.2

Let $M(x)$ be the set of management metadata of x

. . . . 54

7.3

Let $\rightsquigarrow$ be the transfer of management metadata between two files

. . . . 54

7.4

$A \longrightarrow B \wedge A \rightsquigarrow B \Leftrightarrow$ flow transaction from A to B
For two files A and B

. . . . 55

1 Introduction

Research institutions and funding agencies are increasingly adopting open-data science, where data is freely available without restrictions such as copyright [1, 2], or available under some data policy [3]. Institutions that own data can host it on a remote data repository for other researchers to find and use. If done right, this approach enables highly collaborative work; however, problems can emerge if institutions are not careful. Specifically, if institutions upload data with varying formats and access methods, it can be challenging to use and combine with other data. Seeing these issues and following the popularity of open data, GO FAIR[1] proposed the FAIR guidelines [4]. The guidelines intend to make it easier for researchers and machines to use public data. The guidelines are:

- **F**indable: Data should be easy for humans and computers to find.
- **A**ccesible: Data accessibility should be transparent. It should be easy for a human or machine to know *how* to access data.
- **I**nteroperable: In order to work well with other systems, the data should be interoperable and represented using a well-known language/format.
- **R**eusable: Data and metadata should be well-described to make them replicable and usable in other contexts.

1. The GO-FAIR website: `https://www.go-fair.org`

Since one of the core principles of the scientific method is to build upon knowledge, it is essential for researchers that data is reusable and replicable. Services like The Dataverse Project [5] and the Open Data Science Foundation [6] allow institutions to upload data and research projects that follow the FAIR principles. Dataverse, for instance, currently has 78— as of May 2022— data repositories worldwide, hosting over 135 thousand datasets[2].

For researchers to use datasets and build on previous work, the data importantly need to be *immutable*. Immutable data follow the *reusable* principle in the FAIR guidelines. Researchers consequently download the immutable data to use in their research projects. This system of publishing and downloading data works well, but only because the data is immutable and persistent. However, one limitation is that the dataset's meta properties continuously change— like last access, access rights, usage counts, ownership, etc. For instance, Sharma et al. [7] show that subjects in participatory research often change their perception of privacy, which requires a frequent update to data access policies. Other studies show that— depending on the workload— metadata operation could make up over 50% of the operations [8]. Metadata changes can arbitrarily occur by local modifications or remotely by the data owner or another file user. These changes need to propagate to the users it affects.

In this thesis, we are interested in the class of metadata related to data management. Data management metadata is used to represent data management *policies*, which garner the usage of the file, and they follow data as it is propagated and processed in data-processing pipelines. Several different policies can be specified using data management metadata, such as retention policies, usage policies, processing policies, etc. For instance, a data retention policy can specify that data usage is not allowed after five days since retrieval.

There is a small, finite amount of *standard* metadata, like `Access`, `Create`, `Modify` in the Linux operating system. However, an infinite amount of data management metadata exists since endless policies exist for files. Consider only the policy that a file is only available for N days. We can make endless versions of this policy by substituting N with ever-increasing numbers. Because of this, data management metadata needs to be handled differently than standard metadata. While all standard metadata can be represented individually, such as defining an object in memory for each metadata, the same can not be said for management metadata. A system for creating and defining custom metadata is necessary to enable data management policies. Additionally, these metadata fields must be tracked and transferred as data propagate. Suppose a user creates a new file with the data from another file. Then, the new file needs to cohere to the origin file's policy since accessing the new file indirectly

2. The Dataverse Project website: `https://dataverse.org/metrics`

accesses the original. If the policy does not transfer, then writing into a file without a policy is a way to escape the original file's policy.

Metadata tracking and transferring can be accomplished with tainting methods [9], often used in information flow control systems [10, 11]. However, having a mechanism for tracking information flow is also helpful for management policies. For instance, a data processing policy can specify that local metadata changes must propagate back to the original source file.

Although many data management policy decisions can be made on local data alone, some require global coordination. One such example is a restriction on data access based on a count of data usage. For instance, a medical project may restrict data access to not more than two parties at once. Another example is differential privacy, where a global coordinator maintains and enforces a global privacy budget.

We conjecture that metadata update propagation in open-data science fits well with the Publisher-Subscriber (pub-sub) model [12]. In this model, *publishers categorize* their messages into *topics*, instead of sending them directly to the receivers. To receive messages, *subscribers* can subscribe to topics concerning them. We adopt a model where the central data owner creates and defines metadata fields as topics. The owner can make important topics, such as access control, to be mandatory, and others optionally subscribable. Example topics that could be optionally subscribable are last-access, creation time, usage count, etc. Remote or local changes to metadata propagate to the affected file observers.

1.1 Thesis Statement

The advantage of applying the pub-sub model is that the model is well-established and heavily researched. Additionally, modeling metadata as categorizable messages is natural since metadata is usually composed of a unique key and a value.

Our thesis is

> *that scalable decentralized data management for research data can be constructed using a filesystem integrated pub-sub substrate for metadata dissemination.*

We support our thesis by implementing the Gurret prototype using a FUSE filesystem that intercepts system calls to orchestrate metadata. We show that

Gurret can support data management policies with custom metadata. The end system is a substrate that users can use on top of their systems and/or as a base system to build upon. It is therefore important that Gurret is transparent for end-users. Working with files without the substrate should not be noticeably faster than *with* the substrate. Additionally, we implement a *reference monitor* that tracks information flow and metadata propagation.

1.2 Scope and Limitations

This thesis specifies the design, requirements, and evaluation of Gurret. We develop the end-user client responsible for handling local metadata updates based on the requirements and evaluate its different aspects. We propose a scalable design for a backend service based on previously good designs; we do not implement the backend in this thesis. The client we develop supports connecting to a backend; however, we use mockups to simulate and verify the correctness on the client-side.

Many system calls can lead to a flow of information in several ways. We limit this thesis to only look at the system calls `open`, `write`, `rename`, `unlink`.

1.3 Context

This thesis is written in the context of the Cyber Security Group (CSG) work at UIT The Arctic University of Norway. Recently, the group has done extensive research related to Intel SGX, privacy in sports science, and participatory research projects. On the latter, the Lohpi infrastructure [3, 13] has been proposed, which serves as one of the inspirations of this thesis.

Lohpi is a data processing infrastructure for participatory research projects designed to distribute and control datasets with dataset policies in mind. Users interact with Lohpi and download datasets using the CLI `lh`.

1.4 Methodology

In the *Final Report of the ACM Task Force on the Core of Computer Science*, the task force divides the discipline of computing into three paradigms: *Theory*, *Abstraction*, and *Design* [14]. All three paradigms present an infrastructure for iterative progression on a subject. Individuals following one of the paradigms are expected to adjust and iterate the steps when they discover a flaw.

Theory the paradigm theory is related to the study of mathematics and is concerned with developing a sound theory. The steps involved are: (1) find an object to study; (2) create a hypothesis; (3) test the hypothesis rigorously; (4) interpret the results.

Abstraction the paradigm abstraction is an experimental paradigm for investigating a phenomenon. The infrastructure presents four steps for validating a phenomenon: (1) form a hypothesis; (2) construct a model for predicting the phenomenon; (3) create experiments and collect data; (4) analyze the results.

Design the final paradigm: *design* is an engineering paradigm design in the construction of a system. The four involved steps are: (1) state the requirements; (2) state the specifications; (3) design and implement the system; (4) test the system.

Again, the paradigms are designed to discover and solve flaws by going back to a previous step and adjusting. This thesis is rooted in the last paradigm: *design*. Throughout this thesis we

1. Define the properties (requirements) of the system.
2. Outline the specific details on how we implement said properties.
3. Define and conduct experiments.
4. Reflect over the result.

1.5 Outline

The rest of this thesis is structured as follows:

Chapter 2 details the technical background relevant to the thesis. The topics we elaborate on are: filesystems, FUSE, access control, and information flow control.

Chapter 3 describes the architecture of the Gurret system and gives implementation details for the Gurret client.

Chapter 4, 5, and 7 further elaborates on the design and implementation details of the Gurret client. Namely, (1) how Gurret implements metadata, (2) how end-users can subscribe to said metadata, (3) and how we use taint-tracking in Gurret to enable data policy propagation to prevent| policy violations occurring from an information flow.

Chapter 6 describes our design for the data-sharing and notification engine.

Chapter 8 benchmarks and evaluates the Gurret client. We benchmark different filesystems and core components of Gurret.

Chapter 9 contains related work, future work, and finally, a summary of our important findings in the thesis.

2 Background

This chapter covers the necessary background before introducing the Gurret system. We describe filesystems and how they store metadata, FUSE, access control, and information flow control. We begin by describing filesystems and focusing on how they store files and metadata.

2.1 Filesystem

Operating systems use filesystems to organize their data. Most operating systems have a filesystem, like NTFS for Windows [15], APFS for Mac [16]. A typical environment where filesystems are not present is embedded systems and microcontrollers, where memory is limited. On Linux, multiple filesystems are available, like ext4 and XFS, to name a couple.

Since this thesis focuses on filesystem technologies on Linux, we focus on the filesystem available on that platform. We will describe their main features and how they store data and metadata. The filesystems we consider are XFS [17], Btrfs [18], F2FS [19], and ext4 [20]. We choose these filesystems because they are among the most popular Linux filesystems [1] and mature, well-established filesystems [21]

1. According to `https://www.maketecheasier.com/choosing-the-best-linux-filesystem/` and `https://linuxhint.com/best_linux_file_systems_5/`

We begin with the XFS filesystem.

2.2 XFS

Developed by Silicon Graphics, Inc, XFS' purpose was to replace its previous filesystem, EFS [22]. XFS is designed to be a general-purpose filesystem with high scalability and the ability to store large files. Some of the main features of XFS are *allocation groups*, its extensive use of B+ trees, and its file storage capacity.

Allocation Groups

Allocation groups is one of the concepts introduced in XFS to address scalability. Allocation groups are ranges of addresses— typically between 0.5 to 4 gigabytes in size— where each group has control of the address range. One can think of allocation groups as sub-filesystems. Each allocation group has data structures for managing its address range. Allocation groups make XFS more scalable by enabling multiple processes to access different allocation groups in parallel. Silicon Graphics' previous filesystem, EFS, used a single thread for allocating and freeing blocks. This approach works fine for small workloads but scales poorly. As the number of processes running I/O operations increases, the single thread increasingly becomes a bottleneck.

B+ Trees

XFS makes good use of B+ trees to increase performance and scalability. XFS uses B+ trees to keep track of free blocks and directory indexes. The traditional data structure to store this information are bitmaps and lookup tables. The issue with these data structures is that their time complexity, $O(n)$, scales poorly. B+ trees, on the other hand, have a time complexity of $O(\log n)$.

Storage Capabilities

To support large files, XFS uses a 64-bit address space. This enables XFS to address almost eight exbibytes of data ($2^{63} - 1$ bytes).

File and Metadata Storage

For many filesystems and most Linux-based filesystems, the *inode* is the data structure that describes files and directories. The inode contains information about the file/directory, such as access mode, ownership, type, and more [2]. Every available filesystem on Linux uses this inode structure internally; however, filesystems usually create their own inode structure that contains the basic inode. XFS's inode `xfs_inode` contains XFS implementation-specific fields, such as its size.

To store *many* files, XFS *dynamically* allocates inodes. The alternative is to pre-allocate inodes when the filesystem is created. The disadvantage to pre-allocation is that inodes can run out. The data itself is stored using *extent trees*. This method utilizes trees to keep track of data. The method is very efficient when data is spread across multiple blocks. Section 2.5.2 further elaborates on the technique.

XFS stores file metadata in the inode. For additional metadata, XFS implements extended attributes to allow arbitrary key-value pairs to be associated with files. XFS can store extended attributes inline in the inode for small (≈100 bytes) attributes. Larger attributes are stored in a different data block. Increasing the inode size when creating the filesystem enables larger inline attributes. XFS has no limit on the *number* of extended attributes— unlike the ext family of filesystems— however, XFS *does* have a limit on attribute *size*.

2.3 Btrfs

Btrfs is an open-source, copy-on-write (COW) filesystem that started development in 2007. The goal of Btrfs is to work as a general filesystem capable of being used efficiently for many different workloads and devices such as mobile phones and servers.

Btrfs's architectural design is a *forest of trees*. Each disk volume on the filesystem is a B-tree, and a centralized B-tree root node (the superblock) contains all of the sub-volumes as its children. Btrfs makes good use of B-trees' $O(\log n)$ operations to increase performance and scalability. One of the main features in Btrfs is *checkpoints*.

2. `https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L614`

Checkpoints

Modifying files causes changes in the tree structure where the files reside. Instead of writing the changes directly to the said tree, they are first stored and captured as a new tree in the forest of trees. After a timeout (default is 30 seconds), or if enough pages are modified, the new tree is written to the disk, creating a *checkpoint*. The root node containing the forest of trees then switches the pointer of the modified root to point to the new root. Btrfs saves the old tree in case of a crash and data gets corrupted. In that case, the filesystem can roll back to the previous three.

File and Metadata Storage

Btrfs represent their files and directories using *inodes*. Btrfs's inode contains—among others— information about which subvolume a file is contained in and where the data is stored. Btrfs stores the data similar to XFS by using a tree structure. Unlike XFS, Btrfs does not support inline extended attributes.

2.4 F2FS

F2FS is a filesystem developed by Samsung Electronics that specifically targets flash storage. The goal of F2FS is to be a fast and reliable filesystem that can outperform traditional filesystems, such as ext4 and XFS, under the assumption that the host machines stores data on flash storage.

One of the ways F2FS makes use of flash storage is with a flash-friendly disk layout. F2FS divides the disk into three logical units: *segments*, *sections*, and *zones*. These units are configured to align well with the flash storage structure.

File and Metadata Storage

F2FS uses *node blocks* to represent inodes, direct nodes, and indirect nodes. Each node block is assigned a unique identifier that references them. The address of the node block can be located by indexing the *Node Address Table* (NAT) using the node identifier.

F2FS stores file content using *direct pointers*, a technique we further elaborate on in Secton 2.5; however, F2FS additionally support inline data and inline extended attribute, storing them in the inode structure. By default, F2FS is capable of inlining up to 3692 bytes of data and 200 bytes of extended attributes

one block	many blocks	many blocks	one blocks	one block	many blocks	many blocks
The Superblock	Group Descriptors	Reserved GDT Blocks	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks

Table 2.1: Disk layout of ext4.

directly in the inode. The extended attribute size is configurable with the `inline_xattr_size` mount option [3].

2.5 ext4

Since we use ext4 as our filesystem, we will spend more time on it than the other filesystems to better understand it. ext4 is the fourth version of the ext family of filesystems. The main improvement from the previous filesystem ext3 was more storage capability and better performance.

2.5.1 ext4 Structure

The filesystem is divided into a series of blocks. The size of each block is a power of two, ranging from 1 KiB to 64 KiB, but typically the same size as a page. On x86, this is 4KiB. Table 2.1 shows the basic disk layout for the filesystem. In the table, each square represents a named segment in the layout, with the size of the segment above. The following sections explain each segment in detail.

The Superblock

The superblock contains metadata of the mounted filesystem, such as how many blocks and inodes are on the filesystem, the filesystem's status, and more. An overview of the metadata contained in the superblock can be seen by running the following command in a Linux terminal:

```
$ sudo dumpe2fs -h {Your mounted disk}
```

We run the command on our disk: `/dev/nvme0n1p2`. Figure 2.1 show the output.

3. https://www.kernel.org/doc/html/latest/filesystems/f2fs.html

```
Last mounted on:          /
Filesystem magic number:  0xEF53
Filesystem features:      has_journal ext_attr resize_inode dir_index
        filetype needs_recovery extent 64bit flex_bg sparse_super
        large_file huge_file dir_nlink extra_isize metadata_csum
Filesystem state:         clean
Filesystem OS type:       Linux
Inode count:              31162368
Block count:              124645632
First block:              0
Block size:               4096
First inode:              11
Inode size:               256
Filesystem created:       Mon Aug 23 13:17:04 2021
```

Figure 2.1: Output from dumpe2fs. A lot of lines are omitted for clarity.

The output has 58 lines of information; we omit many lines for clarity.

Group Descriptors and Reserved GDT Blocks

The group descriptor blocks contain information about the other blocks. It contains the block numbers of the data block bitmap, inode bitmap, inode table, and data blocks. The Reserved GDT blocks are used to add more group descriptors to the table.

Data Block and inode Bitmap

The data block bitmap and inode bitmap holds the availability status of every block and inode. By availability status, we mean if the block is in use or free. The bitmaps are essentially two large arrays where index *I* specify whether block/inode number *I* is occupied or not. They are bitmaps to preserve space.

inode Table

ext4 uses inode to represent files and folders on the filesystem. The content of the file is stored in data blocks. The inode has 15 pointers to blocks to access the file's content. A *pointer* in this context is just the unique block number of a block. We can get the address on disk from a block pointer with :

$$\texttt{BLOCK\_ADDR} = \texttt{BLOCK\_BASE\_ADDR} + \texttt{BLOCK\_POINTER} \times \texttt{BLOCK\_SIZE}$$

The 15 pointers contained in the inode are:

- 12 pointers to blocks where data resides.
- 1 singly indirect pointer, pointing to a block containing block pointers
- 1 doubly indirect pointer, pointing to a block containing singly indirect pointers.
- 1 triply indirect pointer, pointing to a block containing doubly indirect pointers

Figure 2.2 shows an overview of the 15 pointers contained in the inode. Each square represents a block, and the arrows indicate what is being pointed to. The letter D is a shorthand for Data. Some arrows and the triply indirect pointer are omitted for clarity.

This structure is very inefficient for storing large files. A file with content spanning 10,000 blocks would require an indirect mapping to each of the 10,000 blocks. Reading the whole file from beginning to end using this structure is also expensive.

Data Blocks

Data blocks contain the content of files and folders. This block group is by far the largest in the system. Files and directories reference data blocks by using the block pointers.

2.5.2 Extent Trees

To improve the inode's block pointer structure, ext4 introduced *extent trees*. Extent trees are trees where the leaf nodes in the tree specify where data begins and how many blocks it spans. Using extent trees, storing big files within an inode can be reduced to a single extent node specifying where it starts and how many blocks it spans. If the file is fragmented and split into N chunks, the extent tree would have N leaf nodes specifying the blocks' location.

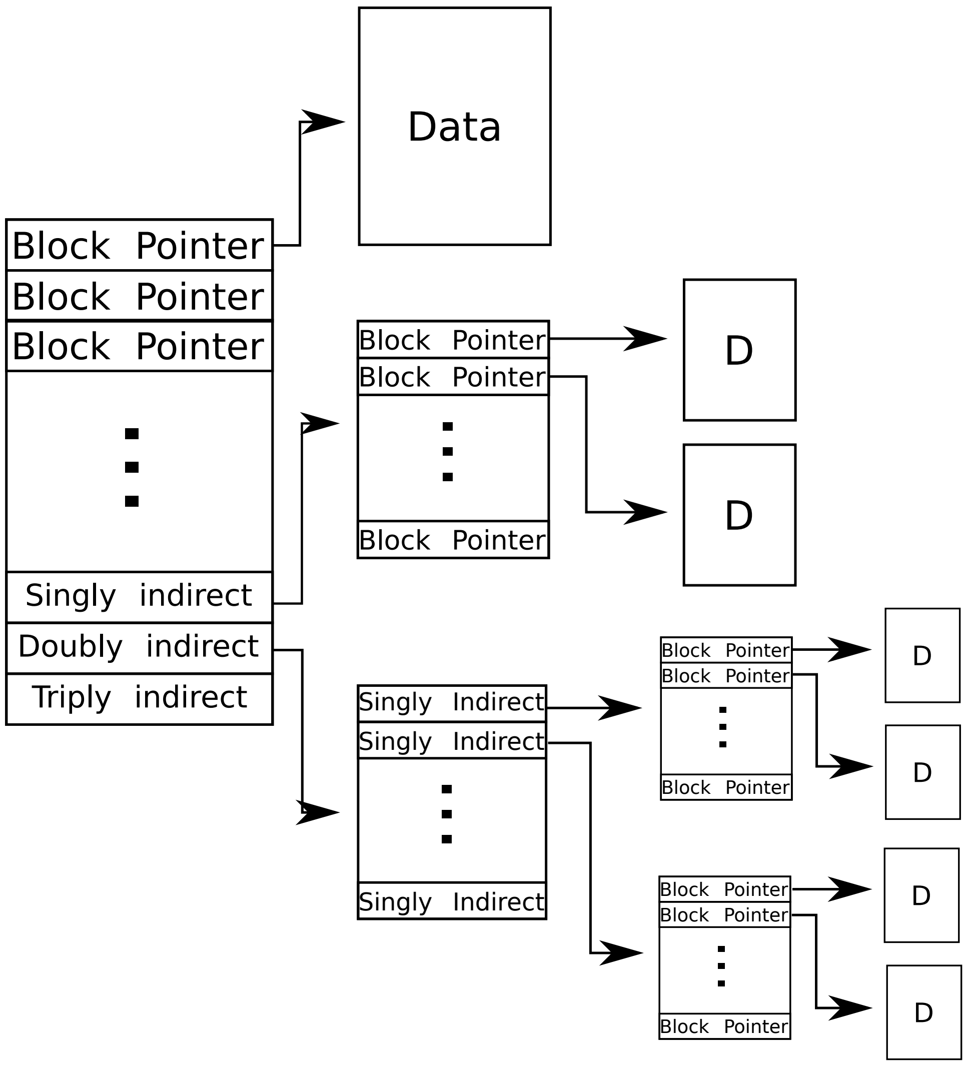

Figure 2.2: Overview of the inode block pointers. Tripley indirect pointer is omitted.

2.5.3 Inline Data

Both file data and extended attributes can be stored in the ext4 inode for small data sizes. The default inode size is 256 bytes. Increasing the inode size can accommodate more space for inline data; however, increasing the size is only possible when creating the filesystem.

2.5.4 Directories

ext4 represents directories as files. To distinguish between regular files and directories, ext4 uses the `type` metadata field to mark inodes as either files or directories. The content of Directories are blocks of *directory entries*. Directory entries contain information about the file, such as its inode number and filename. To traverse into a sub-directory, the OS performs a linear scan within the directory to find the inode number of the sub-directory. The linear scan works fine for small to medium-sized directories. However, the linear scan scales poorly because of the $O(n)$ time complexity. ext4 allows directories to use a B-tree hashmap instead of the linear collection of blocks to increase scalability. The Hash Tree uses more space in memory, but scales better with its $O(\log n)$ operations.

2.5.5 Journaling

To improve performance in the case of a crash or corruption, ext4 uses a technique called *Journaling*. Instead of writing disk changes directly to disk, they are first written to a special Journal file that keeps track of all changes. If a crash occurs, the filesystem can recover by looking in the Journal and redoing the tasks that left the filesystem in an inconsistent state. The alternative to using Journaling is to do a filesystem check `fsck` [23]. The filesystem check traverses the file system and checks for inconsistencies and corruption. The problem with the filesystem check is its speed. It is slow since it has to traverse a lot of memory and check for inconsistencies. It might not take too long for small filesystems; however, it can take hours for larger filesystems [23] or even days according to some accounts [4].

4. `https://serverfault.com/questions/966244/how-long-can-fsck-take-on-a-30-tb-volume`

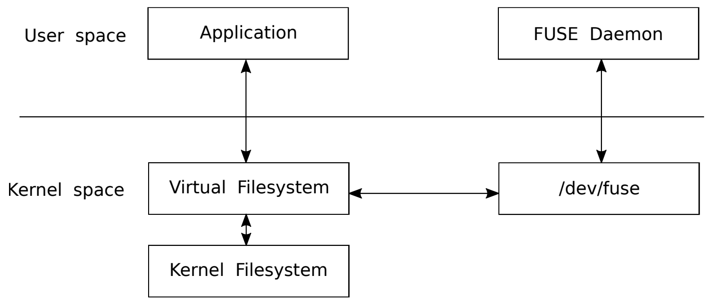

Figure 2.3: The FUSE architecture.

2.6 FUSE

FUSE is a framework for filesystems that run in user space rather than kernel space. Working with kernel-level filesystems is hard. Filesystems usually have extensive APIs and are difficult to debug since they live at the kernel level. A mistake in a filesystem can result in a reboot or corrupted data.

FUSE simplifies creating a filesystem by having a simpler API and running it in user space. Having the filesystem run in user space makes it easier to debug since user space crashes usually do not require a system reboot.

FUSE works by intercepting system calls— e.g., `READ`, `WRITE`— and handling them according to a virtual filesystem specification. The virtual filesystem is mounted with FUSE on top of a folder. This allows the virtual filesystem to intercept system calls within the folder. Probably the simplest system one can build with FUSE is a *passthrough*. A passthrough is a filesystem that forwards system calls to the OS. A more practical and well-known example is *sshfs*, which allows users to mount remote directories on their machine using ssh [24]. It achieves this by intercepting various system calls, running them remotely, and sending the results back.

The basic FUSE architecture consists of a kernel-level module and a user-level daemon process. The two components communicate using the device file `/dev/fuse`.

When a user-level application triggers a system call within a FUSE-mounted directory, they are translated into a FUSE request and put into the device file by the virtual filesystem. Afterward, the user application will go to sleep, and the FUSE-daemon will read the request from the device file. The daemon then handles the request, writes the result back to the device file, and wakes the application up again to continue execution [25]. Figure 2.3 show the architecture of FUSE.

One drawback with FUSE is the overhead of intercepting system calls. In a simple passthrough implementation, system calls need to be communicated and translated through the `/dev/fuse` file, in addition to the actual file operation. We measure the overhead associated with FUSE in Section 8.2.

2.7 Access Control

Access Control is a security technique concerned with allowing and limiting access to resources based on an *access policy*. There are two main access control model categories: discretionary access control (DAC) and mandatory access control (MAC) [26]. The main difference between the two categories is who decides the access control policy. In DAC, the owner of a resource decides the access policy. In MAC, however, additional rules and frameworks play a role in the policy. Filesystems and social media are typical places where DAC is used. On the other hand, MAC is typically used in large corporations and institutions.

2.8 Discretionary Access Control

In DAC, the owner/creator of a resource decides the access policy. Examples of systems under DAC are Access-Control List (ACL) and Capability-Based Access Control (CBAC)

2.8.1 Access-Control List

In an ACL system, the resource owner has a list that determines the permissions for other users to the resource. Many filesystems implement ACL; examples include ZFS, ext3, and ext4. ext4 sets ACL with the `setfacl` commands. Example usage:

```
$ setfacl -m "u:alice:rw" messages
```

This command gives the user alice read and write permissions to the messages file. The command `getfacl` is used to retrieve the ACL for a file:

```
$ getfacl messages
# file: messages
# owner: {owner}
# group: {group}
```

```
user::rw-
user:alice:rw-
group::r--
mask::rw-
other::r--
```

2.8.2 Capability-Based Access Control

In CBAC, access to a resource is evaluated using a *capability*. A capability is a unique tuple containing a reference to a resource and a set of permissions to the resource. For instance: ⟨`/user/messages`, {`READ`, `WRITE`}⟩. To use a resource, the capability is validated by an *authority* or *ambient authority* [27]. An important point in CBAC is the ability to share capabilities. For instance, the example tuple above should be possible to share among processes in a CBAC system.

One of the use cases for CBAC is the *confused deputy problem* (CDP) [28]. The confused deputy problem is when an elevated/privileged program is exploited to do something malicious. In the classical CDP example [28], a Fortran compiler is given elevated privileges to write debug information to a file in the root folder. The exploit occurs when a user specifies that the output file should have the same name as a privileged file in the root folder. For instance:

```
$ fortcc file.f90 -o /etc/passwd
```

A solution to the problem would be to introduce capability-based access control. Using CBAC, the compiler only has the capability to the debug file, e.g., ⟨`/root/debug`, {`READ`, `WRITE`}⟩. Trying to do the same exploit will not work since the compiler *only* has access permission to the debug file.

2.9 Mandatory Access Control

In contrast to discretionary access control, in MAC, an *institution* creates a structure/hierarchy of labels assigned to resources. Users are given a *clearance level* that determines what they can access. For instance, a MAC system could define the labels: `top-secret`, `secret`, and `unclassified`. Then, for a user to access a *secret* resource, they would need a clearance of secret as well. The hierarchy of labels and clearances is usually implemented using a *lattice*. We further explore two popular MAC models: The Bell-LaPadula and the Biba models.

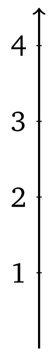

Figure 2.4: Linear lattice with the values 1 - 4.

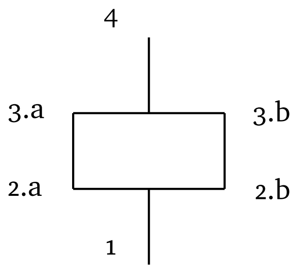

Figure 2.5: A more complicated lattice.

2.9.1 Lattice

A lattice is a partially ordered set in which every pair of elements has a greatest lower bound and a least upper bound [29]. *Bounded* lattices have the additional requirements that the set is also finite. Although we continue to only use the word *lattice* what we mean is a *bounded* lattice. More formally:

$$\begin{aligned}
&L \text{ is a lattice} \Leftrightarrow \\
&(1)\ |L| \in \omega \\
&(2)\ \forall x, y \in L\ \exists z : x \geq z \ \&\ y \geq z \\
&(3)\ \forall x, y \in L\ \exists z : x \leq z \ \&\ y \leq z
\end{aligned}$$

An example of a lattice structure are all positive numbers from 0 to 100 e.g $\{x : x \geq 0 \ \&\ x \leq 100\}$. With lattices, simple restriction levels such as the linear restriction level seen in Figure 2.4, as well as more complicated ones such as Figure 2.5, can be described using lattices.

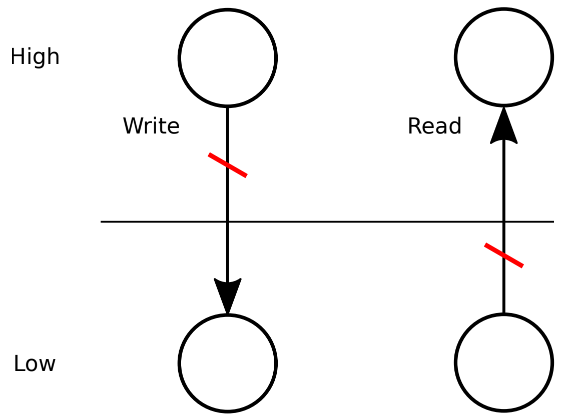

Figure 2.6: no-read-up, no-write-down rules.

2.9.2 The Bell-LaPadula Model

The Bell-LaPadula model [30] is a MAC model that is concerned with data confidentiality. The purpose of the model is to avoid data leaking from a higher security label, say `top-secret`, to a lower level, `secret`, for instance. To prevent this, the model defines the rules: *no read up* and *no write down*.

The purpose of the first rule: *no read up* is to avoid users with a lower clearance reading a more classified resource. For instance, a user with *unclassifide* clearance should not be able to read *top-secret* resources. The second rule: *no write down* is to disallow resources leaking to a lower level. For instance, the content of a *top-secret* file should not be able to be written to a *secret* file. Figure 2.6 displays the rules in the model.

2.9.3 The Biba Model

In contrast to Bell-LaPadula, the Biba model [31] is concerned with data *integrity*. The Biba model has similar rules as Bell-LaPadula. The rules in the model are: *no read down* and *no write up*. To understand why these rules preserve data integrity, consider a lattice hierarchy with the labels `unclassified` and `secret`. If users are allowed to use content from an *unclassified* document and write it to a *secret* document, then the document would be *less secret*. Figure 2.7 shows the rules in the model.

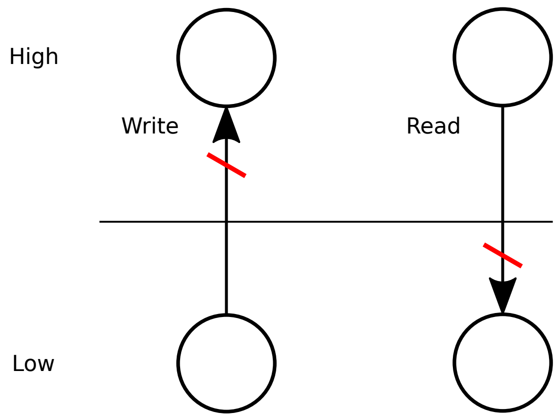

Figure 2.7: no-read-down, no-write-up rules.

2.9.4 Access-Control Limitations

One limitation of access control is that access control can only be used to check whether access is valid or not. In order to monitor information *flow* and data provenance, other mechanisms are needed [32].

2.10 Information Flow Control

Information Flow Control (IFC) is a security measurement concerned with access control and—more importantly— data provenance [33]. Data provenance is how data *flows* in a system between classes. Class, in this context, is some actor trying to access a resource, e.g., user, file, process. IFC systems are useful when data have a policy associated with them. The data policies define how classes are allowed to use the data.

2.10.1 Information Flow

A *flow* in an IFC system is the transfer of data from one class to another. We define the symbol "$\longrightarrow$" to mean the relation of information flow between two classes. Consequently, $A \longrightarrow B$ indicates a flow from class A to class B. Flows come in two flavors: explicit and implicit.

A flow $X \longrightarrow Y$ is *explicit* if the data of X is directly used to assign Y. For example:

```
X = True
```

```
Y = !X
```

A flow $X \rightarrow Y$ is *implicit* if Y is related to the data of X through program control flow (e.g., `if`, `else`, `while`, etc). Consider the example:

```
X = ... # True or False
Y = True

if X = True
    then Y = False
```

Even though X never assigns Y, we can still infer both variables knowing only one.

2.10.2 Flow Control

Each flow in an IFC needs to be verified by a *reference monitor* to ensure classes follow data policy. A reference monitor is an external component that monitors and verifies flow between classes.

There are two main ways to verify flows: *statically* and *dynamically*.

Statically

Denning and Denning [34] showed that it is possible to do static analysis and create a language that, at compile-time, checks for flow violations. The advantage of this method is that the enforcement does not affect the program's runtime since programs that successfully compile do not need runtime checks; however, dynamic runtime checks can still be necessary to guard against hardware malfunction. Multiple programming languages do static enforcement, one of which is Jif [35].

One limitation with static enforcement is that everything needs to be known at compile time. In a dynamic system where class labels can change arbitrarily, static enforcement might not be sufficient.

Dynamically

The reference monitor inspects data flow at runtime to detect flow violations in dynamic enforcement. Dynamic enforcement is more flexible than static enforcement; however, it comes with a runtime overhead. Another advantage of dynamic enforcement is that special programming languages—like Jif— are unnecessary.

2.11 Summary

This chapter has explored different topics necessary to understand our system, Gurret. We have explored topics within different filesystem alternatives on Linux, ext4 in particular; FUSE; Access Control; and Information Flow Control.

3 The Gurret System

This chapter describes the Gurret system: a distributed system that uses the pub-sub model at the filesystem level to distribute management metadata. First, we give a general overview of the architecture of Gurret, before describing specific details about the implementation of the Gurret *client*. Further expositions on specific details of Gurret can be found in later chapters.

3.1 Architecture

Gurret is a distributed system consisting of two main components: (1) a distributed data-sharing and notification engine; (2) the Gurret client, hereafter referred to just as Gurret, running on end-user machines. Figure 3.1 shows an overview of the system.

The data-sharing and notification engine— or *the backend*— runs in the cloud and is responsible for distributing data and metadata. The backend stores data that follows the FAIR principles, i.e., they are findable, accessible, interoperable, and reusable. End-users send requests to the backed when they want access shared data. The other part of the backend is the *notification engine*, which collects and distributes metadata to the affected users.

Gurret is a user client composed of a filesystem container and a daemon, both running on end-user machines. The filesystem container is a mounted

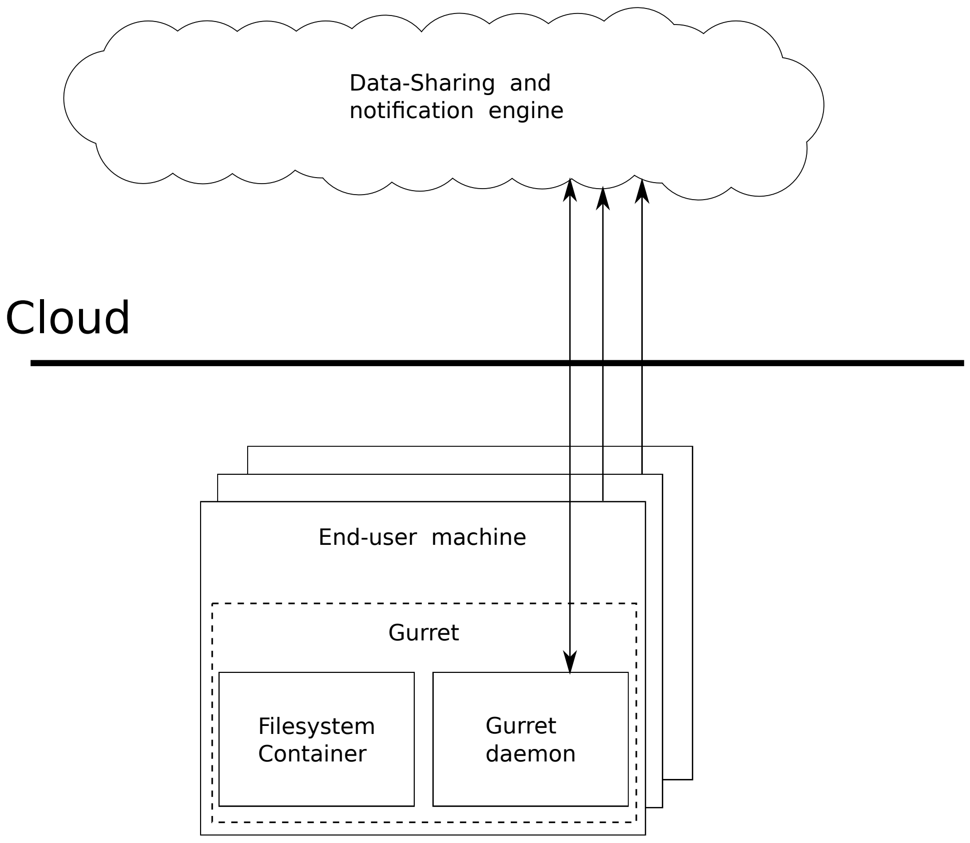

Figure 3.1: Overview of the Gurret system.

file/loop device containing a filesystem. The Gurret daemon is a daemon that intercepts system calls, orchestrates metadata, and tracks information flow within the filesystem container. The backend and the Gurret client communicate to exchange metadata updates.

The rest of this chapter focuses on the architecture of the Gurret client and the implementation decisions. We elaborate on the backend in Chapter 6.

3.2 Filesystem Container

Gurret uses a mounted file/loop device containing a filesystem to store downloaded files from the backend. The file initially has 1 GB of storage capacity and grows if a file demands more space than available. Gurret stores files on a loop device for three main reasons: (1) we have more control over when the data is available. The content inside the file is not available as long as the file is not mounted; (2) we have a well-defined, isolated area for placing files; (3)

```
$ dd if=/dev/zero of={your-file} bs=1G count=1
$ sudo mkfs.ext4 {your-file}
$ sudo mount -o loop {your-file} {destination-path}
```

Figure 3.2: Creating and mounting a loop device containg ext4.

```
-- files
-- .custom-metadata
{side-car-files}
```

Figure 3.3: The pre-arranged filesystem structure.

the file is easy to share.

The filesystem on the loop device is ext4. The primary reasons for using ext4 are its mature and well-established support for inline extended attributes [21]. Storing metadata inline in the inode requires fewer reads from the operating system, increasing performance. There are many alternative filesystems to use, such as XFS, that also support storing inline extended attributes; however, the benchmarks in Chapter 8 show that the performance of different filesystems is, in general, very similar. Figure 3.2 illustrates the process involved in creating and mounting a loop device with ext4.

3.3 Filesystem Structure

The filesystem comes with a simple pre-arranged file structure. The structure contains two folders and some sidecar files, as shown in Figure 3.3, where we use -- to denote directories. This section describes this filesystem structure.

The *files* folder contains the files downloaded from the backend. Users are free to move files around and create folders; however, the *files* folder is the default download location.

The *.custom-metadata* folder contains custom metadata. Custom metadata is arbitrary metadata fields defined by the data owner. Section 4.3 covers this topic in more depth.

Gurret stores many internal data structures in sidecar files in the format .{*name-of-structure*}. We mention these files as they appear.

```
{
    "Access": 1644396257,
    "Access-Label", "private",
    "Access-List", ["UiT", "VU"]
}
```

Figure 3.4: Example metadata for a file.

3.4 Storing Metadata

Gurret stores metadata for files using extended attributes. The metadata storage format is JSON. Gurret represents metadata as key-value pairs, where the key is the name of the metadata and the value is the value of the metadata. Figure 3.4 shows an example of the metadata format.

Some filesystems allow extended attributes to be stored inline in the inode. The available space depends on the size of the inode. In the ext4 filesystem we currently use, the default inode size is 256 bytes. However, the inode only uses 160 of the 256 bytes. This leaves $256 - 160 = 96$ bytes available for storing extended attributes. The filesystem store the extended attributes in other data blocks if the extended attributes use more space than available. The inode size can be increased to accommodate more space for extended attributes. For instance, an inode size of 4096 would leave 3936 bytes for extended attributes. However, the inode size can only be changed when creating the filesystem.

3.5 Gurret Daemon

The Gurret daemon is a daemon mounted on top of the filesystem container. Gurret intercepts system calls within the directory and communicates metadata updates with the backend. To track data provenance, the Gurret daemon intercepts the system calls: `open`, `write`, `rename`, and `unlink` inside the filesystem container. Chapter 7 goes into depth about how Gurret handles each system call. The Gurret daemon is implemented in the Rust programming language [1] and the FUSE interface [25].

1. `https://www.rust-lang.org/`

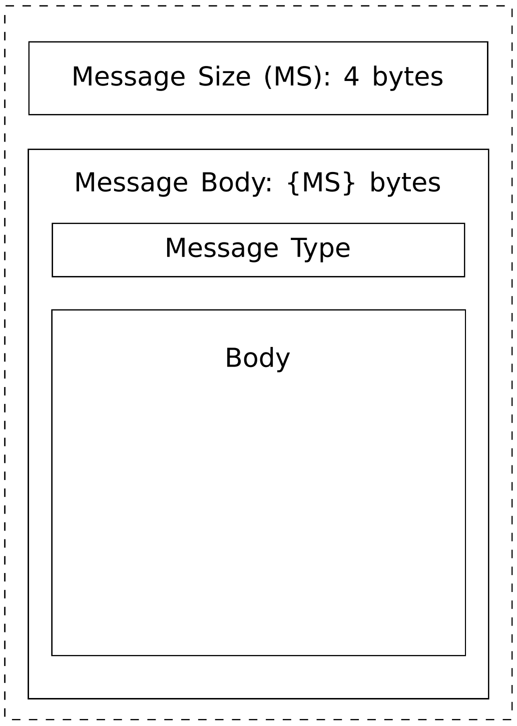

Figure 3.5: The structure of a message.

3.6 Message Structure

The backend and Gurret communicate by writing *messages* to a socket. The first four bytes in a message is the message size. The next N bytes in the socket, where N is the message size, is the body of the message. The body contains two fields: the type of message and the message itself. The message body uses JSON format. Figure 3.5 shows an overview of the message structure.

We represent the message type with an enum. The possible values are: `MetadataUpdate`, `GetSubscriptions`, and `SetSubscriptions`. Although the names are quite explanatory, Chapter 4 explains the purposes for each type.

3.7 Summary

This chapter focuses on an overview of the entire system and implementation details of Gurret. The following chapters describe the other parts of the system in more detail: how Gurret defines and handles metadata, how end-users subscribe to metadata, the backend design, and finally, taint tracking in Gurret.

/4 Metadata

Gurret uses metadata to represent data management policies. We support various policies by allowing users to define custom metadata that represents their policy. This chapter describes the implementation details of how Gurret represents metadata, how users can create data management policies, and how metadata is received and updated to and from the backend. Since we implement the Gurret client in Rust, the code figures are in Rust. First, we explain how Gurret represents metadata before explaining how Gurret and the data-sharing and notification engine (the backend) communicate.

4.1 Metadata Interface

On the file level, Gurret represents metadata with key-value pairs and stores it in the extended attributes of files. Internally, on the other hand, Gurret represents metadata by an interface (called 'traits' in Rust). The interface has three methods: *check*, *update*, and *access*.

```
trait Metadata
{
    type Item;
    fn check(&self, op: Operation) -> io::Result<bool>;
    fn update(&self) -> io::Result<()>;
    fn access(&self) -> io::Result<Self::Item>;
}
```

Check tests whether a system call (e.g., `open`, `write`) triggered a metadata change, *update* updates the metadata, and *access* accesses the metadata field. The access method returns a generic type *Item* to support multiple types of metadata. The most common, however, are integers and strings. The check method takes in an enumeration value representing all possible system calls.

```
enum Operation
{
    Open,
    Read,
    Write,
    // etc.
}
```

Gurret defines and implements the interface for two types of metadata: standard metadata and *custom* metadata. Standard metadata is metadata that every file in Unix comes with—Access, Modify, Birth. The `stat` command can be used on a file to view the standard metadata. Custom metadata is file specific and is— as the name implies— customizable by end-users. We explain how we implement the interface for the two types.

4.2 Standard Metadata

We implement the metadata interface for the `Access` metadata field for files. The access field stores the last time a user accessed a file. In Gurret, we define access as when an `open` system call is performed on a file. The *check* returns `true` only if the operation was `open`. The *update* method updates the extended attribute value with the latest system time, and the access method fetches the value. Figure 4.1 show the full implementation. The process for implementing the interface for the other standard metadata is the same.

4.3 Custom Metadata

Meta-code are small programs that Gurret dynamically execute during runtime [36, 37]. End-users use meta-code to define and specify the behavior of *custom metadata*. Custom metadata is highly extensible metadata that end-users can create and are associated with files. Many types of metadata require additional steps to check or update. For instance, suppose a file has a label that

```
struct Access(PathBuf); // name of the file associated with this metadata
impl Metadata for Access
{
    type Item=u32;
    fn check(&self, op: Operation) -> io::Result<bool>
    {
        Ok(op == Open)
    }
    fn update(&self) -> io::Result<()>
    {
        set_ext_attr_field(&self.0, "Access", SystemTime::now())
    }
    fn access(&self ) -> io::Result<Self::Item>
    {
        get_ext_attr_field(&self.0, "Access")
    }
}
```

Figure 4.1: Access metadata implementation.

only resides on a trusted third-party machine. The only way to access such a label would be to send a request to the machine. Custom metadata allows users to create metadata that conforms to these types of behavior. End-users can create custom metadata for a file f by first creating a folder inside the *.custom-metadata* folder with the name of f. All custom metadata fields for files are defined inside a folder with the same name as the file. Then, create a folder with the custom metadata's name inside the file's folder, e.g.,

```
-- .custom-metadata
        -- {file}
                -- {new-metadata}
```

This new folder should contain three other folders: *check*, *update*, and *access*, that correspond to the metadata interface. Three meta-codes: one that checks whether an update occurred, one that updates metadata, and one that accesses metadata, should be contained in the corresponding folder. Figure 4.2 shows the hierarchy of the folders.

4.3.1 Meta-Code Requirements

Gurret runs meta-code during runtime in a separate process. Gurret communicates with meta-code by using stdin and stdout. The meta-code that checks

```
.custom-metadata
    {file}
        {new-field}
            check
                {check-meta-code}
            update
                {update-meta-code}
            access
                {access-meta-code}
```

Figure 4.2: Custom Metadata hierarchy.

for updates receives the intercepted system call through stdin and responds by writing to the stdout. A check that resulted in a metadata change should communicate this back to Gurret by writing `"true"` to stdout, or `"false"` if no update occurred. The update meta-code receives optional arguments through stdin; however, it does not write any result to stdout. Gurret assumes that the update meta-code always runs successfully. The access meta-code receives no arguments and writes the result of the access to stdout. As previously mentioned, we store metadata in extended attributes; however, custom metadata has the *option* to store them differently, such as on a remote server.

The meta-code can look at the file structure to figure out which file and field it concerns. For instance, suppose an update meta-code is contained in the following filestructre

```
-- .custom-metadata
    -- sportsData
        -- Access-List
            -- update
                main.rs
```

The meta-code can deduct that it needs to change the `Access-List` field for the `sportsData` file.

We supply a metadata interface implementation that uses Rust files as meta-code. However, any program that conforms to the specified meta-code requirements and implements the metadata interface can be used to implement custom metadata. Therefore, an implementation using python, C/C++, Java, or even raw executables, could easily be added.

```
struct FiveDays;
impl Metadata for FiveDays
{
    type Item=bool;
    fn check(&self, _: Operation) -> io::Result<bool>
    {
        // reads filename from the current working directory
        let file = get_file();
        let create = get_create_time(file);
        let today = SystemTime::now();
        let seconds_since_create = today.duration_since(create)?.as_secs();
        Ok(seconds_since_create >= 5 * SECS_PER_DAY);
    }
    fn update(&self) -> io::Result<()>
    {
        set_ext_attr_field(get_file(), "five-days", true)
    }
    fn access(&self) -> io::Result<Self::Item>
    {
        get_ext_attr_field(get_file(), "five-days")
    }
}
```

Figure 4.3: Implementation of the FiveDays data policy

4.4 Creating Data Management Policies

Data owners can create data management policies with custom metadata. We show how this is possible by creating a policy `FiveDays` that only allows access to a file for five days. We represent the metadata with a boolean value representing whether five days have passed. The value is initially zero. We store the value in extended attributes, and the access method returns the stored value. The check method checks and responds with `"true"` if five days have passed since the file was created. Since Gurret only invokes the update meta-code after five days, it can simply change the value stored in extended attributes to true. Figure 4.3 shows the implementation. For this particular metadata, the operation does not matter. To enforce this policy, Gurret needs a reference monitor similar to the one in Chapter 7.

```
loop
{
    thread::sleep(T);
    let messages = read_socket();
    for message in messages
    {
        if message.type == MESSAGE_TYPE_METADATAUPDATE
        {
            let mut xattr = read_xattr(&message.body.name);
            for (field, value) in message.body.updates
            {
                xattr[field] = value;
            }
            write_xattr(&message.name, xattr);
        }
    }
}
```

Figure 4.4: Algorithm for receiving metadata updates.

4.5 Metadata Update

Gurret defines two components for metadata updates. One to receive and update metadata, and one to send local metadata changes to the remote.

4.5.1 Receiving Updates

Gurret communicates with the remote backend using a TCP socket. Gurret regularly checks the socket for updates after a configurable timeout T (default is 30 seconds). Metadata updates for files are received in the form of the `MetadataUpdate` message type, containing the file and the changed fields. To update a file's metadata, Gurret reads the old metadata stored in extended attributes and replaces the old values with the new for each changed field. Figure 4.4 shows the algorithm.

4.5.2 Sending Updates

We define a `MetadataHandler` interface to deal with local metadata changes. The interface has two methods: `changes` and `update_remote`. Figure 4.5 contains simplified code for the interface.

```
trait MetadataHandler
{
    fn changes(&self, file: Path, operation: Operation)
        -> Vec<Metadata>;
    fn update_remote(&self);
}
```

Figure 4.5: Algorithm for receiving metadata updates.

```
    // is called on the system call open
    fn open(file: Path)
    {
        // ...
        for changes in metadata_handler.changes(file, Operation::Open)
        {
            change.update();
        }
    }
```

Figure 4.6: Pseudo-code for detecting metadata updates.

The `changes` method checks whether a system call triggers metadata updates. The arguments for the methods are the path to the file being operated on and the operation itself. The operation type is an enum that enumerates all the possible systems calls, like `open` and `write`. The changes method calls the metadata check method in Section 4.1 for each metadata defined on the file and returns the ones that returned true. The resulting list (if any) is then iterated over, and the update method is called for each file. Additionally, any file whose metadata changes is stored in an internal data structure. Figure 4.6 shows pseudo-code for the described metadata checking procedure.

Gurret has a dedicated thread that batch uploads metadata updates to the remote system. After a timeout of 30 seconds, the `update_remote` method is invoked, which compiles every recorded metadata change into messages and writes them to the socket connection with the backend.

4.6 Summary

This chapter has explained how Gurret represents standard and custom metadata, the meta-code requirements, how users can create data management policies, and how Gurret and the backend communicate updates.

5 Metadata Subscriptions

Gurret effectively distributes metadata by using the pub-sub pattern, where end-users subscribe and configure the topics they want to receive updates on. We believe the pub-sub model is a good architecture and natural to use in systems like Gurret. There are two primary reasons for this.

First, users of Gurret likely follow a workflow that heavily utilizes metadata. However, users do not necessarily want *all* metadata, and not necessarily *always*. For instance, a file might have *hundreds* of metadata fields, and some workflows might only require updating one field; on a daily basis. These requirements can be expressed naturally with subscriptions to metadata fields [38].

Second, the publisher-subscriber pattern facilitates highly scalable and flexible systems [38]. These are essential traits for the data-sharing and notification engine system since it must be capable of serving potentially thousands of end-users simultaneously.

This chapter focuses on how end-users subscribe to metadata and how users can change the update frequency.

```
{ "name": <filename>, "options": [
     // subscription options
    {"name" <metadata-field>, "filters": <filters>},
    {"name" <metadata-field>, "filters": <filters>},
    // ...
]}
```

Figure 5.1: The Document structure of subscription messages.

5.1 Subscription Language

Gurret uses a simple JSON/document-based language— inspired by MongoDB's query language MQL— to subscribe to metadata. Documents have two main fields: (1) the name of the file the end-user wants to subscribe/unsubscribe to; (2) the metadata subscription options. The file's name is a unique identifier that the backend data-sharing system distributes. The *subscription options* is an array with the fields the users wish to subscribe to and the *filters* for the field. Filters are simple patterns that specify how often (if ever) users wish to receive updates for fields. Many scalable and expressive languages already exist to specify filters, like GEM [39]; however, we choose to use a very simple Document-based language.

the `filters` array contains individual filters for fields. The available filters are are: *status*, *interval*, and *count*. The status field specifies whether users wish to subscribe or unsubscribe to a field. The other fields: *interval* and *count* specify how often the end-user should be notified. For instance, *count* specifies how many times a metadata field need to change before notifying the user. We go into more depth on the optional fields *count* and *interval* in Section 5.2. We represent individual filters as JSON objects with the *name* of the filter (e.g., status, count) and the *value* of the option. Figure 5.1 show the complete document structure.

Gurret and the notification engine use this format to communicate subscriptions. However, some fields have a different meaning for Gurret clients when they are sent vs. when they are received. When receiving filter option from the backend, the *name* field is the name of the filter, and the *value* field specifies whether it is required or optional. For `status`, this is:

```
{ "name": "status", "value": <"optional" | "required"> }
```

We use the ⟨{option1}|{option2}⟩ construct to mean that the value is *either* {option1} *or* {option2}

When sending filters, the *name* is the name of the filter, however, the *value* is the user's choice for that filter. For the *status* filter, the available choices are *subscribe* and *unsubscribe*.

```
{ "name": "status", "value": <"subscribe" | "unsubscribe"> }
```

5.2 Metadata-Field Querying

Some metadata-field filters specify how often end-users should receive metadata updates. The justification for this is that some users may wish to limit the number of updates. For instance, suppose a file receives hundreds of updates each minute; this amount can be too big for users with low bandwidth.

Currently, the only available update limiting filters are: *count* and *interval*. Both limit how often end-users receive updates on the fields. The *count* filter specifies how many times a metadata field has to change before sending an update, and the *interval* filter specifies how often (timewise) an update should occur. The syntax for the two filters are:

```
{ "name": <"count" | "interval">, "value": <query-expr> }
```

The query expression takes inspiration from MQL and uses JSON objects/documents to express the query. The syntax for the `<query-expr>` is

```
{ <comparison-operator>: <value> }
```

For instance, the query to receive updates after (at least) 1 hour is

```
{
    "name": "interval", "value": {
        "$gte": 3600
    }
}
```

The comparison operators $<, >, \leq, \geq, =$ are available through the `$le`, `$gt`, `$lte`, `$gte`, and `$eq` strings. Table 5.1 shows the full list of operators.

At the time of writing, Gurret only has two update limiting filters; however, because of JSON's dynamic nature, it would be easy to imagine other options being available to end-users or even the option for end-users to create new ones.

Mathematical operator	JSON operator
$<$	`$le`
$\leq$	`$lte`
$>$	`$gt`
$\geq$	`$gte`
$=$	`$eq`

Table 5.1: Avalable comparison operators.

```
{
    "name": "sportsData.csv",
    "updates": [
        { "name": "Access", "value": 1644396257 },
        { "name": "Access-Label", "value": "private" }
    ]
}
```

Figure 5.2: Example message of the metadata update type.

5.3 Sending Subscription Messages to the Backend

Gurret and the backend exhange metadata information by writing *messages* to a socket connection. We define three types of message: `MetadataUpdate`, `GetSubscriptions`, and `SetSubscriptions`.

5.3.1 MetadataUpdate

The backend and Gurret use the `MetadataUpdate` message type to communicate metadata updates. The Gurret client receives metadata update messages when remote users change metadata for a shared file. Gurret sends metadata update messages backend when one of its metadata changes. The message uses JSON format and contains the name of the file and the updated fields and their values. Figure 5.2 show an example of a message of the `MetadataUpdate` type.

5.3.2 GetSubscriptions

`GetSubscriptions` is used to list the available subscriptions options for a file f. The Gurret client sends a `GetSubscriptions` message to the backend

```
{
    "name": "sportsData.csv"
}
```

Figure 5.3: Typical GetSubscriptions message.

```
{
    "name": "sportsData.csv",
    "options": [
        { "name": "Access", "filters": [
            { "name": "status", "value": "optional" }
        ]},

        { "name": "Access-Label", "filters": [
            { "name": "status", "value": "required" }
        ]},

         { "name": "Open-Count", "filters": [
            { "name": "status", "value": "optional" },
            { "name": "count", "value": "optional"}
        ]},
    ]
}
```

Figure 5.4: Typical GetSubscriptions response message.

with f's name and receives a `GetSubscriptions` message with the available subscription options for f. The response is encoded using the format we described in Section 5.1. Figure 5.3 shows an example of a `GetSubscriptions` message to be sent to the backend and Figure 5.4 shows a typical response. In the response message, two fields are optionally subscribable: `Access` and `Open-Count`, while `Access-Label` is required. Additionally, `Open-Count` have filters available.

5.3.3 SetSubscriptions

Gurret sends a `SetSubscriptions` message to the backend to subscribe or unsubscribe to (optional) metadata fields. The messages use the previously defined format (Section 5.1). The only mandatory filter when subscribing to metadata fields is the *status* field, which specifies if the user wants to subscribe or unsubscribe to a field. Figure 5.5 shows a basic set subscription message. In the message, the user subscribes to the `Access` field and unsubscribes to the

```
{
    "name": "sportsData.csv",
    "options": [
        { "name": "Access", "filters": [
            { "name": "status", "value": "subscribe" }
        ]},

        { "name": "Open-Count", "filters": [
            { "name": "status", "value": "unsubscribe" }
        ]}
    ]
}
```

Figure 5.5: Basic SetSubscriptions message.

`Open-Count` field.

The data owner can decide to set some metadata fields as *required*, and the Gurret client automatically subscribes to these fields. The remote ignores any attempt to unsubscribe to a required field. The users must explicitly subscribe to optional fields to get access to them.

5.4 How End-Users Subscribe to Metadata

Users can get and set subscription options using the provided Gurret CLI tool. The CLI command takes two positional arguments: (1) the operation, either `get` or `set`; (2) a JSON file with the message body. The CLI sends the message to the server and prints the response to the stdout. However, since the backend is not implemented, we mock the server and respond with a mock response. We imagine a website where users can manage their subscription options in the future. Suppose we have a file `options` with Figure 5.5's content. We set the subscription options by doing:

```
$ gurret set options
```

5.5 Metadata Consistency Model

Gurret uses eventual consistency since metadata updates can occur at arbitrary times– configurable by the subscription query language. Updates occur in no particular order, as long they do not involve the same metadata field. Receiving updates in the order

```
[file#1] [field] [new-value]
[file#2] [field] [new-value]

vs

[file#2] [field] [new-value]
[file#1] [field] [new-value]
```

Does not matter. However, receiving updates following order *does* matter:

```
[file] [field] [new-value#1]
[file] [field] [new-value#2]

vs

[file] [field] [new-value#2]
[file] [field] [new-value#1]
```

Gurret timestamps the messages sent to the end-users and discards older messages received out of order.

5.6 Summary

In this chapter, we explain that the pub-sub is a valuable model because of its scalability and natural integration in Gurret, where users request access to different metadata. Additionally, we outline how end-users describe their subscription options and the protocol between the Gurret clients and the backend—with the different message types—to coordinate users' subscriptions.

6 Data-Sharing and Notification Engine

The data-sharing and notification engine distributes files and acts as a broker for metadata subscriptions. One key trait for the backend is *scalability*, since it needs to handle potentially thousands of requests per second. Although the implementation of the backend is outside the scope of this thesis, we propose a design that draws inspiration from other distributed systems. This chapter outlines the design of the data-sharing service and the notification engine.

6.1 Data-sharing service

The data-sharing service stores and distributes data/files accessible to end-users. Some of the files in the data-sharing service might require special authentication. For instance, users might need to authenticate with an organization using OpenID [40] to access its files. We leave it up to the organizations to handle authentication.

We assume the data-sharing service runs on trusted hardware to avoid potential data leaks. Hosting data on untrusted devices is possible using encryption, as shown by OceanStore [41]; however, for simplicity, we assume *trusted* devices.

Data in the data-sharing service must follow the FAIR principle. That is, it must be **f**indable, **a**ccessible, **i**nteroperable, and **r**esusable.

The service is optimized for distributing read-only files, since only the metadata is mutable. Many other distributed system makes similar assumptions, like OceanStore's archive objects [41] and the work of Fu et al. [42].

6.2 Notification Engine

The notification engine receives metadata updates from Gurret and forwards them to the users it concerns. Our design has horizontal scaling in mind to meet demands in traffic with more instances. Additionally, we design the system to be able to work well with cloud computing services, such as Microsoft Azure [43], Amazon Web Services [44], and Google Cloud Platform [45]. That way, we can leverage the cloud computing services directly to automatically scale the number of instances for our system to meet demand. Additionally, basing the design on cloud services can, in many cases, especially for small and medium-sized businesses, be cheaper and more effective [46].

The notification engine has three primary services to collect and distribute metadata: (1) the distributed socket server; (2) the metadata store; (3) the query engine.

6.2.1 Socket Server

The socket server is a distributed server whose primary purpose is to communicate with the Gurret clients. The server uses asynchronous sockets to communicate with the end-users since updates can occur arbitrarily. The socket connections immediately yield after connecting to an end-user to preserve computing power and are restored when the socket becomes active. For instance, when an end-user or the backend writes to the socket.

6.2.2 Metadata Store

When an end-user sends metadata updates to the backend, the backend needs to store the metadata for later use. The metadata needs to be stored in a *distriuted database* since potentially many updates can occur at a given time from multiple different sources. Several distributed databases exist. For instance, DynamoDB [47], MongoDB[48], MySQL Cluster [49], and Cassandra [50]; however, because of metadata's key-value nature, using a document-based system, such as MongoDB, or key-value-based, such as DynamoDB, is very natural. Besides storing metadata, the metadata store also needs to track which users have access to which files. To this, we propose a simple JSON document that contains the name of the file and the users accessing it.

```
{
    "name": "sportsData.csv",
    "users": [
        ...
    ]
}
```

The simplest feasible way to represent the users in the JSON document is probably a static IP address or a MAC address. Alternatively, if we need more information about the users, we can store a unique identifier in the document leading to a separate JSON document for each user.

6.2.3 Query Engine

The query engine orchestrates update propagation to users. We design the engine to support multiple users with various subscription options efficiently. Multiple users subscribing to multiple metadata fields can quickly become convoluted. For instance, consider the situation depicted in Figure 6.1. The figure contains three users' subscription option. The first user subscribes to the `Access` metadata field and wants to receive updates after each 1000th access. The second user subscribes to (1) the `Access` field, and wants to receive each update; (2) the `Access-List` field and wants to receive updates every 10 minutes. The third user subscribes to the `Access-List` field and wants to receive an update for each 10th user access. Table 6.1 succinctly outline the users subscription.

To figure out which users need updates, the query engine stores each end-users metadata-filed options for every file on the backend. We propose a design where the backend has a document with the users and their options for each file.

```
{
    "name": <file>,
    "users": [
        // object for each user and their subscription options
    ]
}
```

```
/* user 1 SetSubscriptions */
{
    "name": "sportsData",
    "subscriptions": [
        { "name": "Access", "options": [
            { "name": "status", "value": "subscribe" },
            { "name": "count", "value": { "$gt": 1000 } }
        ]}
    ]
}

/* user 2 SetSubscriptions */
{
    "name": "sportsData",
    "subscriptions": [
        { "name": "Access", "options": [
            { "name": "status", "value": "subscribe" },
        ]},
        { "name": "Access-List", "options": [
            { "name": "status", "value": "subscribe" },
            { "name": "interval", "value": { "$gt": 600 } }
        ]}
    ]
}

/* user 3 SetSubscriptions */
{
    "name": "sportsData",
    "subscriptions": [
        { "name": "Access-List", "options": [
            { "name": "status", "value": "subscribe" },
            { "count": "status", "value": { "$gt": 10 } }
        ]}
    ]
}
```

Figure 6.1: SetSubscriptions with three users

User 1	Access	After each 1000 access
User 2	Access	When they occur
	Access-List	Every 10 minutes
User 3	Access-List	After each 10 user accesses

Table 6.1: Table representing Figure 6.1

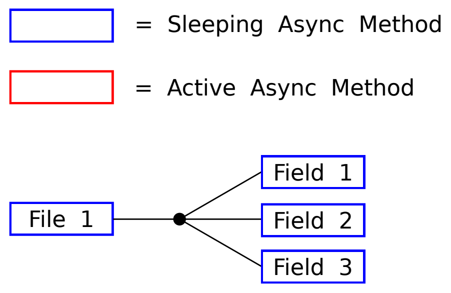

Figure 6.2: Hierarchy of async tasks idling

One weakness with the Query Engine's design is that files are represented as a singular object. This can potentially become a bottleneck if many users change their subscription options simultaneously. To mitigate this, we *shard* the documents over multiple servers to enable concurrent access [51]. Some databases, such as MongoDB, natively support sharding.

To determine when users should receive updates, we propose a design that has a hierarchy of asynchronous tasks. At the top level is a task for each file. The backend delegates an asynchronous task for each file that immediately yields. The backend wakes the task up when an update occurs on it. Inside the task, other tasks are delegated for each metadata field. The file task wakes the field task when an update occurs on the specific field. Finally, when a metadata field update occurs, the field task queries the database and figures out which users should receive updates. The updates are then sent to the socket server and communicated to the end-users. Figure 6.2 show the hierarchy of the asynchronous tasks idling. Figure 6.3 shows what happens when a field is updated, and end-users need to be notified about the update.

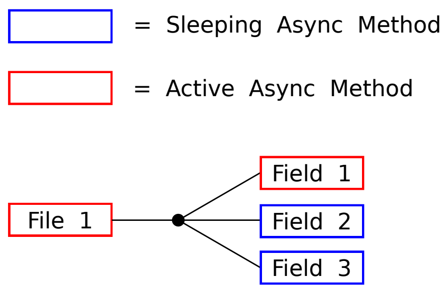

Figure 6.3: Hierarchy of async tasks on update

6.3 Summary

This chapter specifies the design of the Gurret backend system as required by the client. Implementation and verification of the backend system are outside the scope of this thesis.

7 Taint Tracking

In this thesis, we are particularly concerned with the class of metadata related to data management. Management metadata defines access and usage patterns allowed on files. Some examples of management metadata are access policies, usage policies, and processing policies. Importantly, on data propagation— when an *explicit information flow* occurs— these metadata fields need to propagate to the new file.

To show why this is important, consider a data usage policy that specifies that only one person can access a file at one instance. Suppose a flow occurs from a file *with* the policy to a file *without* the policy. In that case, the usage policy must *transfer* to the new file since accessing it implicitly accesses the original file. Suppose the management metadata does *not* transfer to the other file. In that case, more than one person could access the original file's content— since the content is located on both files from the flow— breaking the data usage policy.

This chapter outlines how Gurret handles this issue— by using taint tracking— and the rules Gurret enforces on processes to disallow policy violations resulting from an explicit flow. Finally, we implement a simple information flow control system to demonstrate how one could use Gurret.

7.1 Taint Tracking

Gurret employs taint tracking— a technique often used in information flow control— to solve the issue of policy violations occurring from explicit flows, like in the example above. In taint tracking, data is marked/tainted in order to follow the data throughout the system and to check and verify that the data is not misused [52]. Gurret taints processes that access files containing management metadata. There are two reasons for this: prevent situations such as the one described above and transfer management metadata when explicit flows occur.

7.2 Taint Tracking Rule

To prevent policy violations from occurring from an information flow— e.g., implicitly violating a file policy by accessing its content from two sources— Gurret only allows explicit flows from a source to a destination if the management metadata is also transferred. To express this more mathematically, we give the following definitions:

Definition 1.

$$\textbf{Let } \longrightarrow \textbf{ be an explicit flow between two files}$$

Gurret defines an *explicit flow* to be a process that (1) opens a file A and (2) writes the content to another file B. Using this notation, for two files A and B, we can write $A \longrightarrow B$ if a flow occurred from A to B.

Definition 2.

$$\textbf{Let } M(x) \textbf{ be the set of management metadata of } x$$

For instance, suppose a file f has an access label of `private`. We can express this as $M(f) = \{\langle \texttt{AccessLabel}, \texttt{private} \rangle\}$, where we use the tuple notation $\langle K, V \rangle$ to express a key-value metadata.

Definition 3.

$$\textbf{Let } \rightsquigarrow \textbf{ be the transfer of management metadata between two files}$$

An alternative way to express $\rightsquigarrow$ is with the definition $A \rightsquigarrow B \Rightarrow \forall x(x \in M(A) \Rightarrow x \in M(B))$

With this notation, we define Gurret's taint tracking rule to
Rule 1.

$$A \longrightarrow B \Rightarrow A \rightsquigarrow B$$

That is, the management metadata of the source file must propagate to the destination file whenever an explicit flow occurs. The metadata propagation should cascade down to every file affected. For instance:

$$A \longrightarrow B \longrightarrow C \longrightarrow D \Rightarrow A \rightsquigarrow B \rightsquigarrow C \rightsquigarrow D$$

A useful abstraction for the behavior in the rule is *transactions*. A transaction is a set of operations that are: *consistent*, *atomic*, and *durable* [53]. For us,the atomic property is very important, stating that either all operations occur, or none of them. For Gurret, in order for $A \longrightarrow B$, then $A \rightsquigarrow B$ *must* also occur. We define a *flow transaction* as the transaction of the operations $\longrightarrow$ and $\rightsquigarrow$ for two files.

Definition 4.

$$A \longrightarrow B \land A \rightsquigarrow B \Leftrightarrow \textbf{flow transaction from } A \textbf{ to } B$$
$$\textbf{For two files A and B}$$

Finally, we use the notation A `trans` B to mean a flow transaction from A to B.

Importantly, flow transactions need to be atomic because the result of one transaction can affect the next one. Suppose we have three files: A, B, and C. Suppose that A `trans` B leads to B `trans` C becoming illegal. This will lead to a policy violation if the operations are not atomic and we allow Gurret to interleave operations.

For instance, suppose that Gurret interleave the operations in A `trans` B and B `trans` C, to:

$$A \longrightarrow B$$
$$B \longrightarrow C$$
$$A \rightsquigarrow B$$
$$B \rightsquigarrow C$$

This would be a policy violation if A `trans` B invalidates B `trans` C.

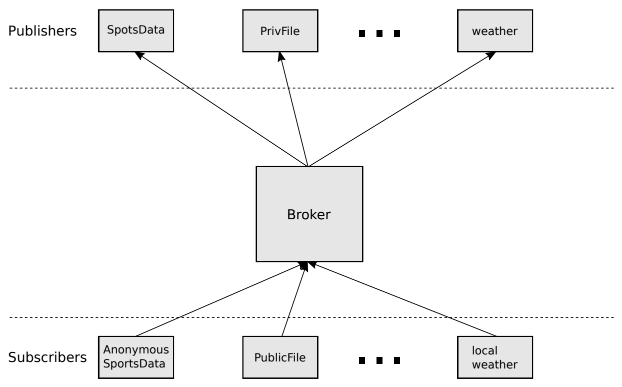

Figure 7.1: Architecture for handling flow transactions.

7.3 Implementing the Taint Tracking Rule

We use the publisher-subscriber model to implement Rule 1. Every file publishes its management metadata information to an internal broker. When $A \longrightarrow B$ occurs, Gurret transfer the metadata to B by automatically *subscribing* B to A's management metadata. When the metadata for a file f needs to be accessed, Gurret first checks to see if f owns the metadata. If it does not, Gurret then checks the metadata f subscribes to. Introducing pub/sub to implement the taint tracking rule is effective because it allows Gurret to manage subscriptions for files dynamically. For instance, many explicit flows can occur to and from files during Gurret's runtime. Using the pub/sub model allows files to alleviate the metadata management to the broker instead of each file managing its metadata. Figure 7.1 show the pub-sub architecture. The squares in the figure represent files, and the arrows indicate subscription. For instance, `Anonymous SportsData` might subscribe to `SportsData` through the broker. Notably, the subscribers in the figures are publishers; however, we omit this in the figure for clarity.

Internally, we implement the broker using a lookup table with structs of file metadata information, called *metadata source*. The metadata source includes the file's metadata and the metadata it subscribes to; in the form of an ID to the matching publishers. The broker creates IDs dynamically whenever files' metadata are published. Given two files A and B, Gurret can subscribe B to A's metadata by adding A's ID to the subscriptions of B. That is:

```
type MetadataId = u64;

struct MetadataSource
{
    subscriptions: Vec<MetadataId>,
    own: PathBuf
}

struct Broker
{
    available_id: MetadataId,
    table: Vec<MetadataSource>,
    table_index: HashMap<PathBuf, MetadataId>
}

impl Broker
{
    // ...
    fn subscribe(&mut self, file: PathBuf, source: PathBuf) { /* ... */ }
    fn publish(&mut self, file: PathBuf) { /* ... */ }
}
```

Figure 7.2: Lookup-table for data management metadata.

```
fn subscribe(broker: &mut Broker, A: PathBuf, B: PathBuf)
{
    let b_id = broker.table_id(B);
    let a_id = broker.table_id(A);

    broker.table[b_id].subscriptions.push(a_idx);
}
```

If an end-user creates or downloads a file, Gurret will automatically publish its management metadata to the broker. Publishing its management metadata is simple and only consists of creating a new metadata source, assigning an ID to it, and adding it to the broker's table. Figure 7.2 show a simplified code of the broker and metadata information.

A naive way to transfer metadata is to copy the entire file structure of the metadata and the metadata itself. This would free the need for a broker and table structure. Gurret does not adopt this approach for two reasons: (1) copying the whole file structure and metadata can be expensive. Depending on the number of metadata fields in the source file, the copying instructions can use significant time and resources. Additionally, copying the metadata uses more space since identical metadata is duplicated. Gurret will use N times more space than necessary if a shared file has a flow to N different files. (2) any upstream metadata changes on a file f need to be copied downstream to the files that received a flow from f. For instance, consider five files that share metadata with a common file and a metadata field for the common file changes. In that case, Gurret must copy the new metadata to all five files.

7.4 Tracking Data Lineage

The broker is well-suited for accessing metadata; however, it has poor *downstream* traversal. Downstream traversal is traversing information flow from a root file to the files that have received flows from it. In some instances, like if an upstream file is invalidated, Gurret might have to travel down the affected files and perform an action. To do this efficiently, Gurret needs another structure besides the broker to track data lineage.

7.5 Forest of Trees

To effectively traverse downstream, we create a *forest of trees* structure to track data lineage. The tree structure consists of root nodes representing files, with the node's children representing files where an information flow occurred. Figure 7.3 illustrates the structure. The nodes in the figure are individual files, and the arrows symbolize a flow from one file to another. Root nodes are files where no flows have occurred. By default, Gurret considers all files downloaded from the backend as root nodes.

We implement the forest of trees using a key-value store. The key is the file's path, like `~/gurret/downloads/sportsData.csv`, and the value is a node structure that contains information about the file. Figure 7.4 shows a *simplified* version of the forest of trees structure.

Internally, if a file has multiple data sources, both sources store the child in their internal data structure. Since we implement Gurret in rust, we use reference counting[1] to allow multiple sources to reference the same node.

1. `https://doc.rust-lang.org/std/rc/index.html`

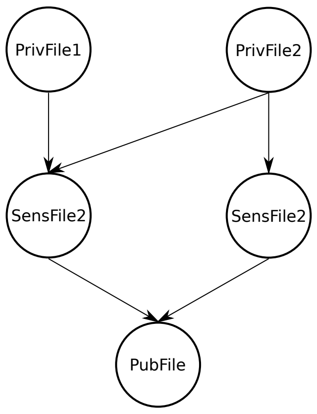

Figure 7.3: Forest of trees structure.

```
struct Node
{
    children:       HashMap<PathBuf, Node>,
    parents:        Vec<PathBuf>
}

struct ForestOfTrees
{
    roots:          HashMap<PathBuf, Node>
}
```

Figure 7.4: Simplified structure of the forest of trees.

7.6 Intercepting Information Flow Related Syscalls

Gurret updates and manages the broker and forest of trees whenever system calls related to information flow occurs. Many system calls can lead to an information flow, for various reasons. We limit this thesis to only consider the system calls `open`, `write`, `rename`, and `unlink`. The rest of this section explains and gives a mathematical/pseudo-code description of how Gurret handles each system call.

- `open`
  We denote the set of opened files for a process P with $\text{Files}(P)$. Gurret tracks every file that a process opens in an internal data structure. When P opens a file F, then Gurret adds F to P's opened files.

  $$P \;\; \texttt{open} \;\; F \Rightarrow \text{Files}(P) \leftarrow \text{Files}(P) \cup \{F\}$$

- `write`
  Gurret considers every `open` followed by a `write` to be an explicit flow from the open files of the process to the destination file being written into. If a process P writes to a file F, then Gurret considers this a flow from every file in $\text{Files}(P)$ to F.

  $$P \;\; \texttt{write} \;\; \text{to } F \Rightarrow \forall f \in \text{Files}(P) \Rightarrow f \rightarrow F$$

  The forest of trees and the broker's state are updated when an explicit flow occurs. The destination file is added as a child in the forest of trees for every opened file. For the broker, Gurret subscribes the destination file to every opened file.

  ```
  for open_file in process.open_files()
  {
      forest_of_trees[open_file].children.add(destination)
      broker.subscribe(destination, open_file);
  }
  ```

- `rename`
  Rename occurs when a file inside the filesystem container is renamed

or moved. The 'name' in `rename` refers to the full path of the file; thus, the system-call `rename` refers to a change in the file's path. For instance, both `mv foo bar` and `mv foo ../foo` are renames. Gurret changes the file's name in the forest of trees and broker if an upstream file is renamed or moved within the filesystem container.

- `unlink`
  An unlink occurs when a file is removed/deleted. For the forest of trees, because some metadata may result in changes downstream, Gurret *marks* the node representing the file as removed instead of removing the node entirely.

  ```
  if unlink
  {
      forest_of_trees[unlink_file].removed = true;
  }
  ```

  For the broker, if a file R is removed, then Gurret remove the table entry and ID for R and every subscription to R.

$$\forall f \in \{f \mid R \in \text{Subscriptions}(f)\} \Rightarrow \\ \text{Subscriptions}(f) \leftarrow \text{Subscriptions}(f) \setminus \{R\}$$

Gurret uses the forest of trees to find the files that subscribe to R.

One issue Gurret does not address is: what should happen to the children of the removed file? Removing them seems like a bad idea; since researchers working on collaborative work would not be happy if Gurret removes all their work. And removing only the content of the deleted file from the children is (1) not trivial and (2) would again leave users unhappy.

The alternative is to do nothing, but this could lead to a policy violation. For instance, if a new policy specified that the data is no longer available. Indeed, what should happen to the removed files is a delicate issue. We suggest a solution where the policy defines the removal strategy, but Gurret does not currently support this.

7.7 Storing the Forest of Trees and Broker

Gurret periodically flushes and stores the content of the forest of trees and broker to a *.forest* and *.broker* sidecar file to preserve the structures in case of a crash. Gurret stores both structures in JSON format and loads and re-instantiated the structs on startup.

7.8 Taint Tracking Weaknesses

One weakness with how Gurret classifies information flows— an open followed by a write— is that false-positive flows can occur. Consider a reclassification process that takes a file with a private label and strips it of all private information to create a public file. However, besides stripping the private info, it also logs the time to some logging file. If the `open` order of the program is:

```
let log = File::open("log");
let private = File::open("private");
let strip = strip_file_of_private_information(private);
let new_file = File::new("public");
write_into_new_file(new_file, strip);
```

Then Gurret would wrongly assume an information flow occurred from the log file to the public file, even though nothing from the log is present in the new file. We discuss a possible solution to this issue in Section 9.2.3

7.9 Implementing an Information Flow Control

As a proof of concept for taint tracking, we implement a simple IFC system using a reference monitor on top of the Gurret daemon. The reference monitor enforces a data access policy using labels. The access policy follow the Bell-LaPadula model [30] with the rules *no read up* and *no write down*.

7.10 Labels

We represent labels using a simple *lattice* structure. A lattice is a partially ordered set, where each element pair in the set have a greatest lower bound and a least upper bound. The lattice have the values `private`, `sensitive`, and `public`; with `private` having the *higest* value.

All files and processes have a label associated with them. For files, Gurret stores the labels in a custom metadata named `Access-Label`. Gurret store the labels of processes in an internal data structure. Initially, a process label equals the user's clearance level; however, Gurret taints the process if it reads a file with a lower label. Gurret store the user's clearance in a `.clearance` file inside the filesystem container.

7.11 Access-Control Rules

Following the Bell-LaPadula rules, users can only read files whose labels are lower or equal to their clearance. For instance, a user with a clearance of `private` can read files with a label lower or equal to theirs, like `public` and `sensitive`. The reference monitor enforces this by comparing the user clearance and the file label in the `open` system call and only giving access if the clearance is higher or equal. A permission denied error is returned if the clearance is lower than the file label.

Gurret enforces the *no write down* rule by comparing the (potentially tainted) label of the process and the label of the file being written into. There are two cases when writing to a file: the file is new, and the file already exists. If the file was new, Gurret gives the file a label equal to the process' (potentially tainted) label. If the file already existed, Gurret will only allow the write to occur if the process' label is lower or equal to the files'. For instance, a process with a `private` label cannot write its content to a file with a `public` label since this would mean *writing down*. Gurret determines whether a file is new or not by assuming any file without a label is a new file.

7.12 File Removal Policy

We experiment by defining what should occur to the children of removed files in the access policy and using the reference monitor to enforce it for demonstrative purposes. We use a simple policy that removes the children. To implement the removal policy, the reference monitor removes the children of the file in the procedure for updating the forest of trees. The removal function is recursive such that all files with an ancestor to the original file are removed.

7.13 Information Flow Control System Use Cases

Users can deploy Gurret's IFC system in the class of problems where the Bell-LaPadula model fits. For instance, a natural place to use the system would be in a medical research institution where collaborative work occurs on restricted files involving private patient information. In such an environment, only researchers with a high enough clearance can work on the files in the system.

Another example is in a military environment where data confidentiality is critical. Only personnel with the correct label can access confidential files using our system. More importantly, data leaking to a lower level (e.g., from `secret` to `public`) is impossible. Finally, if an external system marks a data source as invalid, then Gurret will automatically remove the affected files.

7.14 Summary

In this chapter, we explain how Gurret uses taint-tracking and IFC techniques to enforce metadata propagation. We go into depth about how Gurret intercepts and handles systems calls and what internal data structure Gurret uses. Finally, we give some example scenarios where one could use the Gurret system.

8 Evaluation

One of the non-functional requirements of Gurret is to be transparent to the user with respect to performance. This enforces strict limitations in execution times for certain operations. For instance, opening a file in the filesystem container should not take noticeably longer than in the regular filesystem. We evaluate and adjust our implementation accordingly to verify that our system upholds these non-functional requirements. We measure FUSE and metadata overhead, filesystem performance, and subscription overhead. This chapter outlines the benchmark and evaluation results and potential adjustments to Gurret.

8.1 Hardware Specification

All benchmarks are run on the following hardware:

OS	Debian GNU/Linux 11 (bullseye) x86_64
Kernel	5.10.0-9-amd64
CPU	Intel i7-8700T (12) @ 4.000GHz
GPU	Intel CometLake-S GT2 [UHD Graphics 630]
RAM	16 GB
Storage device type	SSD
Native Filesystem	ext4

8.1.1 Benchmarking Method

Most benchmarks involve creating a file of some size S and benchmarking an operation's performance, such as reading and writing. We benchmark how the system scale by performing the operation for different file sizes. The file size is double the previous file size. The form of the file size is 2^N UB, where N is the number from 1 up to some upper limit L, $N \in \{1, 2, \ldots, L\}$. The variable U denotes the unit of the file size. In some benchmarks, we use bytes, and in other benchmarks, mega-bytes. We run each benchmark at least 30 times to get the most consistent measurements.

8.2 FUSE Overhead

Using FUSE comes with a time cost due to intercepting system calls. We benchmark FUSE compared to the native filesystem to find this cost. Since reading and writing operations are widespread when working with files, we benchmark and compare the reading and writing speed of FUSE compared to the native filesystem (ext4).

8.2.1 Setup

We write 2^N MB of data, where $N \in \{1 \ldots 10\}$, into a newly created file with the same size to benchmark the writing performance. The reading performance is measured by reading the same file back.

8.2.2 Results

Figure 8.1 and Figure 8.2 show the results from the benchmarks.

The reading results show that FUSE is just as fast as the underlying filesystem, up until 512 MB. However, FUSE is increasingly slower in the writing benchmark. To see how much, we plot the graph in Figure 8.3, only with a logarithmic y-axis. The graph shows that both graphs scale exponentially, with FUSE having a higher exponent.

We tried to increase the writing speed by enabling asynchronous read/writes and increasing the `max_write` size; however, this made no noticeable difference when re-running the benchmarks.

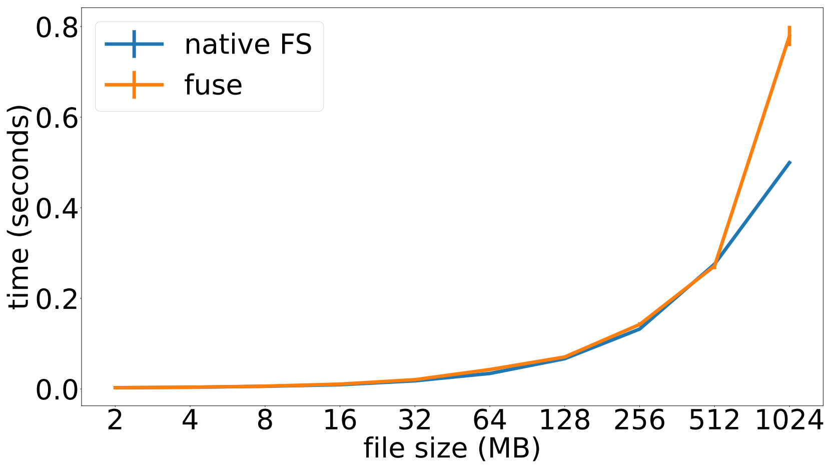

Figure 8.1: FUSE overead reading.

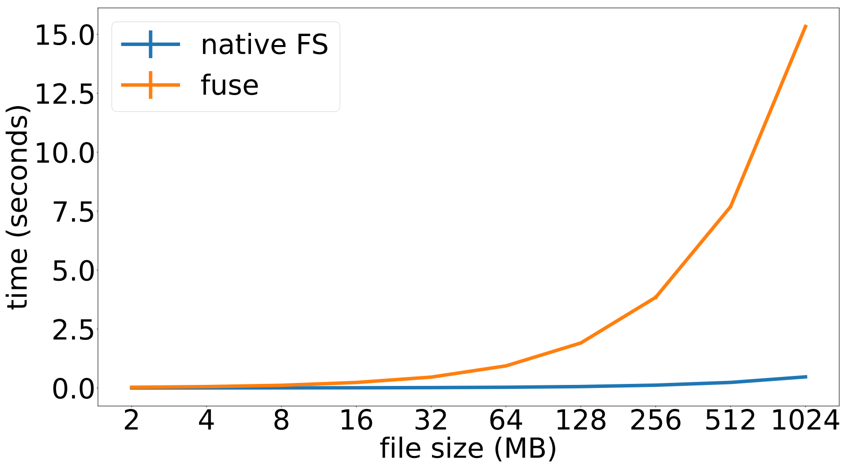

Figure 8.2: FUSE overead writing.

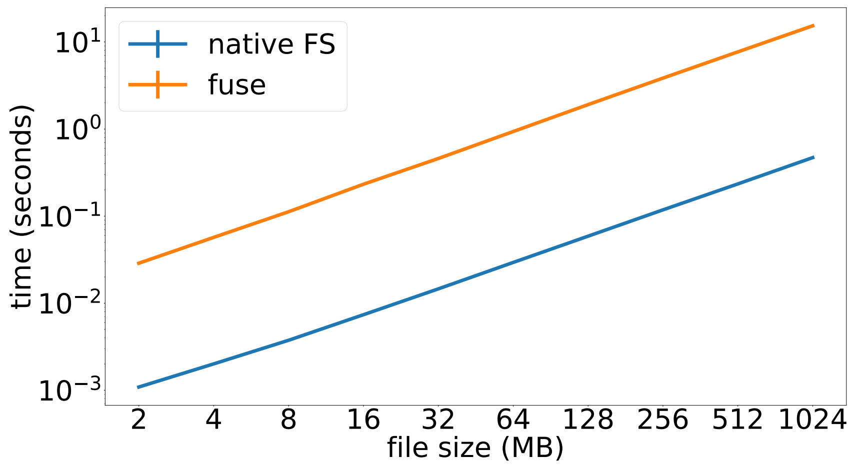

Figure 8.3: FUSE overhead writing, logarithmic y scale.

8.3 Metadata Benchmarks

We check how much metadata a file can have without having a noticeable slowdown. We define *noticeable* to be over half a second. Card et al. [54] define *immediate response* as one second; however, we believe the modern users have become more accustomed to faster than one-second responses. At least in applications not bound by remote server connections. We split this section into two benchmarks: (1) the time it takes to check whether an update occurred; (2) the time it takes to update metadata (in the case of an update).

8.3.1 Setup

We benchmark how metadata scales. We start with an empty file with one custom metadata and record the time it takes to: (1) check the field; (2) update the field. Then, we create a new file with two custom metadata fields, and so forth, up to and including ten. Additionally, we run the test N times for each benchmark for reasons that will become apparent later. For instance, we trigger and update metadata for the first file with one custom metadata N times. We use an N of 11 in our benchmarks. That means that Gurret will check and update the metadata for a file f 11 times for each benchmark.

We use the simplest possible custom metadata in the benchmark. The checking meta-code immediately returns `"true"`, and the update meta-code updates the same field with a constant value.

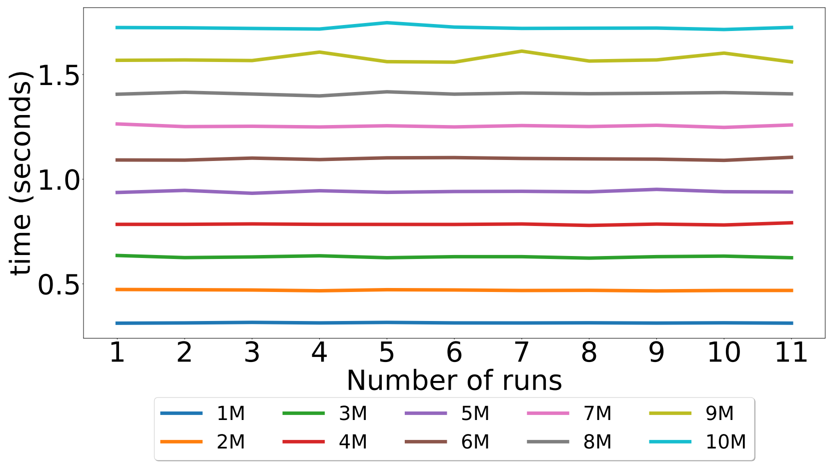

Figure 8.4: Metadata checking time.

```
// checking meta-code
fn main()
{
    println!("true");
}

// update meta-code
fn main()
{
    let file = // ...
    update_field(file, "field",  "0");
}
```

8.3.2 Checking Metadata Result

Figure 8.4 show the result of the metadata check benchmark. The label *N*M is the benchmark for the file with *N* (custom) **M**etadata. For instance, 3M represents a file with three custom metadata.

We can see that the operation is noticeable at three custom metadata. This is because Gurret *compiles* the checking meta-code each time, which is costly. To mitigate this, Gurret can store the compiled meta-code for later use in a cache. This assumes the meta-code does not change during runtime. However, Gurret can re-compile the meta-code if it changes. Figure 8.5 shows the resulting graph from caching the check meta-code.

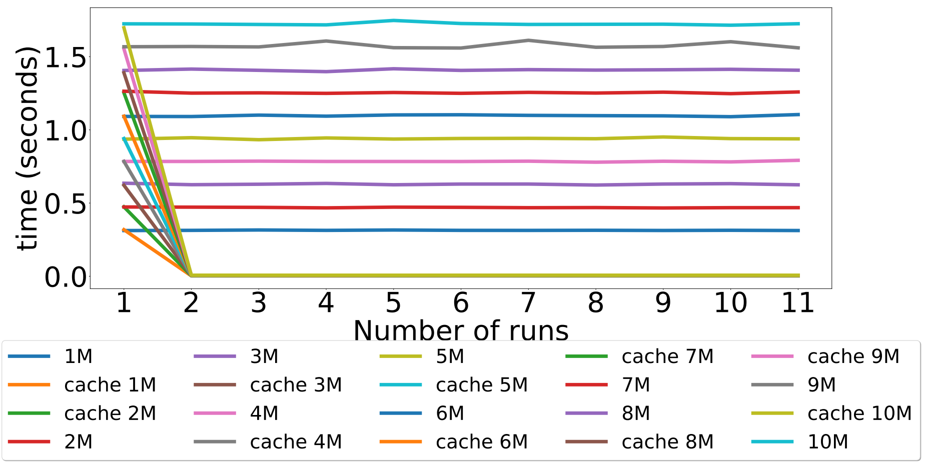

Figure 8.5: Custom metadata scaling with check store.

Although the first run is just as slow, the following is close to zero using this method. To further improve this and remove the first hunch, Gurret can preemptively compile each meta-code on a separate thread during startup or runtime.

8.3.3 Updating Metadata Result

We have a similar result for updating metadata. Initially, Gurret compiles the updating meta-code each time it executes. However, when we introduce a cache, the time is significantly improved. Figure 8.6 shows the result of the benchmark. For clarity, we only display the benchmarks for ten custom metadata since the results closely resemble the previous benchmark.

Again, the first run is noticeable slow; however, preemptive compilation can improve run times. Alternatively, using an implementation with pre-compiled binaries or an interpreted language such as python can also mitigate the initial compile time.

8.4 Filesystem Benchmark

We benchmark the reading and writing speeds of different filesystems. Different filesystems have other pros and cons. We want to determine if switching to an alternative filesystem better suited for Gurret's workload can improve

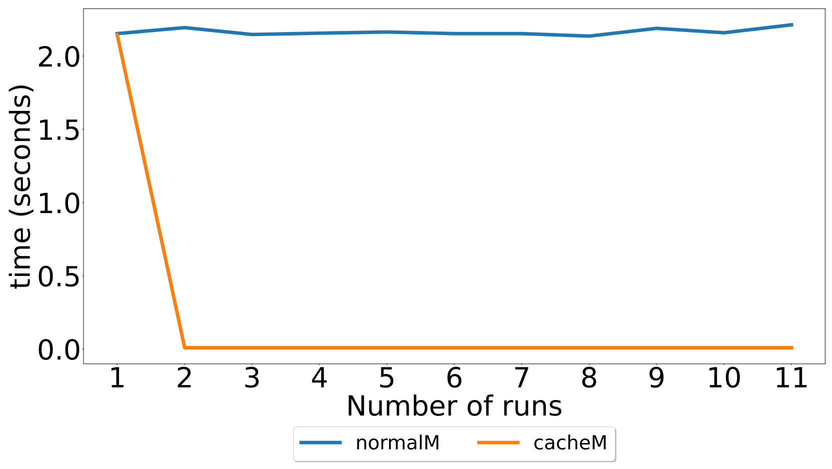

Figure 8.6: Metadata update time.

performance. This section benchmarks reading and writing for *files* and reading and writing for *extended attributes*.

8.4.1 Benchmarking Setup

The filesystems we consider are ext4, XFS, Btrfs, and F2FS. We create a filesystem container and run the benchmarks inside it for each filesystem. For the extended attribute benchmark, we write and read 2^N bytes, where $N \in \{1, \ldots, 10\}$, into the extended attribute of a file f. For the file benchmark, we write and read 2^N MB into a file f, with the same N.

8.4.2 File Benchmark Result

Figure 8.8 and Figure 8.7 show the file reading and writing benchmarks.

F2FS performs significantly better than the other filesystems on the writing benchmark. This is likely because the host system uses an SSD to store data, which F2FS specifically targets. On the other hand, the filesystem with the poorest scalability overall was the filesystem we currently run, ext4.

F2FS had the best performance on the reading benchmark, but only marginally. The most surprising result, however, was Btrfs. Btrfs has very poor reading performance compared to the other file systems. The difference between Btrfs and other filesystems is that Btrfs is a copy-on-write filesystem with modern

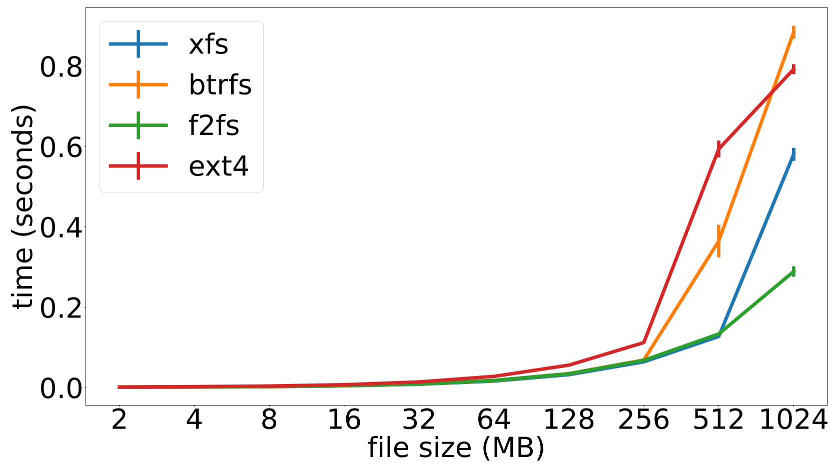

Figure 8.7: Filesystem writing benchmark.

data integrity features, such as *checksums* and *checkpoints*. Whenever Btrfs reads a data block, a checksum is calculated to verify integrity. We suspect this is one of the reasons Btrfs has a slower reading performance. Turning this feature off *is* possible; however, it would defeat one of the main design points of Btrfs. Therefore, comparing Btrfs in this regard is arguably not a fair comparison.

8.4.3 Extended Attributes Result

Figure 8.9 and Figure 8.10 show the results for the extended attributes benchmarking result. The different filesystems have very similar performance, and no particular filesystem performed better or worse than the others. This is despite the fact that some of the filesystems support inline extended attributes. The similar behavior is likely because the content size— 1 up until 1024 bytes— is not so large. We wanted to test for larger sizes; however, some filesystems have limitations on the extended attribute size.

8.5 Filesystem Benchmark Result

The most performant filesystem in the benchmarks is the F2FS filesystem. F2FS performs better or just as good as the other filesystems on the benchmarks, making it the best choice. However, the hardware we performed the benchmarks on was an SSD– F2FS main target. We have not done the same benchmarks

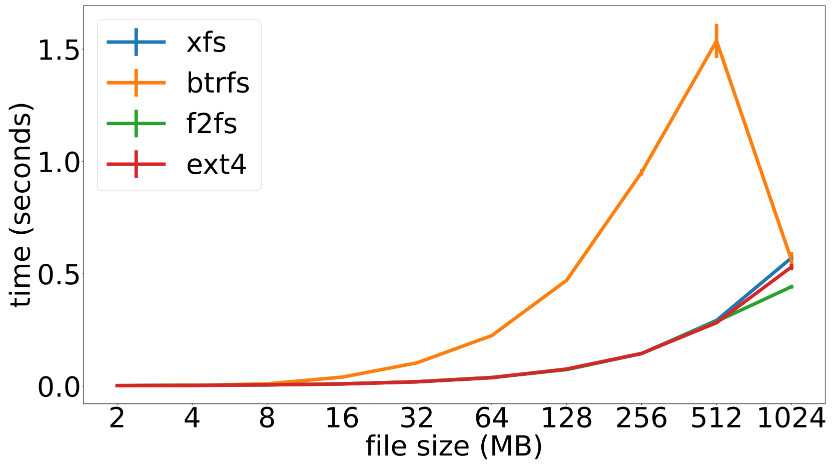

Figure 8.8: Filesystem reading benchmark.

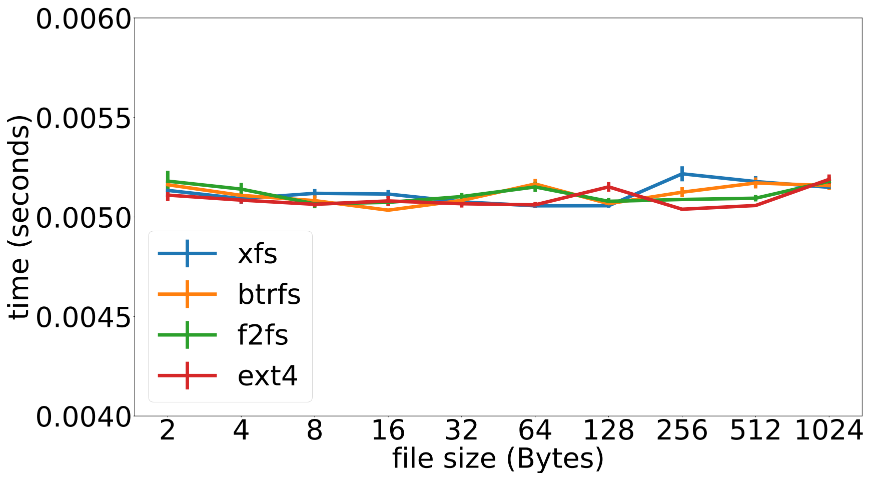

Figure 8.9: Filesystem extended attributes writing benchmark.

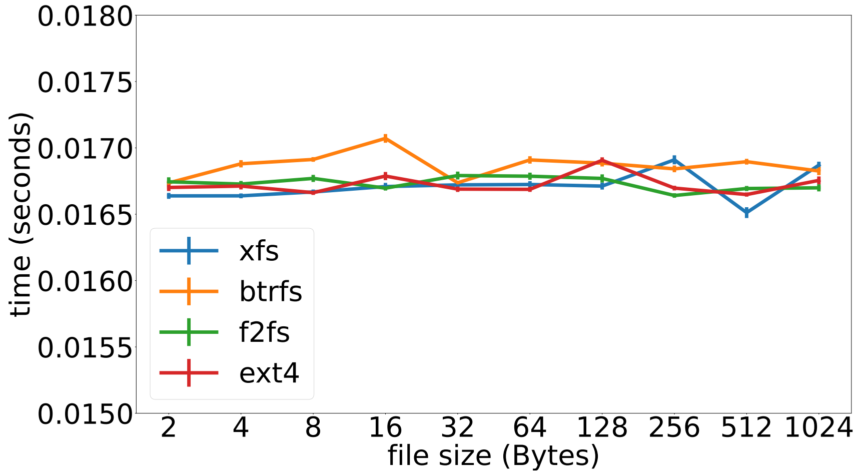

Figure 8.10: Filesystem extended attributes reading benchmark.

using an HDD. Whether these results also apply to HDD is therefore not known. Ext4 and XFS performed better than F2FS on a HDD in SQLite insertions in 2018, according to Phoronix [1]. But this is a different workload. Further tests are needed to get a definitive answer.

8.6 File Subscription Benchmarks

Files whose content comes from an information *flow* subscribe to the file metadata of the source. We want to investigate how many subscriptions a file can have before end-users notice. Again, we define *end-users notice* as over half a second. We benchmark the time it takes for a file to check its metadata and the metadata it subscribes to.

8.6.1 Benchmarking Setup

We create 100 files, each with a simple check meta-code that returns true. The files are: $f_0, f_1, \ldots, f_{99}$. The first file, f_0, does not subscribe to any file metadata. File f_n, subscribe to the metadata of every file with a smaller n. That is, $f_{n-1}, f_{n-2}, ..., f_0$. With this, a given file f_n has the metadata of itself, and every file with a smaller n.

1. `https://www.phoronix.com/scan.php?page=article&item=f2fs-hdd-test&num=2`

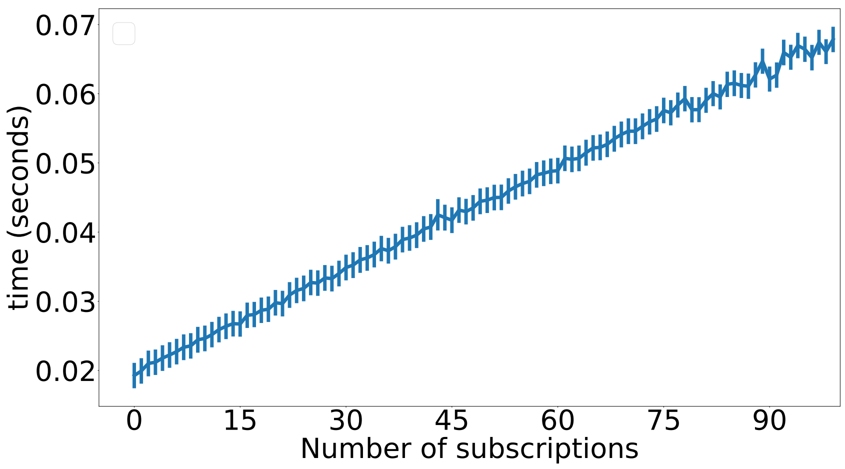

Figure 8.11: Time to check metadata for files that subscribes to n metadata.

We compile every meta-code before running the benchmarks by triggering an update for the last file. We do this to avoid compiling each file's check meta-code, which would pollute the benchmarks.

8.6.2 Benchmarking Result

Figure 8.11 shows the results of the benchmark. The benchmarks show a clear linear relationship. We consider the performance of 70 milliseconds for 100 subscriptions well under an acceptable level. We imagine that a file with over ten subscriptions is pretty uncommon, let alone 100. The scalability is linear, as expected since the algorithm is linear. By averaging the difference in time between file n and $n-1$, we find that an additional subscription, on average, adds 0.485 milliseconds.

8.7 Summary

In this chapter, we benchmark and evaluate the Gurret client. We consider the FUSE overhead, metadata scalability, filesystem performance, and file subscription scalability. We adjust or implement when necessary, for instance, the meta-code cache.

9 Conclusion

This chapter discusses related work to Gurret, future work, and our concluding remarks, summarizing the problem statement and our important findings.

9.1 Related Work

Gurret borrows and implements many common ideas in different fields within computer science and systems. The four main fields are *distributed data management*, *filesystems*, *publisher-subscriber model*, and *information flow control* systems. The following sections will discussion common ideas between and those found in the research litterature.

9.1.1 Distributed Data Management

Similar to Gurret, Dynamic Metadata Management for Petabyte-Scale File systems (DMM-FS) separates file data and metadata [55]. DMM-FS stores metadata on disk in a cluster of metadata servers. Metadata accesses are subsequently read from disk; however, unlike Gurret, DMM-FS utilizes an in-memory cache to retrieve hot metadata quickly. Metadata is distributed across the cluster such that end-users doing typical workloads (Scientific workloads, general computing workloads, etc.) can access and utilize multiple servers simultaneously.

OceanStore [41] and Gurret both distribute and shard data across servers. OceanStore is designed to be run on potentially untrusted hardware, unlike Gurret, which assumes trusted devices. Gurret shard metadata across multiple machines for high availability, similar to how OceanStore shard archive/read-only document; however, OceanStore shard data to several orders of magnitude more servers.

9.1.2 Filesystems

Perhaps one of the papers Gurret shares the most with is the Low-bandwidth Network File System [56] (LBFS). LBFS divides files into smaller *chunks*, which are indexable via a hash value. LBFS efficiently sends files of a low-bandwidth network by only sending the chunks an end-user does not have. Initially, the server sends every chunk in the file; however, only the changed chunks are transferred as the file changes. Similarly, Gurret only sends the metadata that changed over to the end-users. LBFS uses a timer-based *read lease* to determine if an update has occurred. A *read lease* is a contract between the server and client that ensures the client receives updates for a file as long as the lease is active. Read leases are similar to Gurret's `interval`, only the inverse. Instead of promising updates *after* an interval, read leases promises updates *within* the time interval. End-users have a local cache in memory containing chunks. End-users check the local cache for chunks before sending a request to the server. LBFS can reconstruct files with similar chunks using the chunks in the cache from another file. For instance, LBFS can use license declaration chunks in several files with license text, e.g., GPL, MIT. Reconstructing/using data from another file to save resources is similar to how Gurret avoids storing metadata for derived files by subscribing files to their source files' metadata. The consistency LBFS provides is *close-to-open* consistency. Changes for a file that has been closed are guaranteed to be reflected in subsequent opens for other users. Gurret differs by having a more relaxed consistency model; configurable by the `count` and `interval` subscription options.

NFS version 4 [57] uses a `COMPOUND RPC` procedure to group multiple filesystem operations together. Instead of sending operations such as `LOOKUP`, `ACCESS`, and `READ` individually, NFS4 batch uploads them, sending them in one request. Similar to how Gurret batch uploads local metadata changes.

LoNet [36] and Gurret both use meta-code. Gurret uses meta-code to implement the metadata interface 4.1, while LoNet link them in a *policy file* that specifies what meta-code should run in different events and transitions.

9.1.3 Publisher-Subscriber Model

Gurret's metadata-field filters are similar to Siena's [38] filters and patterns; however, Gurret cannot use the field as an expression. While Gurret can receive updates when a field has changed N times, or after an interval of T, it cannot receive updates based on the field F. The architecture we propose is similar to Siena's Hierarchical Client/Server Architecture, where our version of the master/root servers is the database storing metadata information.

9.1.4 Information Flow Control Systems

The PASS storage system [58] detects and stores data lineage for files stored in it, similar to how Gurret tracks lineage inside its filesystem container. PASS stores lineage information as a DAG, unlike Gurret, which stores the forest of trees. Although not common, in Gurret, two files can subscribe to each other, unlike in a DAG. One key difference between PASS and Gurret is that PASS provides utilities to query the lineage information. Chimera [59] provides a similar service as PASS and Gurret and lets users query lineage information. Chimera differs from Gurret in that they store lineage information in a SQL database, unlike Gurret, which uses a sidecar file. Although more complex, this storage option scales better as the number of lineage information increases. Chimera does not specify how data dependency information is produced, unlike PASS and Gurret. Perhaps the most similar to Gurret is SPADE [60], which, just like Gurret, leverages FUSE in order to track data lineage. SPADE defines an interface to store data, allowing *any* form of persistent storage to be used. This lets end-users optimize how data is stored based on the expected workload. For instance, end-users can use Neo4j [61, 62] if they expect to do many graph-like operations, such as joins.

9.2 Future Work

Although significant work has been done on the Gurret client, more work is needed to have a complete system. This section explores parts of Gurret that could be improved and parts that are missing.

9.2.1 Filesystem Container

One weakness with Gurret's filesystem container is that it is possible to move files outside the container. Simply executing `mv file <outside-location>` would completely bypass any information flow control check and, in some cases, could crash the Gurret daemon. For instance, consider a custom metadata field that collects information about a file's data sources. If the source is moved from the container without the Gurret daemon noticing, the metadata might crash because of an *file not found* error. Solving this would involve making the Gurret daemon aware of `rename` system calls (e.g., `mv`) outside the container. For instance, one solution might be to mount the whole root or home directory instead of only the filesystem container. This, however, could introduce other issues related to security and data integrity if Gurret is allowed full access.

9.2.2 Filesystem

The filesystem container uses ext4 as its primary filesystem despite the benchmarks in Chapter 8 showing that F2FS is the fastest. We continue to use ext4 because we do not have a comparison for the benchmarks with an SSD and an HDD. We need additional benchmarks running the same tests on HDDs and SSDs alike before switching and committing to a different filesystem such as F2FS.

9.2.3 False-Positive Information Flows

Gurret wrongly assumes that a flow occurred from one file to another in some instances. For instance, if a process opens a log file before doing an explicit flow, then Gurret would assume a flow from the log file as well. This is the result of what Gurret constitutes as a flow. One solution to this problem is to use a config file to specify which files a process opens should ignore. For instance, something similar to `ignore = ["log", "token"]`. However, this is not so elegant since this might require end-users to fiddle with config files. A more interesting solution would be to: (1) prepend the data with some identifier to the data source whenever a read occurs; (2) when writing data to a file, check if the data is prepended with an identifier(s) and only add the file as children of the identifiable file(s).

9.2.4 Gurret Daemon

The Gurret client running on top of the filesystem container have poor reading and writing performance for larger files. A better implementation of the filesystem or an entirely different method of intercepting and handling system calls can probably reduce the overhead associated with FUSE.

9.2.5 Distinguishing between Data Management Metadata

Currently, Gurret copies all dynamic metadata over to the destination file when an information flows occurs. Indeed, there are many ways to distinguish between metadata that should and should not propagate to new files. One of the simplest ways would be to copy all metadata where a file named `data-management` was present. However, a more sophisticated system might be more appropriate if Gurret is expected to copy different metadata in certain situations.

9.2.6 Distributed Data-Sharing and Notification Service

The backend data-sharing and notification service is the most significant part of Gurret left unimplemented. The implementation would need to consider several hard questions within distributed systems. For instance, how should the backend handle:

- Two or more updates to a field at the same time
- Consistency conflicts
- Replication
- Fault Tolerance
- Geo-distribution

And probably much more. Luckily, many distributed databases implement solutions to some of these issues, such as MongoDB; however, a decision on the configuration is still required. Additionally, Sharma et al. [63] outlies many additional relevant requirements for a system like Gurret, some of which include logging each operation and automatic garbage collection for data contained on unsolicited machines.

We can only do a complete benchmark once the backend is implemented. For the full system benchmark, we propose benchmarking and testing the following:

- **Benchmarks for the backend**: How long do common operations such as Set/Get-Subscription take, and how does it scale.
- **Finding and sending updates to end-users**: For a popular file, when an update occurs, how long does it take to find the recipients of the new

update.

- **Scaling**: Test the overall scalability of the system as the number of users, files, and subscription options increase.

- **Stress test**: Stress testing the whole system to figure out how many concurrent users/request it can handle.

Without these benchmarking results, a complete system evaluation is not possible.

9.3 Concluding Remarks

Our thesis, as stated in Section 1.1, was to show that scalable decentralized data management is possible using pub-sub at the filesystem level. We show that this is possible by prototyping and benchmarking the Gurret client: the end-user CLI to interact with the Gurret system.

Our findings suggest that pub-sub is a valuable abstraction for distributing and querying metadata for several reasons: (1) Metadata for a particular file can be represented as topics and is easy to represent and store because of its key-value nature. (2) End-users can use the subscription language to specify—with fine granularity— which fields they are interested in and how often they want to receive updates. (3) The pub-sub model is shown to be highly expressive, extensive, and scalable [38]; all traits that are desirable in a distribution service for potentially thousands of files and end-users.

Gurret efficiently propagates metadata whenever information flows occur by using taint tracking and the pub-sub model. Custom metadata is implemented using meta-code, which allows for highly specialized metadata, such as data management metadata. Our evaluation shows that Gurret supports hundreds of propagated metadata with minimal overhead. Finally, we implement a reference monitor to enforce an access policy based on the Bell-LaPadula model.

Although not complete, we show the Gurret client and Gurret system can be a widely used distributed system for files following the FAIR principles. Because of Gurret's unique concept of metadata distribution, Gurret could be especially relevant in environments where metadata operations are an important part of the workflow.

Bibliography

[1] Katrin Braunschweig, Julian Eberius, Maik Thiele, and Wolfgang Lehner. The state of open data limits of current open data platforms. 2012.

[2] Marijn Janssen, Yannis Charalabidis, and Anneke Zuiderwijk. Benefits, adoption barriers and myths of open data and open government. *Information Systems Management*, 29:258 – 268, 2012.

[3] Aakash Sharma, Thomas Bye Nilsen, Katja Pauline Czerwinska, Daria Onitiu, Lars Brenna, Dag Johansen, and Håvard Dagenborg Johansen. Up-to-the-minute privacy policies via gossips in participatory epidemiological studies. *Frontiers in Big Data*, 4:14, 2021.

[4] RDA FAIR Data Maturity Model Working Group et al. Fair data maturity model: specification and guidelines. *Research Data Alliance. DOI*, 10, 2020.

[5] Danny Brooke. Community built infrastructure: The dataverse project. In *EGU General Assembly Conference Abstracts*, page 12006, 2020.

[6] Erin D Foster and Ariel Deardorff. Open science framework (osf). *Journal of the Medical Library Association: JMLA*, 105(2):203, 2017.

[7] Aakash Sharma, Katja P Czerwinska, Lars Brenna, Dag Johansen, and Håvard D Johansen. Privacy perceptions and concerns in image-based dietary assessment systems: Questionnaire-based study. *JMIR Hum Factors*, 7(4):e19085, Oct 2020. ISSN 2292-9495. doi: 10.2196/19085. URL `http://humanfactors.jmir.org/2020/4/e19085/`.

[8] Drew Roselli, Jacob R Lorch, and Thomas E Anderson. A comparison of file system workloads. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, 2000.

[9] Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *2019 IEEE Symposium on Security and Privacy*

(SP), pages 490–504, 2019. doi: 10.1109/SP.2019.00043.

[10] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[11] Benjamin Davis and Hao Chen. {DBTaint}:{Cross-Application} information flow tracking via databases. In *USENIX Conference on Web Application Development (WebApps 10)*, 2010.

[12] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[13] Aakash Sharma, Thomas Bye Nilsen, Lars Brenna, Dag Johansen, and Håvard D Johansen. Accountable human subject research data processing using lohpi. 2021.

[14] Peter J. Denning, Douglas E Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R Young. Computing as a discipline. *Computer*, 22(2):63–70, 1989.

[15] Fu-Hau Hsu, Min-Hao Wu, Syun-Cheng Ou, and Shiuh-Jeng Wang. Data concealments with high privacy in new technology file system. *The Journal of Supercomputing*, 72(1):120–140, 2016.

[16] Thomas Göbel, Jan Türr, and Harald Baier. Revisiting data hiding techniques for apple file system. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–10, 2019.

[17] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[18] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[19] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[20] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger,

Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[21] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, San Jose, CA, February 2013. USENIX Association. ISBN 978-1-931971-99-7. URL `https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu`.

[22] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[23] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *Proceedings of the Linux Symposium*, pages 395–408. Citeseer, 2006.

[24] Matthew E Hoskins. Sshfs: super easy file access over ssh. *Linux Journal*, 2006(146):4, 2006.

[25] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To {FUSE} or not to {FUSE}: Performance of user-space file systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 59–72, 2017.

[26] RS Sandhu, EJ Coyne, HL Feinstein, and CE Youman Role-Based. Access control models. *IEEE computer*, 29(2):38–47, 2013.

[27] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research . . . , 2003.

[28] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[29] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[30] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE CORP BEDFORD MA, 1973.

[31] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

[32] Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997.

[33] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 95–106, 2011.

[34] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[35] Andrew Myers, Owen Arden, Tom Magrino, et al. Jif 3.5: Java information flow, 2016. URL `https://www.cs.cornell.edu/jif/`. Accessed: 2022-Jan-03.

[36] Håvard D Johansen, Eleanor Birrell, Robbert Van Renesse, Fred B Schneider, Magnus Stenhaug, and Dag Johansen. Enforcing privacy policies with meta-code. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, pages 1–7, 2015.

[37] Dag Johansen and Joseph Hurley. Overlay cloud networking through meta-code. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 273–278. IEEE, 2011.

[38] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.

[39] Masoud Mansouri-Samani and Morris Sloman. Gem: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96, 1997.

[40] David Recordon and Drummond Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, pages 11–16, 2006.

[41] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGOPS Operating Systems Review*, 34(5):190–201, 2000.

[42] Kevin Fu, M Frans Kaashoek, and David Mazieres. Fast and secure distributed {Read-Only} file system. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[43] Roger Jennings. *Cloud computing with the Windows Azure platform*. John Wiley & Sons, 2010.

[44] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182, 2016. doi: 10.1109/CCGrid.2016.37.

[45] Ekaba Bisong. *Building machine learning and deep learning models on Google cloud platform: A comprehensive guide for beginners*. Apress, 2019.

[46] Maricela-Georgiana Avram. Advantages and challenges of adopting cloud computing from an enterprise perspective. *Procedia Technology*, 12:529–534, 2014.

[47] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[48] Kristina Chodorow. *Scaling MongoDB: Sharding, Cluster Setup, and Administration*. " O'Reilly Media, Inc.", 2011.

[49] Ammar Fuad, Alva Erwin, and Heru Purnomo Ipung. Processing performance on apache pig, apache hive and mysql cluster. In *Proceedings of International Conference on Information, Communication Technology and System (ICTS) 2014*, pages 297–302, 2014. doi: 10.1109/ICTS.2014.7010600.

[50] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[51] Yimeng Liu, Yizhi Wang, and Yi Jin. Research on the improvement of mongodb auto-sharding in cloud environment. In *2012 7th International Conference on Computer Science Education (ICCSE)*, pages 851–854, 2012. doi: 10.1109/ICCSE.2012.6295203.

[52] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 39–50, 2008.

[53] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[54] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181–186, 1991.

[55] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. Dynamic metadata management for petabyte-scale file systems. In *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 4–4. IEEE, 2004.

[56] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[57] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[58] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *Usenix annual technical conference, general track*, pages 43–56, 2006.

[59] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings 14th International Conference on Scientific and Statistical Database Management*, pages 37–46. IEEE, 2002.

[60] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 101–120. Springer, 2012.

[61] Mattias Finné, Anders Nawroth, Misha Demianenko, Pontus Melke, et al. Neo4j enterprise edition 4.4.5. Software, Neo4j, Inc., March 2002. URL `https://github.com/neo4j/neo4j`.

[62] Florian Holzschuher and René Peinl. Performance of graph query lan-

guages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204, 2013.

[63] Aakash Sharma, Thomas Bye Nilsen, Sivert Johansen, Dag Johansen, and Håvard D Johansen. Designing a service for compliant sharing of sensitive research data. In *International Conference on Risks and Security of Internet and Systems*, pages 155–161. Springer, 2022.