SINGIDUNUM UNIVERSITY

TECHNICAL FACULTY

Implementation of a Distributed Code Versioning System

diploma thesis

Mentor: **Candidate:**

Prof. Dr. ***Miodrag Živković*** ***Željko Stojković***

Belgrade, ***2025*****.**

Content

Introduction

Version control systems (VCS) are among the essential tools used in modern software development, allowing developers to track and store changes made to files over time. Version control systems enable developers to maintain history, revert to an earlier state and develop projects collaboratively without overwriting each other's work. Despite being fundamental to software development, internally, how version control systems operate remains abstracted from developers using them daily.

Lack of knowledge of how version code systems work leads to problem that developers often treat these systems as "black boxes" which limits their effectiveness when dealing with complex scenarios like merge conflicts or repository corruption.

One of the main reasons for choosing this topic is my personal experience with the difficulties of keeping up with code changes while developing my high school graduation thesis. I was then developing a game in Delphi, and as a beginner programmer, I soon found myself having a hard time keeping track with the various versions of my code. Without a clear system, my solution was to create duplicate copies of the project folder like final_v1, final_v2, final_v3 manually to prevent loss of progress when trying out new features in the game.

Although it seemed like a good solution at first, it caused me to copy the wrong version of the project to a flesh drive for my project defense. This is a situation where a proper version control system would have prevented the mistake.

To address those problems, this thesis presents learn-by-doing approach to understanding and demystifying version control systems by implementing such as system from scratch, entirely within the Java Virtual Machine (JVM) environment using the Kotlin programming language. This implementation is inspired by Git, the most widely adopted and popular version control tool, which, despite being so popular, lacks appropriate documentation explaining how it works under the hood. Since Git is open-source project, its functionality and design are commonly taken for granted as being self-explanatory, but in fact, it is so complex that it is hard for most developers particularly beginners to understand its internal processes.

Beyond its educational purpose, this thesis also seeks to examine whether JVM-based implementation can be used for development of a version control system which has traditionally been developed in low-level languages that offer more memory control.

Primary goals of this thesis are to demystify the internal workings of distributed version control systems through a practical development. This include analyzing the Git's metadata database and object model that enable version tracking, understanding how cryptographic hashing and compression work together to ensure efficiency and integrity, and examining the mechanisms that enable distributed collaboration between repositories. By creating an operational implementation replicating Git's fundamental functionality, this thesis provides developers immediate insight into the design decision and algorithms that make distributed version control possible, filling the gap in documentation provided in current resources.

This thesis combines experimental and comparative research methodologies, supported by reverse engineering techniques. The experimental methodology tests implementation of a Git-like version control system in Kotlin within JVM environment under controlled conditions to evaluate its viability in an environment which is not traditionally used for these systems.

Comparative analysis is used as an analytical technique to evaluate differences between JVM-based implementation and Git's native C implementation. This comparison focuses on key metrics such as:

- Differences in hash computation, object compression
- Storage efficiency and repository size
- Architectural adaptations required to implement Git-like functionality within the JVM ecosystem.
- Runtime efficiency vs development simplicity trade-offs

To support the experimental methodology, reverse engineering techniques are employed to examine Git's internal structure and algorithms, this systematic deconstruction provides foundational understanding that are required for accurate replication of Git's basic functionality.

By combining these approaches through the JVM-based implementation, this thesis contributes to the understanding of both distributed version control systems as well as the ability of the JVM platform for system-level programming.

This thesis is divided into several key chapters that deal sequentially with all areas of research:

Chapter 1 introduces version control systems, how they work, and how they evolved from local to distributed. It has a special subsection on distributed version control architecture in which terms such as working directory, staging area, local repository, and remote repository are explained.

Chapter 2 is most important part of this work, which deals with theory and implementation aspects of object storage in distributed version control systems. This chapter elaborates thoroughly on how blob, tree, and commit objects are created, the relationship between objects, and operations like hashing, compression, branching, merging and remote communication. It includes dedicated subsection on how the ZLIB compression library uses DEFLATE algorithm.

Chapter 3 includes a comparative analysis of Git and our implementation, evaluating performance metrics, architectural differences, and the ability of JVM-based implementations for such systems.

1. Version Control System

Version control also known as source control or revision control is an important software development practice for tracking and managing changes made to code and other files. It is closely related to source code management.

With version control, every change made to the codebase is tracked. This allows software developers to see the entire history of who changed what at any given time and rollback from the current version to an earlier version if needed. It also creates a single source of truth [1]

Version control is particularly valuable beyond just software development. For example, if you are graphic or web designer wanting to keep every version of an image or layout, a Version Control System (VCS) is essential. It provides several key benefits such as ability to revert files back to a previous state, ability to revert the entire project back to a previous state, ability to compare changes over time, ability to see who last modified something that might be causing a problem, ability to identify who introduced an issue and when and easy recovery if mistakes are made or files are lost [2]

Most people have already used some form of version control, without realizing it, by saving different versions of their work in separate folders. While this manual approach is common due to its simplicity, it is incredibly error prone. It is easy to forget which directory contains the current version or accidentally overwrite important files.

To deal with these issues, different types of version control systems were developed over time. These systems can be categorized into three main types:

1. Local Version Control Systems (LVCS)
2. Centralized Version Control Systems (CVCS)
3. Distributed Version Control Systems (DVCS)

Each type represents an evolution in how developers manage and track changes to their code, with each new type addressing limitations of its predecessors.

1.1. Local Version Control Systems

To deal with the issue of manual file copying, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control. This represented the first systematic approach to version control. One of the more popular VCS tools was a system called RCS (Revision Control System), which is still distributed with many computers today. RCS works by keeping patch sets (that is, the difference between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches. [1]

As shown in Figure 1 a LVCS manages files on a single local computer. The systems maintain a version database that stores snapshots of file changes over time (Version 1, Version 2, Version 3), creating a chain of versions that tracks how file changed over time. When developer wants to see a particular version of a file RCS would reconstruct it by applying the stored changes.

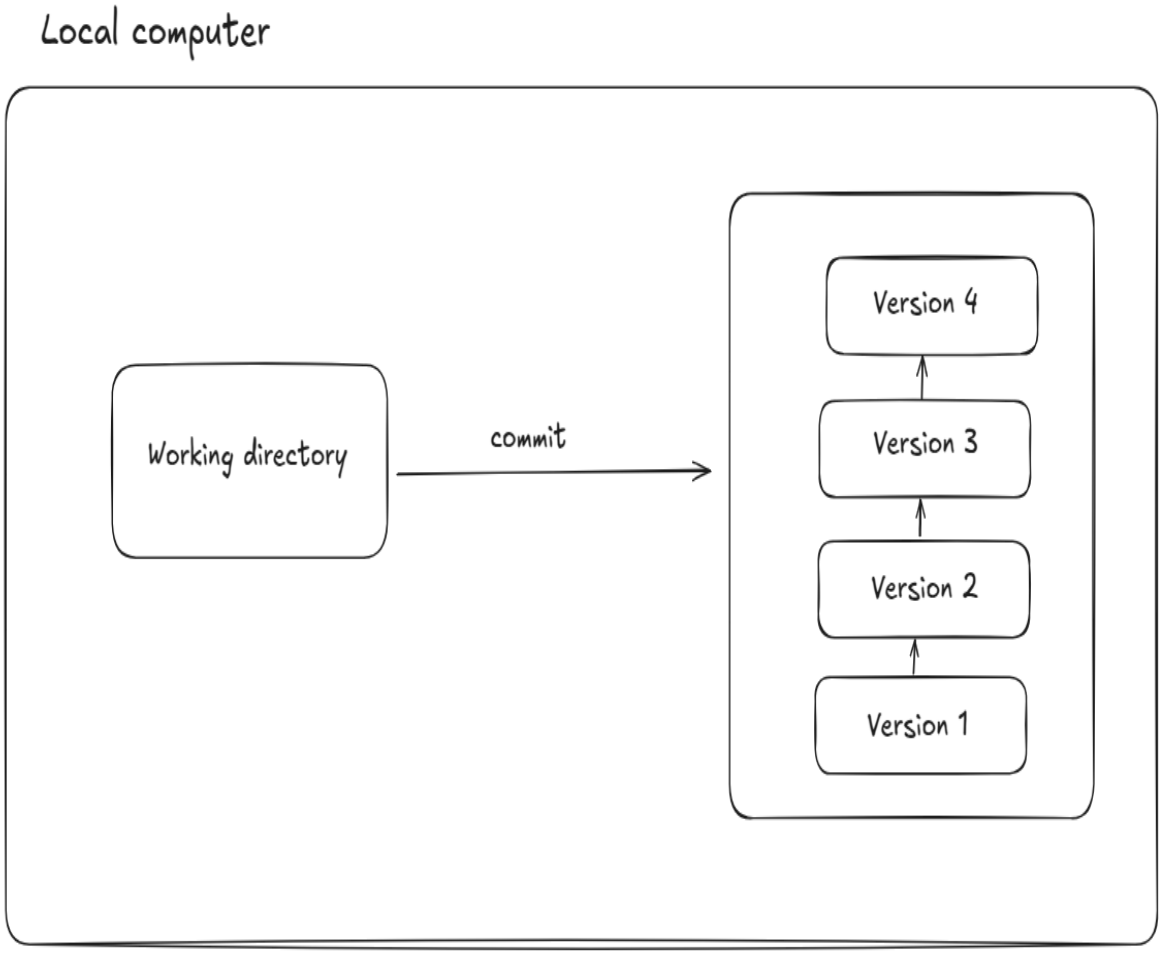

Figure 1 Local version control [3]

However, this approach had limitations. Since everything was stored locally on a single computer, there was no way for developers to collaborate with others working on different machines. If something happened to the computer where the RCS database was stored, all project history would be lost unless there were backups.

1.2. Centralized Version Control Systems

The next major issue that developers encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCS) were developed. These systems introduced a new model where all versioned files are stored on a single central server, with developers checking out files from this centralized location. [2]

In Figure 2 we can see that the centralized version control system architecture consists of a central server that maintains a version database containing the complete history of all file changes. The diagram shows how multiple computers (Computer A and Computer B) can connect to this central server to access and modify files. While developers work with local copies of the files, all version history and change tracking is managed through the central server. Developers must first pull the latest version of code from the server to their local machine. After making changes, the push their commits back to the central repository, where other team members can then access them.

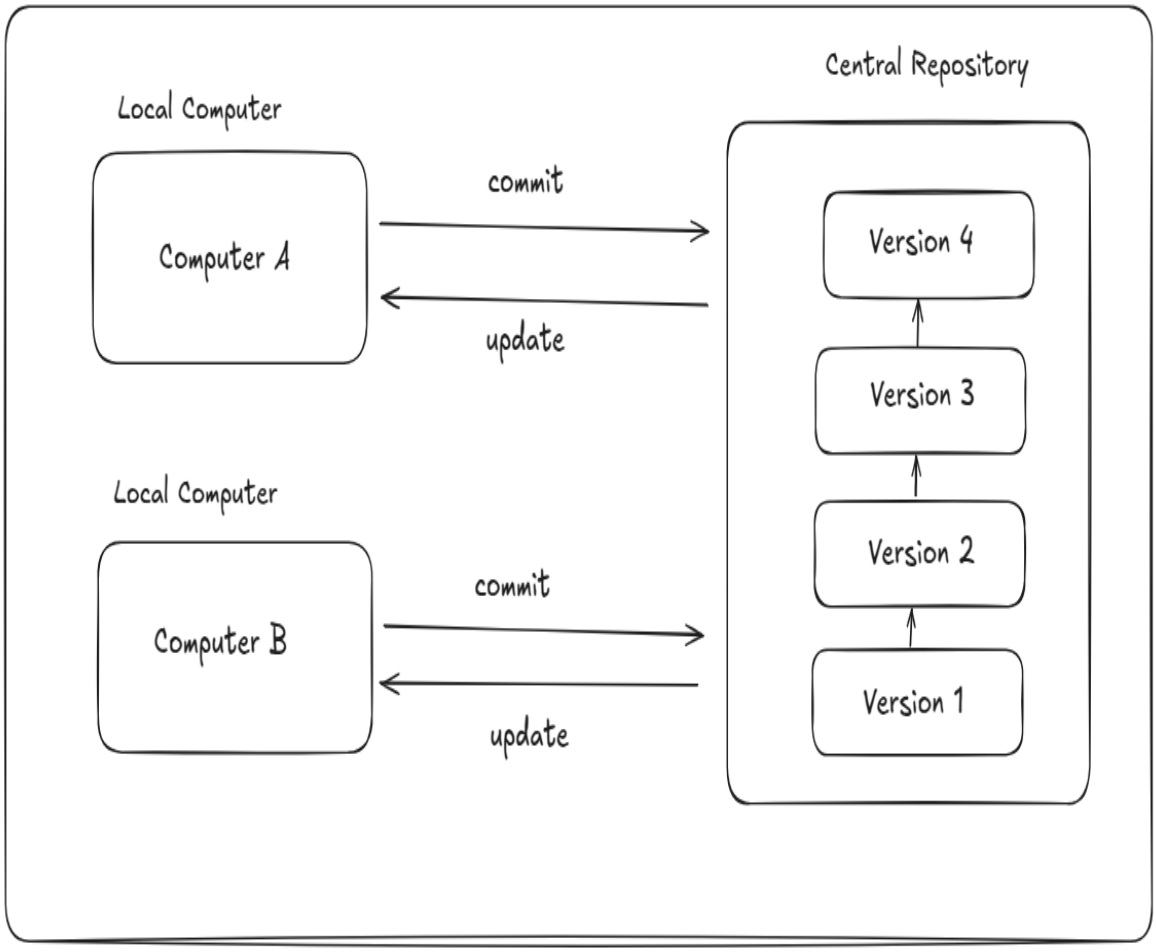

Figure 2 Centralized version control system [3]

With centralized approach developers have clear visibility into everyone's activities and contributions to the project, while administrators can maintain control over access permissions and user rights. Centralized version control systems significantly simplify system administration compared to managing local databases on each developer machine, making centralized approach much better solution for team-based development. The centralized architecture enables a standardized workflow where developers can see each commit, track progress, and understand the project's evolution through a single primary repository. [1]

This system has many advantages and is easy to use, most obvious disadvantage is the single point of failure that the centralized server represents. If the server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they are working on. If the hard disk of the centralized server becomes corrupted, and proper backups have not been kept, you lose everything, the entire history of the project expect whatever single snapshots developers happen to have on their local machines.

1.3. Distributed Version Control Systems

Distributed Version Control Systems (DVCS) represent the next evolution in version control systems. Distributed architecturally refers to a system design where collection of computer programs spread across multiple computational nodes. Each node is a separate physical device or software process but works towards a shared objective. Distributed architectures vary in the way they organize nodes, share computational tasks, or handle communication between various parts of the system. Each component operates independently, allowing for parallel processing and reducing bottlenecks that can occur in centralized systems. This independence means that developers can work on various parts of a project simultaneously without waiting for a central server to process their requests. [4].

These systems introduced a fundamentally different approach where clients do not just check out the latest snapshot of files, they fully mirror the entire repository. [2]. This change approach was so significant that in 2010, software development author Joel Spolsky described Distributed Version Control Systems as "possibly the biggest advance in software development technology in the past ten years" [5].

In Figure 3, we can see the key difference in distributed version control systems architecture compared to previous systems. Each computer maintains its own complete version database, containing the full history of all changes. This means that in addition to the server having a version database Computer A and Computer B also have their own complete copies of the entire repository.

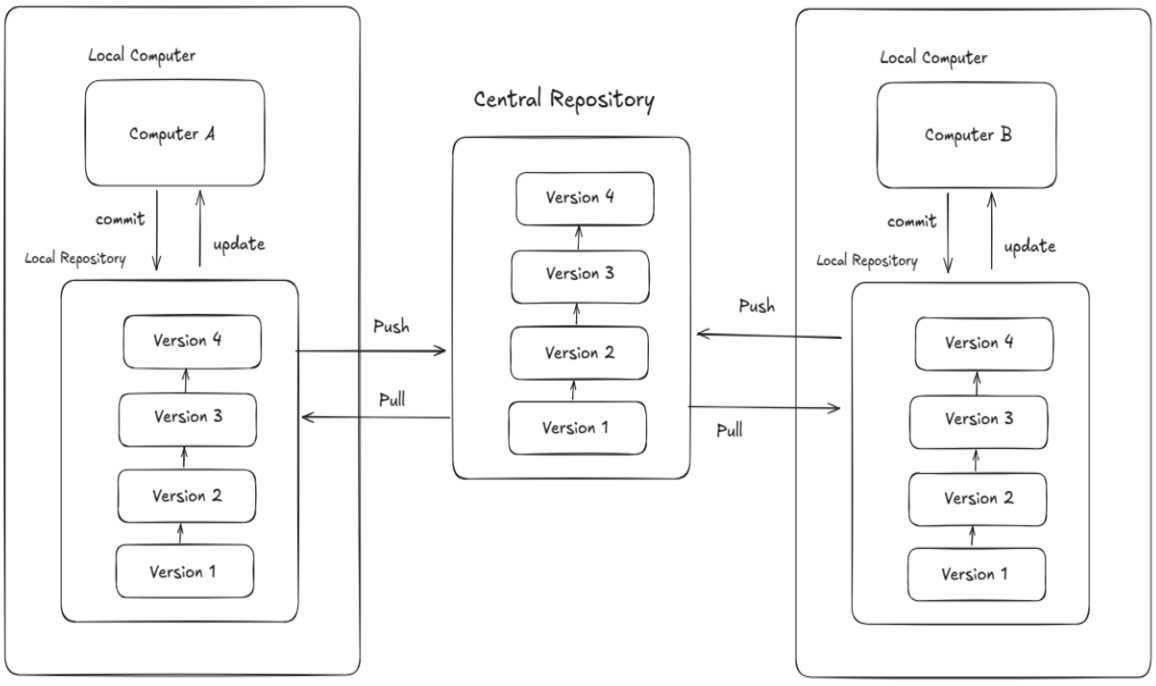

Figure 3 Distributed Version Control System [3]

The figure 3 illustrates how developers on Computer A and Computer B can work independently on their local copies, making commits without requiring constant access to a central server. To share changes with others, they use push and pull commands. Pull command allows developer to fetch updates from a central repository into their local copy. Push operation sends locally committed changes to a central repository making them available to others.

While following Peer-to-peer (P2P) architecture, where each node can function as both client and server. Most DVCS implementations like Git use a hybrid model with a central repository serving as a coordination of point but is not technically required for the system to function. Its primary acts as the “source of truth” for the official project state because each developer have own local repository, developers can work offline and commit changes. Common operations such as commit, viewing history, and reverting changes are also significantly faster because there is no need to communicate with a central server, communication is only necessary when sharing changes with other peers. This flexibility has made DVCS the standard choice for modern software development with Git being most widely used example.

Git, created by Linus Torvalds in 2005, represents one of the most powerful and flexible DVCS. It was originally developed to support Linux kernel development but has since become an essential tool for a wide range of projects. Git was designed with several key principles in mind, including enabling distributed development, allowing parallel and independent contributions, and ensuring that developers could work in private repositories without needing constant synchronization with a central server. [6]

Regarding the name "Git," Linus Torvalds, known for his humor and candid personality, gave multiple explanations. In his own words:

> *"I'm an egotistical bastard, and I name all my projects after myself. First Linux, now Git."*

He also jokingly referred to Git as a British slang term for "a stupid or unpleasant person," implying that the name was self-deprecating. However, the name later took on more technical interpretations, with some considering it an acronym for "Global Information Tracker." [6]

Git's impact on modern software development is undeniable. What began as a necessity for managing the Linux kernel has evolved into the de facto version control system used by major companies, open-source communities, and individual developers worldwide.

1.3.1 Distributed Version Control Systems Components

This hybrid architecture works through four main components that manage changes to source code, DVCS tracks not just the files themselves, but also their states. Files can exist in three main states: committed, modified, and staged. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot. [2]

Figure 4 demonstrates how states of file are managed through different components and their mutual connection.

1. Working Directory – where files are modified
2. Staging Area/Index – where changes are prepared for committing
3. Local Repository – where committed changes are stored
4. Remote Repository – where changes are shared between developers

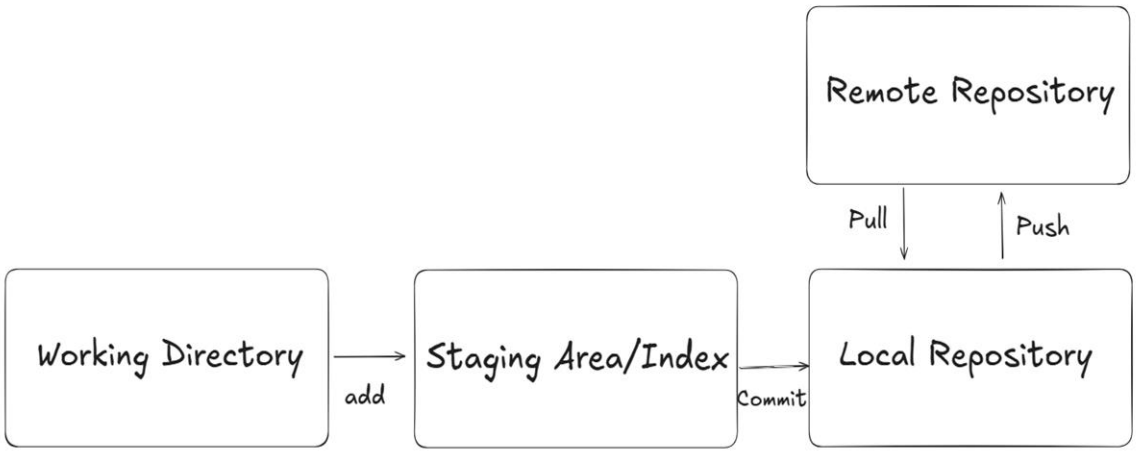

Figure 4 Architecture of Distributed Version Control System

These components are managed through a Command Line Interface (CLI) application that interacts with a special directory (in our implementation. zsv) that contains all the version control information. The CLI application writes and reads metadata from this directory, which contains all the version control information. Simply explained CLI application transforms files and directories into compressed metadata stored in the. zsv directory. When tracking changes, the CLI compresses file contents and stores them along with their metadata, and when retrieving files, it decompresses this data back into the working directory. This separation of version control data from project files prevents conflicts between source code and version control metadata.

The actual project files that are being edited/modified are in working directory where developers amend them. These changes are not tracked by version control system until they are added to the staging area. This staging area acts as intermediate area between working directory and local repository. The staging area is important because it enables developers to select exactly which changes should be included in commit.

Once changes are staged, they can be committed to the local repository, which is a complete version of project with entire history of modifications. Each developer maintains their own local repository, enabling independent work without requiring constant connection to central repository.

The remote repository functions as a shared version of project, it serves as a central hub where developers can push their local commits and pull updates made by others. While the remote repository is centralized, it is important to note that it is just another complete copy of the repository, like local repositories. Difference between local repository and remote repository is that remote is hosted in a location which is accessible to all developers in team. Even if every developer has complete copy of the repository, the remote version typically represents the agreed-upon official state of the project.

2. Architecture and Implementation of Object Storage in DVCS

In the previous section, we explained the architecture of DVCS and how its components interact, but not lifecycle of file and how is stored and transferred to object. This chapter demonstrates both theoretical aspects and implementation of object storage in DVCS.

There are few steps that need to be done in process of transforming file to DVCS object:

1. File compression (DEFLATE algorithm)
2. Generating unique identifier (SHA-1)
3. Store file in index:
    3.1. Store SHA-1 in index
    3.2. Put entry in index
4. Store objects (blob, tree)
5. Parse index
6. Create commit object (hierarchy)

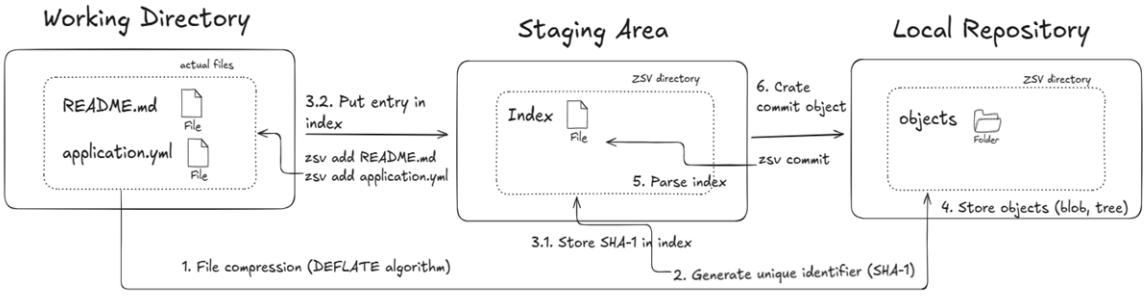

Figure 5 Lifecycle of file in Distributed Version Control Systems

2.1. Local Repository Setup

To better understand process of transforming file to object we need to look at the. zsv directory structure. The dot before the name zsv indicates that this is a hidden directory in Unix-like systems, this is done because we want to keep the version control metadata separate and hidden from regular project files.

In this section we will explain how zsv init command crates the necessary directory structure and metadata files that form the basis of the entire system.

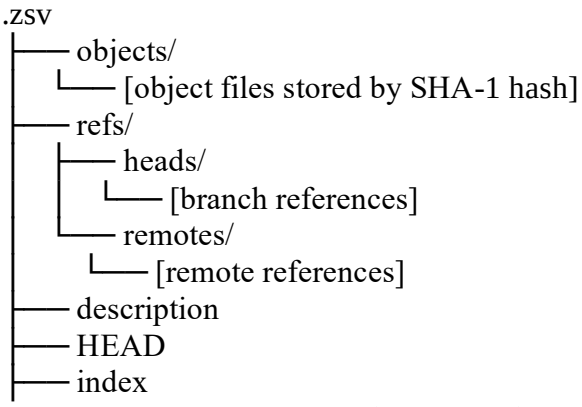

Figure 6 Zsv Directory Structure

Implementation of `zsv init` command is straightforward we need to create necessary directory structure and initialize a few configuration files that can be seen in Figure 6.

```
@Command(command = ["init"], description = "Initialize empty $ZSV_DIR repository")
fun initRepository(): String

   if (Files.exists(Paths.get(ZSV_DIR))) {
      return "Zsv repository already exist"
   }

   val zsvPath = createZsvStructure()
   return "Initialized empty zsv repository in $zsvPath"
}

fun createZsvStructure(path: Path = getCurrentPath()): Path {
   val zsvPath = path.resolve(ZSV_DIR)

   listOf("objects", "refs/heads", "refs/remotes").forEach {
      Files.createDirectories(zsvPath.resolve(it))
   }
   Files.writeString(zsvPath.resolve("HEAD"), "ref: refs/heads/master\n", UTF_8)
   Files.writeString(
      zsvPath.resolve("description"),
      "Unnamed repository; edit this file 'description' to name the repository.\n", UTF_8
   )
   return zsvPath
}
```

The `initRepository()` function first checks if a zsv directory already exist in the current location. This check prevents accidentally initializing a repository where one already exists, ensuring idempotency of the operation. If no repository exists, the function calls `createZsvStructure()` which creates. zsv directory at the current path and creates several subdirectories:

- objects/ where all content (blobs, trees, commits) will be stored.
- refs/heads/ - stores references to branch heads
- refs/tags/ - stores references to tags

In additional to creating the directory structure, the function also initializes two important files

- HEAD - points to the default branch (master)
- description – contains a default message that can be edited by user.

Once the structure is successfully created, the command returns a confirmation.

2.2. Content-Addressable Storage System

In this section we will explain the implementation of the object storage system that uses content-addressable-storage principle where files and directories are stored as objects identified by their SHA-1 hash value. This include creation and storage of blob, tree and commit objects and everything needed in that process such as technical mechanisms for transforming file contents into repository objects, the process of generating SHA-1 hashes, and how is compression applied to optimize storage space.

Content-addressable-storage is principle which means that content is stored and retrieved based on its content rather than its location or name. [6] This principle is possible because when creating an object identifier, DVCS only considers the file's content not its name or location. These identifiers function like fingerprints for each object, for example text file which contains hello world will always produce same "2aae6c" SHA-1 hash. This behavior is known as deterministic, meaning same content must always produce same identifier. This is why we use Secure Hash Algorithm 1 (SHA-1), as it is a cryptographic hash function that generates a unique identifier (hash). SHA-1 produces a 160-bit (20-byte) hash value, this hash value is known as a message digest, typically rendered as forty hexadecimal digits. [7]

2.2.1. Blob Objects

Each version of a file is represented as a blob, a contraction of "binary large object" is a term that's commonly used in computing to refer to some variable or file that can contain any data and whose internal structure is ignored by the program. A blob is treated as being opaque. A blob holds a file's data bud does not contain any metadata about the file or even its name [2]

The first step in implementing our storage system is to define the format for our blob objects. Our blob structure follows a specific format: blob {size}\u0000{content}. This format begins with the object type identifier (blob), followed by the size of the content, a null byte separator (\u0000) and the actual file content. This format is chosen because blob prefix allows us to distinguish blobs from other object types (tree, commits), size filed ensures we can verify whether a file is complete or corrupted by comparing its stored size with actual size and null byte provides us separator between metadata and content for easier parsing. Below is the implementation of our Blob data class.

```
/**
 * Blob object structure:
 * blob {size}\u0000{content}
 **/
data class Blob(
    val content: ByteArray,
    val sha: String
) {
    override fun toString(): String {
        return content.toString(StandardCharsets.UTF_8).substringAfter("\u0000")
    }

    fun getContentWithoutHeader(): ByteArray {
        val nullByte = content.indexOf(0)
        return content.slice(nullByte + 1 until content.size).toByteArray()
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as Blob
        return this.sha == other.sha
    }

   override fun hashCode(): Int {
       return sha.hashCode()
   }
}
```

The Blob class contains two primary fields: content (the raw byte array including the header) and sha (the SHA-1 hash that identifies this blob). Importantly, the equality of two blob objects is determined by comparing their SHA-1 hashes rather than their content. This aligns with the content-addressable-storage principle, meaning that content itself determines the object's unique identifier. With SHA-1 we also have benefits of efficient storage, because if two files have the same hash, they must also have the identical content. This allows us to store only one copy in zsv/objects regardless of how many times that identical content appears in different directories or at different points in the project's history. This deduplication is a consequence of content-addressable storage and significantly reduces the storage requirement for repositories with repeated content.

Now that we have defined our blob model class, we need to create a command `hash-object` that accepts a file and performs the compression.

```
@Command(command = ["hash-object"], description = "Create blob object")
fun compressFileToBlobObject(
    @Option(shortNames = ['f'], required = true, description = "Path to the file to compress")
fileToCompress: String
): String {
    val path = Paths.get(fileToCompress)
    return blobService.compressFromFile(path)
}
```

The command has only one option, -f which indicates the path to the file that should be compressed and stored. Actual compression and storage logic is implemented in `blobService.compressFromFile()`

```
fun compressFromFile(path: Path): String {
    val fileContent = Files.readAllBytes(path)
    val blobHeader = "blob ${fileContent.size}\u0000".toByteArray(Charsets.UTF_8)
    val content = blobHeader + fileContent
    val compressedContent = content.zlibCompress()

    val blobSHA1 = content.toSha1()

    val blob = Blob(
        content = compressedContent,
        sha = blobSHA1
    )
    val currentDirectory = getCurrentPath()
    val objectsDirectory = currentDirectory.resolve(OBJECTS_DIR)
    storeObject(objectsDirectory, blob.sha, blob.content)

    return blob.sha
}
```

This method reads the content of the file into a byte array and create header for the blob that includes object type and content size (blob {size}\u0000). The header and file content are combined into a single byte array, which is then compressed using the ZLIB compression algorithm.

```
fun ByteArray.zlibCompress(): ByteArray {
    val output = ByteArray(this.size * 2)
    val compressor = Deflater().apply {
        setInput(this@zlibCompress)
        finish()
    }
    val compressedDataLength: Int = compressor.deflate(output)
    return output.copyOfRange(0, compressedDataLength)
}
```

Fortunately, we don't have to implement the ZLIB compression algorithm from scratch. The JVM provides the Deflater class that implements this algorithm. This extension function takes byte array and returns its compressed version. It creates an output buffer with twice the size of the input (to ensure there is enough space even in worst-case scenarios). After compression, it trims the output array to the actual size of the compressed data. This
algorithm is explained in detail, along with examples in compression section (2.4.)

The SHA-1 hash is generated from the uncompressed content (header + file content) using simple extension function. This function uses JVM Message Digest class, which provides implementations of standard cryptographic hash functions. We specify "SHA-1" as the algorithm, pass our byte array to the digest method, and then format the resulting bytes as a hexadecimal string. Each byte is converted to a two-character hexadecimal representation ("%02x") and these characters are jointed together without separators to form the complete 40-character SHA-1 hash. SHA-1 is used instead of SHA-256 due to performance, while SHA-1 have provided stronger security, SHA-1 is much faster for hashing enormous amounts of data. Many existing CAS systems like Git still use SHA-1, making it a natural choice.

```
fun ByteArray.toSha1(): String {
    return MessageDigest
        .getInstance("SHA-1")
        .digest(this)
        .joinToString(separator = "", transform = { "%02x".format(it) })
}
```

In the end we store compressed objects in repository object storage system using a subdivided directory structure. Reason we subdivide object storage to SHA-prefix folders is to improve file system efficiency, we split the 40-character hash: the first two characters are used as a directory name, and the remaining 38 characters become the filename. Although it is possible to put all the files in one directory, many systems have a limit. For example, File Allocation Table (FAT) - FAT32 can only store 65,535 files in a single directory, while even small repository might contain over 180,000 objects. By splitting the objects based on the first two hex characters we ensure that no single directory becomes too large, which maintains reliable performance across different file systems and operating systems. [8]

```
fun storeObject(directory: Path, objectSha: String, compressedContent: ByteArray) {
    val subDirectory = directory.resolve(objectSha.substring(0, 2))
    Files.createDirectories(subDirectory)
    val blobFile = subDirectory.resolve(objectSha.substring(2))
    Files.write(blobFile, compressedContent)
}
```

This function takes first two characters of the hash and crate a subdirectory name, the renaming 38 characters becomes the filename. For example, a blob with SHA-1 hash "a1b2c3d4e5f6" would be stored at path zsv/objects/a1/b2c3d4e5f6.

After compressing blob objects, we need to be able to decompress them as well. This functionally is implemented through the `cat-file` command.

```
// zsv cat-file -f a3c241ab148df99bc5924738958c3aaad76a322b
@Command(command = ["cat-file"], description = "Read blob object")
fun decompressBlobObject(
    @Option(shortNames = ['f'], required = true, description = "Path to the file to decompress") sha: String
): String {
    val blob = blobService.decompress(sha, getCurrentPath()).toString()
    return blob.substringAfter('\u0000')    // remove header (blob content.length)
}
```

This command takes a SHA-1 hash as input and returns the decompressed content of the blob object. Decompression logic is implemented in the `blobService.decompress()` method. This method verifies that the SHA-1 hash is valid (40 characters longs), locates the object file, reads its compressed content, and decompresses it.

```
fun decompress(sha: String, basePath: Path = getCurrentPath()): Blob {
    if (sha.length != 40) {
        throw InvalidHashException("Invalid blob hash. It must be exactly 40 characters long.")
    }

    val path = getObjectShaPath(basePath, sha)
    val compressedContent = Files.readAllBytes(path)
```

```
    val decompressedContent = compressedContent.zlibDecompress()

    val header = decompressedContent.take(4).toByteArray().toString(Charsets.UTF_8)

    if (header != "blob") {
        throw InvalidObjectHeaderException("Not a blob object")
    }

    return Blob(
        content = decompressedContent,
        sha = sha
    )
}
```

Decompression function uses JVM Inflater class to decompress data, it reads the decompressed data in chunks (using a 1024-byte buffer) and iteratively calling `inflate()` method which decompresses chunk by chunk until no more data can be decompressed, releasing resources with `inflater.end()` and returning the decompressed data.

```
fun ByteArray.zlibDecompress(): ByteArray {
    val inflater = Inflater()
    val outputStream = ByteArrayOutputStream()

    return outputStream.use {
        val buffer = ByteArray(1024)

        inflater.setInput(this)

        var count = -1
        while (count != 0) {
            count = inflater.inflate(buffer)
            outputStream.write(buffer, 0, count)
        }

        inflater.end()
        outputStream.toByteArray()
    }
}
```

2.2.2. Tree Objects

Tree object represents one level of directory information. It records blob identifiers, path names, and a bit of metadata for the files in one directory. It can also recursively reference other (sub)tree objects and thus build a complete hierarchy of files and subdirectories [6]. Tree object solves the problem of storing only content in blob and allows you to store a group of files together. DVCS stores content in a manner like a UNIX file system but simplified. Trees correspond to UNIX directory entries while blobs correspond to inodes or file contents. A single tree object contains one or more entries, each with a SHA-1 pointer to a blob or subtree with its associated mode, type and filename. [2]

The tree object structure follows a specific format:

```
tree {size}\u0000{content}
{mode} {name}\u0000{sha1}\n  (repeated for each entry)
```

```
data class Tree(
    val fileMode: String,
    val fileName: String,
    val sha: String
) {
    override fun toString(): String {
        return String.format(
            "%-6s %-4s %-40s %s",
            fileMode,
            getObjectType(),
            sha,
            fileName
        )
    }
    private val objectType: ObjectType
        get() = ObjectType.fromMode(fileMode)

    private fun getObjectType(): String {
        return when (objectType) {
            REGULAR_FILE, EXECUTABLE_FILE, SYMBOLIC_LINK -> "blob"
            DIRECTORY -> "tree"
        }
    }
}
```

Each tree entry has file mode which is string and indicate the type and permission of the file or directory, file name which is name of the file or directory and sha which is object identifier. The process of creating and storing tree objects is implemented in zsv `write-tree` command. This command recursively builds a tree structure from a directory. For each file in the directory, if it is a subdirectory, the method calls itself recursively to create a tree object for that subdirectory, if it's a file it creates a blob object using the blob service. Important, this command only works with current path, it is not possible to give specific directory, and it is possible to ignore some files if you want to.

```
@Command(command = ["write-tree"], description = "Create tree object")
fun compressTreeObject(): String {
    return treeService.compressFromFile()
}
fun compressFromFile(path: Path = getCurrentPath()): String {
    val ignoredItems = setOf( ZSV_DIR, ".git", ".gradle", ".idea", "build")
    val objects = mutableListOf<Tree>()

    Files.list(path).use { stream ->
        stream
            .filter { file ->
                !ignoredItems.contains(file.fileName.toString())
            }
            .forEach { file ->
                val name = file.fileName.toString()
                if (Files.isDirectory(file)) {
                    val sha = compressFromFile(file)
                    objects.add(Tree(DIRECTORY.mode, name, sha))
                } else {
                    val blobSha = blobService.compressFromFile(file)
                    val fileMode = when {
                        Files.isExecutable(file) -> EXECUTABLE_FILE
                        Files.isSymbolicLink(file) -> SYMBOLIC_LINK
                        else -> REGULAR_FILE
                    }
                    objects.add(Tree(fileMode.mode, name, blobSha))
                }
            }
    }
    return storeTree(objects)
}
```

```
private fun storeTree(objects: List<Tree>): String {
    val sortedObjects = objects.sortedBy { it.fileName }
    val treeContent = buildTreeContent(sortedObjects)
    val treeHeader = "tree ${treeContent.size}\u0000".toByteArray()
    val content = treeHeader + treeContent

    val compressedContent = content.zlibCompress()
    val sha = content.toSha1()

    val currentDirectory = Paths.get("").toAbsolutePath()
    val objectsDirectory = currentDirectory.resolve(OBJECTS_DIR)
    storeObject(objectsDirectory, sha, compressedContent)
    return sha
   }
```

```
private fun buildTreeContent(trees: List<Tree>): ByteArray {
    val outputStream = ByteArrayOutputStream()
    trees.forEach { tree ->
        val modeAndName = "${tree.fileMode} ${tree.fileName}\u0000".toByteArray()
        val sha = tree.sha.toShaByte()
        outputStream.write(modeAndName)
        outputStream.write(sha)
    }
    return outputStream.toByteArray()
}
```

When the `write-tree` command is executed on a project directory with following structure:

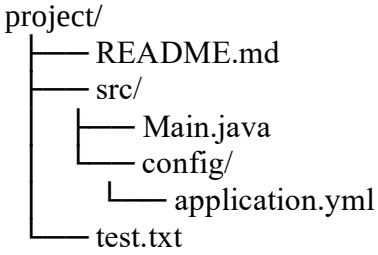

Figure 7 Project directory structure example

First, it lists all entries in the project directory: README.md, src/ and test.txt. For each file it creates a blob object by calling blob service compress method and adds a Tree entry with file mode, name, and blob's SHA-1.

For the src/ directory, it recursively calls `compressFromFile(src/)`. Inside this recursive call it lists entries again: Main.java and config/. For Main.java, it creates a blob object and adds a Tree entry with the file mode, name, and blob's SHA-1.

For the config/ directory, it again recursively calls `compressFromFile(config/)`. Inside this recursive call for config/, it lists the entry application.yml, creates a blob object for it and adds a tree entry with its file mode, name and SHA-1. It then calls `storeTree()` to create a tree object for the config/ directory, which binary format would look like this:

```
tree {size}\0100644 application.yml\0[20-byte SHA-1 hash]
```

Back in the src/ directory, it now has entries for both Main.java (blob) and config (tree, represented by its SHA-1). The tree object binary format would look like this:

```
tree {size}\0100644 Main.java\0[20-byte SHA-1 hash]040000 config\0[20-byte SHA-1 hash]
```

Finally, we are back in the root directory processing, it calls `storeTree()` with entries for README.md (blob), src/ (tree) and test.txt (blob). The final root tree object would be:

```
tree {size}\0100644 README.md\0[20-byte SHA-1 hash]040000 src\0[20-byte SHA-1
hash]100644 test.txt\0[20-byte SHA-1 hash]
```

This crates the root tree object and returns its SHA-1, which uniquely identifiers the entire directory stricture at that point in time. This SHA-1 is particularly important as its servers as a reference to this specific state of the repository. When creating a commit, this SHA-1 will be used to link the commit to the exact directory structure it represents.

After compression, we must be able to read and retrieve trees. That is why we developed command `zsv ls-tree` which accepts `-f` flag indicating tree object identifier. This decompression works by finding the object file, then the `parseTreeContent()` function.

```
@Command(command = ["ls-tree"], description = "Read tree object")
fun decompressTreeObject(
   @Option(shortNames = ['f'], required = true, description = "Path to directory you want to
decompress") sha: String
): String {
   val trees = treeService.decompress(nameOnly, sha)
   return trees.joinToString("\n") { it.fileName }
}
fun decompress(nameOnly: Boolean, sha: String): List<Tree> {

   val path = getObjectShaPath(getCurrentPath(), sha)
   val compressedContent = Files.readAllBytes(path)
   val decompressedContent = compressedContent.zlibDecompress()

   val header = decompressedContent.take(4).toByteArray().toString(Charsets.UTF_8)

   return parseTreeContent(decompressedContent)
}

fun parseTreeContent(content: ByteArray, skipHeader: Boolean = true): List<Tree> {
   var index = if (skipHeader) content.indexOf(0) + 1 else 0
   val result = mutableListOf<Tree>()

   while (index < content.size) {
      val modeAndName = StringBuilder()

      while (index < content.size && content[index] != 0.toByte()) {
         modeAndName.append(content[index].toInt().toChar())
         index++
      }

      index++

      if (index + 20 <= content.size) {
         val objectSHA = content.slice(index until index + 20).toByteArray().toHexString()
         index += 20

         val (mode, name) = modeAndName.toString().split(" ", limit = 2)
         result.add(Tree(mode, name, objectSHA))
      } else {
         break
      }
   }
   return result
}
```

When executing `zsv ls-tree -f 54c4a4a6…` (assuming this is the SHA-1 of our root tree example from tree compression), `decompress()` function locates the object file using this SHA-1, reads and decompresses the file content. It then passes the decompressed content to `parseTreeContent()`.

The `parseTreeContent()` function skips past the header (everything before and including the first null byte) and starts reading bytes from this position. For each entry, it reads characters until its hits a null byte, build the “mode name” string, reads the next twenty bytes as the SHA-1 hash, splits the “mode name” into separate mode and name fields and creates a tree object with those values. For our example, it would process.

```
tree {size}\0100644 README.md\0[20-byte SHA-1 hash]040000 src\0[20-byte SHA-1 hash]100644 test.txt\0[20-byte SHA-1 hash]
```

The function would first extract “100644 README.md”, then its 20-byte SHA-1, creating a tree object for README.md. Next it would extract “040000 src” and its SHA-1, creating a tree object for the src directory. Finally, it would extract “100644 test.txt” and its SHA-1 creating a tree object for test.txt. The command would display output like this:

```
100644 blob  5f4dc74af35ad52e1891e80adb5ab5f573622aee README.md
040000 tree  a45791e635ebe8f9b2789f21b97c1afff256c1b5 src
100644 blob  da39a3ee5e6b4b0d3255bfef95601890afd80709 test.txt
```

The tree implementation approach detailed above is based on several computer science principles. Design choice to use hierarchical tree structures comes from fundamental principles in data organization theory. The content-addressable approach to tree storage creates what is known as Merkle tree. A Merkle tree is a hash-based data structure where each non-leaf node is labeled with cryptographic hash of the labels of its child nodes. In our implementation, this means that tree’s SHA-1 identifier depends not only on its immediate contents but also indirectly on all contents beneath it in the hierarch [9]

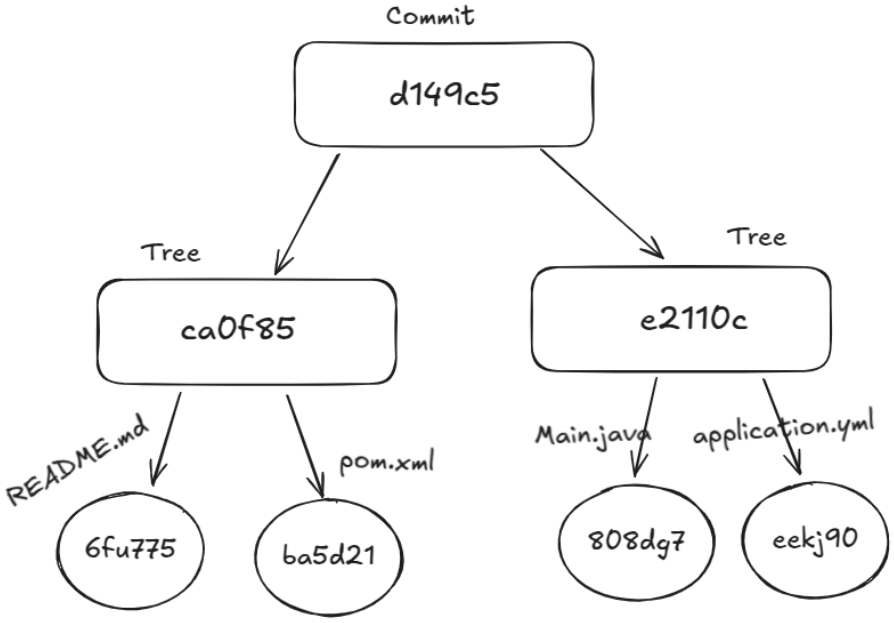

Figure 8 Merkle Tree

As shown in the figure above, this structure creates a relationship between objects. The commit object (d149c5) represents a snapshot of the repository at specific point in time. It points to tree objects (caof85 and e2110c) that represent the directory structure. Each tree object points to blob objects that store the actual file contents (6fu775 for README.md and ba5d21 for pom.xml or 808dg7 for Main.java and eekj90 for application.yml). When a commit points to a tree it indicates that “this is the state of the project at this point in time”. When a tree points to blob or other trees it indicates that “these are the files and subdirectories in this directory.” This Merkle tree principle is central to our tree implementation, providing integrity verification, efficient comparison, and deduplication benefits while maintaining a clean mapping to the filesystem structure.

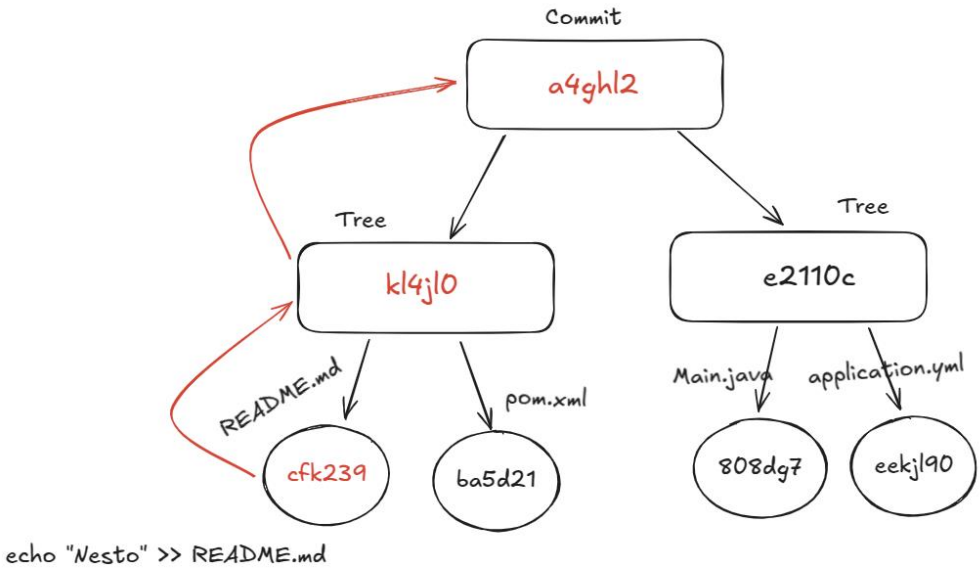

Figure 8 SHA hash propagation in Merkle tree after file modification

Figure 8 illustrates how changes propagate through the Merkle tree of a repository. When content is modified (as shown by the command `echo "Nesto" >> README.md`), a new blob object with hash cfk239 is created for the updated file. Because of this, tree object above is affected and since tree objects store hashes of their children a new tree object kl4j10 must be created. This cascading effect continues up the tree, creating a new commit a4ghl2. The red arrow demonstrates how changes cascade upward through the hierarch. If the content of any file were to change, all parent objects that reference it (directly or indirectly) would change as well.

2.2.3. Commit Objects

Commit object holds metadata for each change introduced into the repository, including the author, committer, commit data, and log message. Each commit points to a tree object that captures, in one complete snapshot, the state of the repository at the time the commit was performed. The initial commit, or root commit, has no parent. Most commits have one commit parent. Although with trees we created literally a snapshot of our project we have problem that remains we must remember all three SHA-1 values to recall the snapshots and we don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is basic information that the commit object stores for you. [2]

After implementing blob and tree objects, we now proceed to implement the commit object. While the root tree SHA is snapshot of the repository structure, the commit objects add metadata and creates the historical chain of changes. Our commit object follows this format:

```
commit {size}\u0000{content}

tree {tree_sha}
parent {parent_sha} (optional, can be multiple)
author {author_name} <{author_email}> {author_timestamp} {author_timezone}
committer {committer_name} <{committer_email}> {committer_timestamp} {committer_timezone}
\n
{commit_message}
```

```
data class Commit(
    val treeSha: String,
    val parentSha: String?,
    val author: String,
    val committer: String,
    val message: String
)
```

We have implemented command zsv commit with a message -m parameter, when executing zsv commit -m “Initial commit”, the `commit()` method in the commit service generates a tree object from the current directory state by calling `compressFromFile()` from tree service and it returns a SHA-1 hash identifying this tree. Then it retrieves the current HEAD pointer, which contains the SHA-1 of the most recent commit.

```
fun getCurrentHead(): String {
    val headPath = Paths.get(HEAD_FILE)
    if (!Files.exists(headPath)) {
        throw ObjectNotFoundException("Not a zsv repository")
    }
    val headContent = Files.readString(headPath).trim()

    return if (headContent.startsWith("ref: ")) {
        val branchReference = headContent.substringAfter("ref: ").trim()
        val branchPath = Paths.get("$ZSV_DIR/$branchReference")

        if (Files.exists(branchPath)) {
            Files.readString(branchPath).trim()
        } else {
            "" // New branch
        }
    } else {
        headContent
    }
}
```

Function `getCurrentHead()` checks for the existence of a HEAD file, which is in zsv/HEAD. There are two states for the HEAD file, it contains a reference to a branch like ref: refs/heads/main or it directly contains a commit SHA-1 known as “detached HEAD” state. When the HEAD file contains a branch reference (starting with “ref:”), the function extracts the branch path, then it looks up for actual file (zsv/ref/heads/main) and reads it commit SHA-1 stored in that file.

If branch file does not exist yet (which happens when creating a new branch that hasn’t had its first commit), it returns an empty string. If the HEAD file directly contains a commit SHA-1, it simply returns that value. This commit SHA-1 is then used as the parent reference when creating a new commit.

```
fun commit(message: String) {
   val treeSha = treeService.compressFromFile()
   val parentSha = getCurrentHead()
   val sha = compressFromMessage(message, treeSha, parentSha)
   // update branch commit
   eventPublisher.publishEvent(sha)
}
```

This `commit()` function is simply combination of root tree object of repository with additional information which are stored in separate object. It needs to retrieve the current HEAD pointer to create a chain of commits, and after creating object it publishes an event with the new commit SHA-1, which will be used to update the branch reference ensuring that HEAD now points to this latest commit.

The result is a parsed commit object that look like this:

```
tree    4d588df077909863fdadfb834e6ae668f1cf1d17
parent  0ee68bd10d375f3ebf96793f3f25e2f94f6322d3
author  zeljko <00zeljkostojkovic@gmail.com> 1727635374 +0200
committer zeljko <00zeljkostojkovic@gmail.com> 1727635374 +0200
message    message
```

2.3. Staging Area

As previously explained in chapter 1 section 1.3.1, staging area servers as intermediate layer between working directory and local repository, enabling developers to select which changes to include in next commit. In the previous implementations of blob, tree and commit objects, we required including all files in commit. However, this creates a limitation, developers often need to commit only a subset of their changes. This design follows the principle of separation of concerns where working directory represents the current state of files, staging area represents what will be committed and the local repository represents history of commits.

Staging area follows a principle known in software engineering as “two-phase commit” – changes are first prepared (staged) and then finalized (committed). This design pattern reduces errors and increases flexibility.

The index is implemented as a binary file stored at zsv/index. The binary format was chosen because of efficiency since binary storage is more compact and faster to parse than text file. It stores list of file names, along with their metadata and pointers to the object database (files that contain the contents of these files at the time they were added to the index)

The index file structure look like this:

```
DIRC <version> <entries>
<ctime> <mtime> <dev> <ino> <mode> <uid> <gid> <SHA> <flags> <path>
<ctime> <mtime> <dev> <ino> <mode> <uid> <gid> <SHA> <flags> <path>
…
```

```
data class IndexEntry(
    val offset: Int?,     // Byte offset in index (not stored)
    val ctime: Long,      // Creation time (8 bytes)
    val mtime: Long,      // Modified time (8 bytes)
    val dev: Long,        // Device ID (8 bytes)
    val ino: Long,        // Inode number (8 bytes)
    val mode: Int,        // Mode (4 bytes)
    val uid: Int,         // User ID (4 bytes)
    val gid: Int,         // Group ID (4 bytes)
    val sha: String?,     // SHA-1 hash (20 bytes)
    val flags: Int,       // Flags (4 bytes)
    val pathName: String  // Path + null terminator
)
```

Each field servers a specific purpose: offset tracks the position in the index file for efficient updates, ctime/mtime are used to detect file changes between working directory and index. dev/ino are used to detect file renames (same inode but different path). Mode is used to store file type and permissions, uid/gid preservers file ownership information. Flags store additional information about entry and pathName is relative path to the file in the repository. This collection of metadata is important for change detection, file tracking and permission preservation.

The index uses binary serialization for efficiency. The serialize method in IndexEntry data class writes an index entry to index.

```
fun serialize(file: RandomAccessFile, blobSha: String) {
    file.writeLong(ctime)
    file.writeLong(mtime)
    file.writeLong(dev)
    file.writeLong(ino)
    file.writeInt(mode)
    file.writeInt(uid)
    file.writeInt(gid)
    file.write(blobSha.toShaByte())
    file.writeInt(flags)
    file.writeBytes(pathName)
    file.write(0)
}
```

For reading index entry, we have deserialize method in same data class that parses binary data.

```
companion object {
    fun deserialize(buffer: ByteBuffer): IndexEntry {
        val offset = buffer.position()
        val ctime = buffer.getLong()
        val mtime = buffer.getLong()
        val dev = buffer.getLong()
        val ino = buffer.getLong()
        val mode = buffer.getInt()
        val uid = buffer.getInt()
        val gid = buffer.getInt()

        val sha = ByteArray(20)
        buffer.get(sha)

        val flags = buffer.getInt()
```

```
    val pathBuilder = StringBuilder()
    var byte = buffer.get()
    while (byte != 0.toByte()) {
      pathBuilder.append(byte.toInt().toChar())
      byte = buffer.get()
    }
    val pathName = pathBuilder.toString()

    return IndexEntry(
      offset, ctime, mtime, dev, ino,
      mode, uid, gid, sha.toHexString(),
      flags, pathName
    )
  }
```

For purpose of adding file to staging area, we implemented zsv add command. This command check if index file does not exist, and if so, it creates fresh empty index file with header only. Then for selected file it checks if already exist in the index. If it exists and has been modified then command updates the entry and if it is new, it adds a new entry.

```
fun saveFileToIndex(filePath: String) {

  if (!Files.exists(indexPath)) {
    Files.createFile(indexPath)
    writeIndexHeader(indexPath)
  }

  val filesToAdd: Set<String> = if (filePath == ".") {
    getAllFiles()
  } else {
    setOf(filePath)
  }

  for (file in filesToAdd) {
    addFileToIndex(file)
  }
}

private fun addFileToIndex(filePath: String) {
  val path = Paths.get(filePath)

  val indexEntry = IndexEntry.getFileAttributes(path)
  val index = parseIndexFile(false)

  index.firstOrNull {
    it.pathName == indexEntry.pathName ||
        it.ino == indexEntry.ino // check if file is renamed
  }?.let { entry ->
    if (entry.mtime != indexEntry.mtime) {
      updateIndexEntry(indexEntry, entry.offset!!)
    }
  } ?: writeIndexEntry(path, indexEntry)
}

private fun updateIndexEntry(indexEntry: IndexEntry, offset: Int) {
  RandomAccessFile(indexPath.toFile(), "rw").use { file ->
    file.seek(offset.toLong())
```

```
    val blobSha = blobService.compressFromFile(Paths.get(indexEntry.pathName))
    indexEntry.serialize(file, blobSha)
  }
}
```

When parsed, the index entries look like this:

```
[IndexEntry(offset=12, ctime=1738391494474, mtime=1738391494474, dev=66306, ino=57551483,
mode=100644, uid=1000, gid=1000, sha=2986dfb71be2e2769daae32e41a220c0f0dadd1e, flags=0,
pathName=.github/workflows/ci.yaml), IndexEntry(offset=106, ctime=1738391494474,
mtime=1738391494474, dev=66306, ino=57551484, mode=100644, uid=1000, gid=1000,
sha=9cb954d36b6374013396302f03da99c4a2a8c9b5, flags=0, pathName=.gitignore),
IndexEntry(offset=185, ctime=1738391494474, mtime=1738391494474, dev=66306, ino=57551485,
mode=100644, uid=1000, gid=1000, sha=95d9e7085fd75626c3fe5898aeef3ee176d16f94, flags=0,
pathName=.sdkmanrc), IndexEntry(offset=263, ctime=1738391494474, mtime=1738391494474,
dev=66306, ino=57551486, mode=100644, uid=1000, gid=1000,
sha=f553d7ab8398c2e74eff51dc8f04f40b68681488, flags=0, pathName=README.md),
IndexEntry(offset=341, ctime=1738391494474, mtime=1740443862532, dev=66306, ino=57551487,
mode=100644, uid=1000, gid=1000, sha=9b886f89a838e74daba4e5f4adf44a58687fcd42, flags=0,
pathName=build.gradle.kts),
...
```

2.4. Storage Optimization Techniques

Process of converting files to objects are straightforward, firstly we need to apply compression to file content. We do this because we want to minimize the file's disk usage. If we just copied the entire file, it would start too much space on disk. The compression is implemented using zlib.

Zlib is a software library used for data compression as well as a data format, it was written by Jean-loup Gailly and Mark Adler and is an abstraction of DEFLATE compression algorithm used in their gzip file compression program. As of September 2018, zlib only supports one algorithm, called DEFLATE, which uses a combination of a variation of LZ77 (Lampel-Ziv 1997) and Huffman coding. This algorithm provides good compression on wide variety of data with minimal use of system resources. This is also the algorithm used in the Zip achieve format [10]

LZ77 is responsible for eliminating repeated sequences of data. It replaces identical sections of data with metadata pointers that indicate the position and length of the previous occurrence. For example, in a sequence "AABCBBABC", repeated patterns like "ABC" would be replaced with pointer to their previous occurrence.

Each pointer contains how far back is offset and how many bytes to copy from the position. Using LZ77 sequence "AABCBBABC" would become "(0,0)A(0,0)B(0,0)C(2,1)(1,1)(5,3)

Step	Position	Match	Byte	Output
1.	1	--	A	(0,0)A
2.	2	A	--	(1,1)

3.	3	--	B	(0,0)B
4.	4	--	C	(0,0)C
5.	5	B	--	(2,1)
6.	6	B	--	(1,1)
7.	7	7	ABC	(5,3)

Figure 9 Compression Process LZ77 [11]

The result of compression, conceptually, is the output column that is, a series of bytes and optional metadata that indicates whether that byte is preceded by some sequence of bytes that is already in the output.

Step	Input Pointer	Append Bytes	Output stream
1.	(0,0)A	A	A
2.	(1,1)	A	AA
3.	(0,0)B	B	AAB
4.	(0,0)C	C	AABC
5.	(2,1)	B	AABCB
6.	(1,1)	B	AABCBB
7.	(5,3)	ABC	AABCBBABC

Figure 10 Decompression process LZ77 [11]

After LZ77 processing, Huffman coding compresses the result by assigning shorter codes to frequently occurring pattern in the data. Huffman coding algorithm was developed by David A. Huffman in 1950, and it can be easily explained through an example.

d	a	t	a	s	t	r	u	c	t	u	r	e	s

Assuming each character requires 8 bits for representation, and we have 14 characters, we would need 8 x 14 = 112 bits to transmit this string over a network. The idea of Huffman coding is to compress such strings by creating frequency tree of characters. [12]

Following table shows the frequency calculation for each character in string in ascending order.

d	c	e	a	s	r	u	t
1	1	1	2	2	2	2	3

After sorting, these characters are stored in a priority queue (PQ). Each character becomes a leaf in the tree through the following steps:

For example, if we have string containing character frequencies of 1,1,1,2,3, we make

priority queue out of this character frequencies. In the first iteration, we take the two smallest values (1,1) and create a new node with their sum of 2. After removing these two values from the queue and inserting the new sum of 2, we continue with the next iteration. The queue now contains 1,2,2,3 and the process repeat with the next pair of smallest values. This iterative process continues until only one value remains in the queue, which will represent the root of Huffman tree.

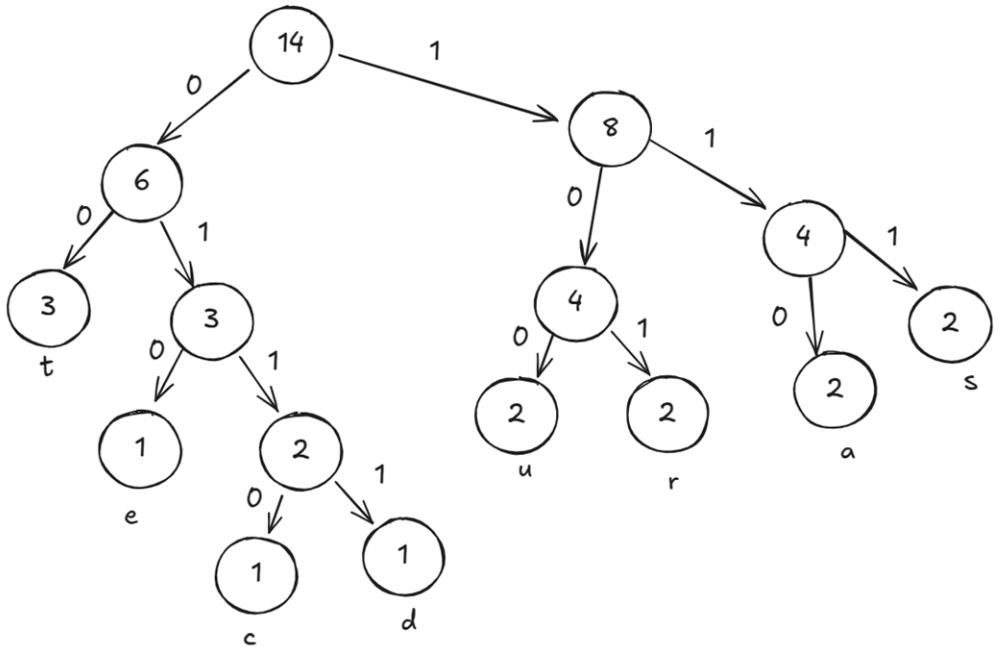

Figure 11 Huffman Tree [12]

In ASCII encoding, every character requires 8 bits for storage regardless of its frequency, by using Huffman coding we can reduce that by assigning variable-length binary codes based on how frequently each character appears. Character ‘t’ is now represented by just 2 bits (00), similarly ‘a’ is encoded with 3 bits (110). WE managed to reduce the total size of string from 112 bits to 41 bits.

Character	Frequency	Code	Bits
a	2	110	6
c	1	0110	4
d	1	0111	4
e	1	010	3
r	2	101	6
s	2	111	6
t	3	00	6
u	2	100	6
			41

Figure 12 Huffman encoding table for characters in compressed string [12]

Huffman tree ensures that no code is a prefix of another code, which guarantees that the compressed data can be decoded back to the original characters. By following the encoded bits from the root to leaf nodes, where 0 means go left, and 1 means go right. Once leaf node is reached, we extract that character and return to the root to continue decoding the remaining bits. If we receive bit sequence “00110”, we first follow “00” from the root which leads us to character ‘t’, then we return to root and follow “110” which leads us to character ‘a’ thus decoding “00110” into ta.

2.5. Local Repository Branching

Branching means you diverge from the main line of development and continue to do work without messing with that main line. To understand the value of branching, consider this common development scenario, you are working on a software project that is currently in production and your team needs to both develop new features and fix critical bugs in the production. Without branching, managing this parallel development would be chaotic, changes for the latest version could interfere with bug fixes, and vice versa. Using branches solves this problem we can maintain a master branch containing the stable production code, create develop branch for new features and create a separate bugfix branch from maser branch. This way developers working on new features do not interfere with those fixing bugs, and fixes can be quickly merged back into production code without including incomplete features.

Branching in DVCS is lightweight because we do not store data as changesets or differences but series of snapshots, when we make a commit, that object contains a pointer to the snapshot of the content we committed [2]. Let us recall our earlier example of file modifications through the Merkle tree. When we modified README.md by adding content, we saw how this created a new blob object with different hash, which in turn required creating a new tree object and a new commit (a4ghl2). This commit stores a pointer to the commit that came immediately before it.

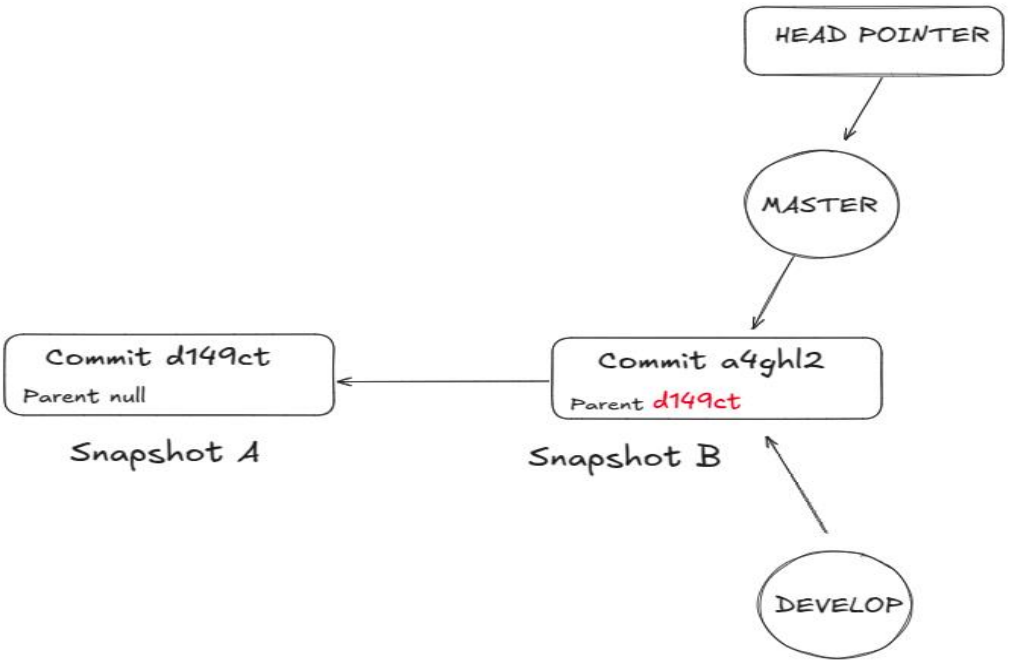

Figure 13 Repository state with commit history and branch references

Figure 13 illustrates how branching works. The diagram shows two commits (d149ct and a4ghl2) where each commit has a reference to its parent commit, with the initial commit having a null parent. The HEAD pointer indicates the current active branch (master), while both master and develop branches are pointing to the same commit (a4ghl2). This represents a state just after creating develop branch but before making any changes on it. From this point new commits on either branch will divergently paths of development, allowing parallel work streams to progress independently.

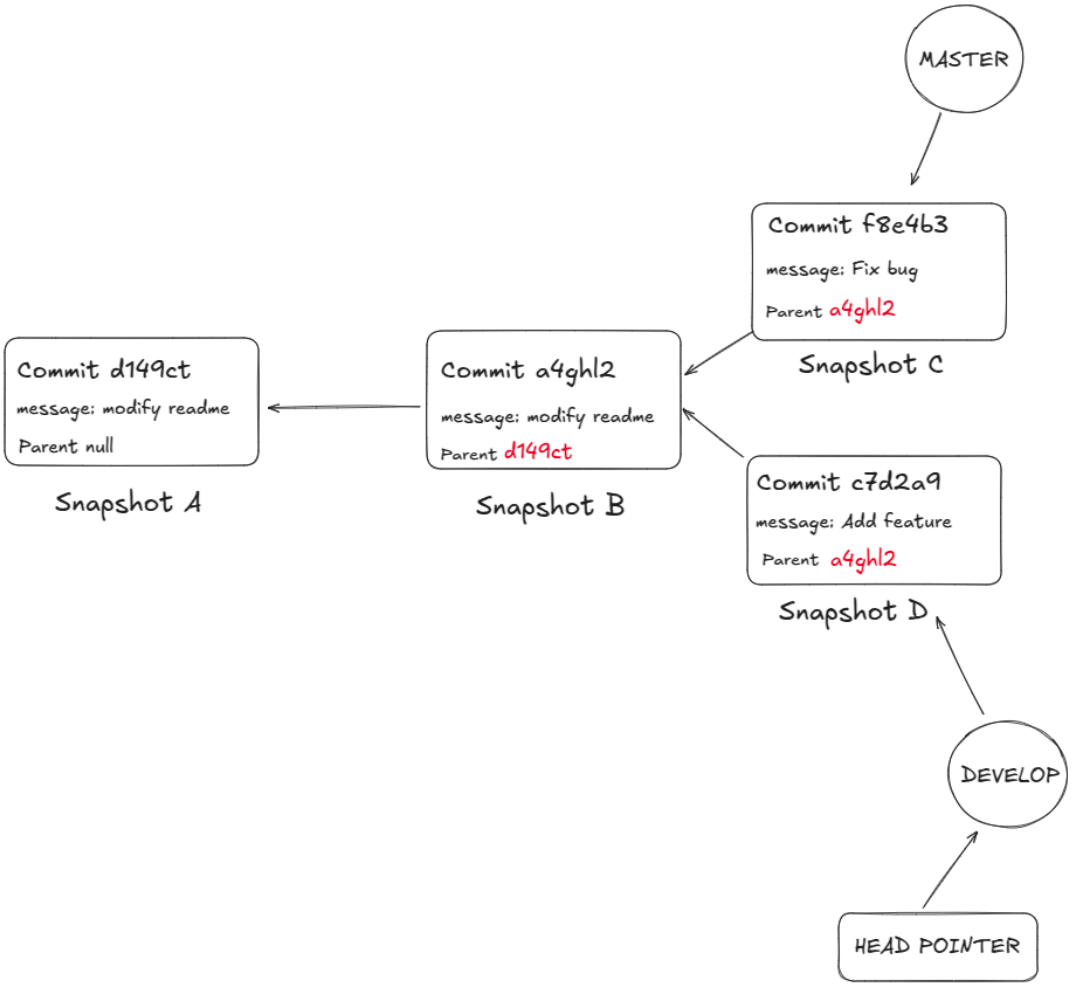

Figure 14 Parallel development with divergent branches

After committing our changes to snapshot B (a4ghl2), development can proceed in parallel on different branches. In example above, the master branch contains a bug fix (f8e4b3) while the develop branch includes a new feature addition (c7d2a9). This is a common scenario in software development where urgent fixes need to be applied to the production code while new features are being developed simultaneously. Since these changes exist in parallel branches, they will eventually need to be merged to combine both the bug fix and the new feature into single codebase.

In our implementation, branches are represented as files in the zsv/refs/heads/ directory, with each file containing the SHA-1 hash of the commit it points to. The currently active branch is referenced by the HEAD file .zsv/HEAD. We implemented two operations for creating new branch and immediately switching to it and for switching to already existing one.

```
@Command(command = ["checkout"], description = "Switch to existing branch")
fun checkout(
    @Option(required = true, description = "Branch name to checkout") branchName: String
): String {
    branchService.checkout(branchName, isNewBranch = false)
    return "Switched to branch $branchName"
}
```

```
@Command(command = ["checkout -b"], description = "Create and switch to new branch")
fun checkoutNewBranch(
    @Option(required = true, description = "New branch name") branchName: String
): String {
    branchService.checkout(branchName, isNewBranch = true)
    return "Switched to a new branch $branchName"
}
```

The checkout method manages both creating new branch and switching to existing one:

```
fun checkout(branchName: String, isNewBranch: Boolean) {
    if (!validateCheckout(branchName)) {
        throw IllegalStateException("Already on branch $branchName")
    }
    if (isNewBranch) {
        createAndCheckoutBranch(branchName)
    } else {
        checkoutExistingBranch(branchName)
    }
}
```

Depending on boolean `isNewBranch` is true or not, two different methods are called. If is true `createAndCheckoutBranch()` will be called, this method creates a new branch pointing to the current commit and updates HEAD. If `isNewBranch` is false, `checkoutExistingBranch()` will be called, this method switches to an existing branch by updating the working directory and HEAD. When switching branches, the system calculates the difference between the current and target branch trees, then applies those changes to the working directory. This ensures that the files on disk match the state of the checked-out branch. `checkoutExistingBranch()` implementation:

```
val commitSha = getShaFromBranchName(branchName) // target branch tree
val (targetTreeSha, _) = commitService.decompress(commitSha)

val currentCommit = getCurrentHead() // current branch tree
val (currentTreeSha, _) = commitService.decompress(currentCommit)
val changes = treeService.compareTwoTrees(targetTreeSha, currentTreeSha)

applyChangesToWorkingDirectory(changes)
updateHeadReference(branchName)
```

This implementation follows a principle of minimum file changes, it only updates files that differ between branches, rather than recreating the entire working directory, improving performances when switching between branches with many common files.

The detailed implementation of the tree comparison algorithm (`compareTwoTrees()`) and the working directory update mechanism (`applyChangesToWorkingDirectory()`) can be found in GitHub repository [https://github.com/zstojkovic00/abstractive-version-control-system-manager/blob/master/src/main/kotlin/com/zeljko/abstractive/zsv/manager/core/services/BranchService.kt] while the following section provides a conceptual overview of their functionality.

2.6. Local Repository Merging

Merge unifies two or more commit history branches. It allows developers to make and record changes independently and it permits the two developers to combine their changes at any time, all without a central repository. When there are no conflicts between changes in different branches, it automatically computes merge result and creates a new commit. In cases where conflict arise (when changes compete to modify the same line in the same file) it marks these changes as "unmerged" and leaves the resolution of conflicts to the developer. This means that manual intervention is required to clarify which changes should be retained [6]

There are three scenarios when merging:

1. Fast-forward merge: This occurs when the target branch is directly ahead of current branch, meaning no divergent works need to be combined. The current branch pointer simply moves forward to point to the same commit as the target branch.

2. Three-way merge: This occurs when the two branches have diverged, requiring a new merge commit that combines changes from both branches. This merge uses three commits: the two branch tips and their common ancestor (Last Common Ancestor – LCA)

3. Merge with conflicts: When the same part of file has been modified differently in both branches, a merge conflict occurs. This requires manual intervention to resolve the conflicts before the merge can complete.

The merge algorithm begins by identifying which type of merge is needed:

```
val currentCommitSha = getCurrentHead()
val targetCommitSha = getShaFromBranchName(targetBranchName)
val lca = findLastCommonAncestor(currentCommitSha, targetCommitSha)

if (lca == targetCommitSha) {
   println("Already up-to-date")
} else if (lca == currentCommitSha) {
   println("Fast forward")
   fastForwardMerge(targetCommitSha, currentCommitSha)
} else {
   threeWayMerge(lca, targetCommitSha, currentCommitSha)
}
```

This algorithm first finds the Last Common Ancestor (LCA) of the two branches. Then it uses this information to determine which of three scenarios applies. IF the LCA equals the target branch's commit, it means the target branch has not progressed beyond

what is already in the current branch, so no merge is needed. If the LCA equals the current branch's commit, it means the current branch has not progressed beyond the point target branch diverged. In this case, we can simply move the current branch pointer forward to the target branch's position. If neither of the above condition is met, it means both branches have diverged from their common ancestor, and a more complex merge operation is required.

To find the Last Common Ancestor (LCA), the algorithm first determines the depth of each commit from the initial commit. It then equalizes these depths by moving the deeper commit upward in its history. Once both commits are at the same depth, it walks both branches backward in parallel until it finds a common commit, which represents their common ancestor. Implementation of this algorithm can be found on previous mentioned GitHub link in section 2.5.

We can show three-way merge process with our previous example in section 2.5. where master branched into two separate paths, bug fix in the master branch and a new feature in the develop branch. To merge these changes, the system needs three points of references: current branch (f8e4b3 with bug fix), target branch (c72d2a9 with new feature) and their last common ancestor – LCA (a4ghl2). By comparing these three states, the system can determine which changes were made in each branch and combine them into a new merge commit.

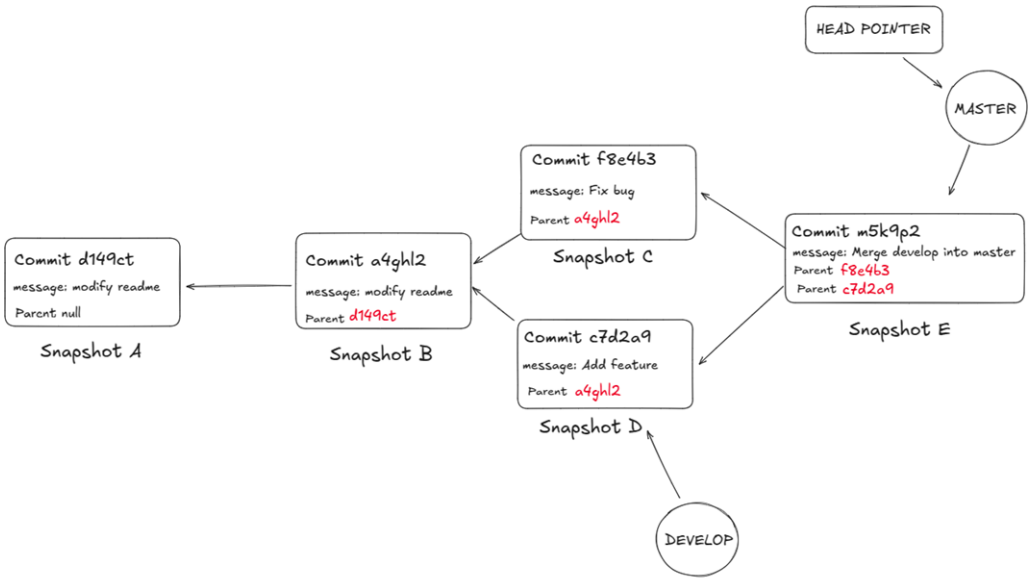

Figure 15 Three-way merge

Figure 15 illustrates three-way merging, creating Snapshot E (m4k9p2). This merge commit is special because it has two parent commits – f8e4b3 from the master branch and c7d2a9 from the develop branch. After the merge, the HEAD pointer points at master branch while the develop branch remains at its last commit (c7d2a9). This maintains the history of both development paths while combining their change.

```
private fun threeWayMerge(lca: String, targetCommitSha: String, currentCommitSha: String) {
  val baseTree = treeService.decompress(false, commitService.decompress(lca).treeSha)
  val targetTree = treeService.decompress(false, commitService.decompress(targetCommitSha).treeSha)
  val currentTree = treeService.decompress(false,
```

```
commitService.decompress(currentCommitSha).treeSha)
    val changes = treeService.compareThreeTrees(baseTree, targetTree, currentTree)
  changes.forEach { (action, file) ->
    when (action) {
      "NO_CONFLICT" -> {applyNonConflictingChange(file)
      "CONFLICT" -> {markFileWithConflict(file)}
      "AUTO_MERGE" -> {autoMergeFile(file)}
    }
  }
  createMergeCommit(Merge",currentCommitSha, targetCommitSha) }
```

2.7. Remote Repository

Remote Repository is final component in DVCS. In previous sections, we covered the Working Directory, Staging Area/Index, and Local repository where all objects mentioned earlier are stored. As noted at the beginning in section 1.3.1, a remote repository is necessary for synchronization between multiple developers and servers as the single source of truth for a project and its state.

Once developer completes their work on the local repository, they can send a push command to update changes on the central server. Of course, before doing this, they must pull any changes other developers have made to ensure everyone has the desired repository state. Git internally supports four major protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git protocol.

The Local protocol is the most basic and is often used if everyone on a team has access to a shared filesystem such as an NFS mount. HTTP comes in two variants: the "Smart" HTTP protocol that operates similarly to SSH but runs over standard HTTP/S ports with various authentication mechanisms, and the "Dumb" HTTP protocol that serves the Git repository like normal files from a web server. SSH is a common transport protocol for Git when self-hosting, as it is authenticated, secure, and efficiently compresses data. The Git protocol is a special daemon that listens on port 9418 and provides a service like SSH but without authentication [2]

Since we are building our own application, and our folder works with zsv rather than. git, we cannot use Git's implementations and must create our own. These topics are quite complex and would require several months of development. I have experimented with creating a Git native protocol implementation for the clone command, which can be seen in Appendix N, but for the purpose of this thesis, we will create a simpler integration with MinIO which is enough for understanding distribute nature of VCS.

MinIO is a high-performance object storage system. It is designed to be an alternative to cloud-native storage systems. In fact, its API is fully compatible with Amazon S3. MinIO also provides a variety of deployment options. It can run as a native application on most popular architectures and can also be deployed as a containerized application using Docker or Kubernetes. [13]. For our implementation, we will use MinIO because its open-source and we can easily run it in Docker container to simulate a remote server that stores our version control objects.

To enable remote collaboration, we have added a push command to synchronize our local repository with a remote repository stored on a MinIO server. The design has been

focused on the core ability needed to collaborate on code changes between developers.

```
@Command(command = ["push"], description = "Push to remote server")
fun push(
    @Option(shortNames = ['b'], required = false, description = "Branch name") branchName: String =
"master"
): String {
    return minioService.push(branchName)
}
```

```
fun push(branchName: String): String {
    val bucket = DEFAULT_BUCKET_NAME
    createBucketIfNotExist(bucket)

    val objectsPushed = pushMissingObjects(bucket)
    val refPushed = pushBranchReference(bucket, branchName)

    return "Push completed: $objectsPushed objects, branch reference ${if (refPushed) "updated" else
"unchanged"}"
}
```

This command takes an optional branch name parameter (default is master) and delegates the actual push operations to push method in MinIO service, this method checks if bucket exist and if necessary, it creates bucket. Then we compare local objects with remote objects and push only those that do not exist remotely. This optimization avoids redundant transfers. After objects are synchronized, we must update the branch reference to point to the latest commit. It should be noted that this implementation lacks many features that Git's push operation provides, such as handling merge conflicts, enforcing fast-forward updates, implementing access control.

To complement the push operation, we also implemented a pull command that retrieves changes from the remote repository.

```
@Command(command = ["pull"], description = "Pull from remote server")
fun pull(
    @Option(shortNames = ['b'], required = false, description = "Branch name") branchName: String =
"master"
): String {
    return minioService.pull(branchName)
}

fun pull(branchName: String): String {
    val bucket = DEFAULT_BUCKET_NAME

    if (!minioClient.bucketExists(BucketExistsArgs.builder().bucket(bucket).build())) {
        return "Remote repository does not exist"
    }

    val objectsPulled = pullMissingObjects(bucket)
    val refUpdated = updateLocalBranchReference(bucket, branchName)

    return "Pull completed: $objectsPulled objects, branch reference ${if (refUpdated) "updated" else
"unchanged"}"
}
```

The pull operation works in the opposite direction of push, it retrieves objects that exist on the remote server but not locally, and ten adjusts the local branch pointer to the remote state. This allows developers to synchronize their local repositories with the latest changes made by other team members.

3. Comparative Analysis of Git and Zsv

This chapter offers a comprehensive comparative analysis between our implementation (Zsv) developed in this thesis, which is JVM-based, and Git developed in the C programming language. The analysis considers qualitative factors, examining how varying technological approaches affect performance and storage efficiency in DVCS.

3.1. Quantitative metrics

To conduct a comparative performance analysis between ZSV and Git, a benchmarking program was developed to measure compression times on files of varied sizes. The tests for ZSV were run from within the Java Virtual Machine (JVM) environment, while the tests for Git were executed natively in the terminal to ensure a fair comparison without the potential overhead of invoking Git from the JVM. The benchmarks were performed on a computer with the following specifications:

- Processor: Intel Core Ultra 7 155H
- RAM: 32GB DDR5
- SSD: 1TB

Three sample binary files were generated for the benchmarking:

```
val testSizes = mapOf(
   "small" to 10 * 1024 * 1024,  // 10 MB
   "medium" to 100 * 1024 * 1024, // 100 MB
   "large" to 500 * 1024 * 1024 // 500 MB
)
```

The compression and object storage times for both Zsv and Git were measured for each of these files. For Git we used the Linux time command combined with git hash-object -w {fileName}, while the Zsv implementation used a custom annotation integrated with Spring Boot's StopWatch class for measuring command execution time. The results are summarized in Figure 16 below:

File	Orginal Size	Zsv Compression Time	Git Compression Time
file_small.bin	10 MB	447573539 ns	369653559 ns
file_medium.bin	100 MB	3578232908 ns	3599355722 ns
file_large.bin	500 MB	17692216380 ns	17646301449 ns

Figure 16 Comparison of compression times

The 10 MB file (file_small.bin) shows Git outperforming Zsv by compressing the file in around 369.65 ms, while Zsv takes around 447.57 ms, representing a difference of about 77.92 ms in favor of Git, which indicates that Git is much more efficient for smaller files. As we move to the larger 100 MB file (file_large.bin), Zsv takes less time to compress the file than Git, with a difference of around 0.02, suggesting that both systems have comparable performance for files of this size. The largest file evaluated, the 500 MB file (file_large.bin), shows Zsv and Git having very similar compression times, with Zsv compressing the file in around 17.69 seconds, while Git takes around 17.65 seconds. The difference is negligible, indicating that both systems scale well for large files.

Regarding decompression, the Zsv implementation required around 10.27 seconds to decompress the largest file. Direct comparison of decompression performance was not possible due to Git's requirement of outputting the entire file contents to the terminal. Admittedly, Linux's time command shows that Git's native implementation processes the 500 MB file much faster compared to the Zsv implementation, with system and user times being around 0.37s and 8.75s, and total time of 47s, which is mostly due to I/O overhead.

For compression size comparison, we cannot use a binary file. Since binary files are often generated randomly, they typically lack repetitive patterns, making them difficult to compress using algorithms like LZ77, which was discussed in 2.4. section. For the purposes of measuring the compression efficiency of these implementations, we will use three PDF files of various size: 1.1 MB (our thesis), 8 MB, 17 MB.

File	Original Size	Zsv compression size	Git compression size
thesis.pdf	1.1 MB	779 KB	773 KB
pro-git.pdf	8 MB	7.1 MB	6.9 MB
vcs-2-edition.pdf	17 MB	14.5 MB	14 MB

Figure 17 Comparison of compression size

The smallest file (thesis.pdf, 1.1 MB) achieved around 30% compression on both systems, with the medium file (pro-git.pdf, 8 MB) compressed by around 12-13% and the largest file (vcs-2-edition.pdf, 17 MB) showing compression of about 15-16%. While Git was slightly better in compression efficiency, the Zsv JVM-based implementation proved to be a strong rival. This close performance across different file sizes indicates the effectiveness of the JVM implementation. The same creation of the SHA-1 hashes attests to the compatibility and consistency of the Zsv system.

To test the repository size differences between Zsv and Git, we will use abstractive-history-tracking-system-manager project, this is a project that contains the implementation of this thesis. After initializing the repository, results are below:

- Zsv repository: 32 KB
- Git repository: 56 KB

The Zsv repository has a smaller repository size, which is expected due to Git having much more developed and functionality built into its repository structure. After performing commits, the repository size for both Zsv and Git remained unchanged when adding files using `echo "nesto" > test1, test2, test3`, indicating that the Zsv implementation scales well and effectively manages object storage.

4. Conclusion

In this thesis, we set out to demystify the internal workings of distributed version control systems, particularly Git, by implementing such a system from scratch. Our primary question was, how do the components and mechanisms of Git operate under the hood, and can they be effectively implemented in a JVM-based language that does not offer low-level memory control like C.

Through our practical implementations, we have been able to answer both questions. We explained the internal workings of distributed version control systems at architectural and implementation levels. At the architectural level, we provided fundamental explanations of distributed code versioning systems by describing each component, examining how local, then centralized, and finally distributed version control system evolved, discussing the advantages that led to this evolution in software process.

At the implementation level was core of our thesis, and it was dedicated to both theory and practical implementation of object storage in distributed version control systems. We demonstrated complete lifecycle of transforming a file into an object, describing the essential process of compression, hashing, and storing with abstraction of special objects that represents UNIX's file and directory hierarchies. Through our implementation, we also demonstrated how content-addressable storage works with SHA-1 hashing to construct a Merkle tree structure, enabling efficient tracking of changes throughout the repository's history.

By implementing branching, merging, and remote communication capabilities, we have illustrated how distributed version control systems enable developers to work independently even without access to a network, yet still have the option of synchronizing with a central repository when needed. This approach provides the option of local development, and the advantages of coordination realized through a centralized system without making the central authority a single point of failure.

Our comparative analysis between JVM-based implementation and Git's native C implementation answers our second thesis question. Though Git's C implementation showed better performance, especially with smaller files, our JVM-based solution showed good performance for having been coded by a single individual versus Git's thousands of contributions over years. These findings suggest that languages with less direct memory control such as Kotlin would make sense as alternatives for small repositories or specialized use like plugins. With increasing size of repository, the performance gap would increase to benefit C's memory optimization for production-scale systems. Kotlin could be well-suited for developing specialized Git plugins or extensions where its strong typing, and development efficiency would be advantageous because for plugins runtime efficiency is not matter of urgency.

No significant architectural adaptations were required to implement Git-like functionality within the JVM ecosystem. The Kotlin and Java Virtual Machine environment provided everything necessary to replicate distributed version control system.

Although this implementation was successful in demonstrating the internal workings of a DVCS, there are a few aspects that are left to experimentation. More advanced network protocols than our MinIO-based solution would be needed to necessary to take distributed capability of the system to the next level. Our solution relies on MinIO for remote storage,

which supports authentication but lacks the customized protocol optimized for DVCS objects transfer. Additionally, exploring use of a different hashing algorithms such as SHA-256 would also mitigate the theoretical collision vulnerabilities of SHA-1. Implementing such capabilities would address the limitations of our implementation and enhance its security and reliability even further.

Another interesting direction would be the application of artificial intelligence (AI) in DVCS. AI could be used to resolve merge conflicts based on related JIRA tickets/issues, understanding the intent of changes, and learning from previous conflict resolutions. An AI system could understand code semantics rather than just text changes, allowing for sophisticated merge strategies not natively provided by DVCS. Combination of AI and DVCS would possible solve one of most challenging aspects of collaborative software development.

This thesis has significance to the software engineering filed, by revealing the internal workings of Git's object model, we have created and educational resource that helps bridge the knowledge gap between using these tools and understanding them. Understanding these concepts enables developers to make informed choices when faced with complex issues like merge conflicts or repository corruption that require knowledge of the internal workings. Our documented implementation gives this information in a form that is useful to developers without for them having to conduct their own reverse engineering o

References

1. https://about.gitlab.com/topics
2. Chacon, S., & Straub, B. (2014). Pro Git (2 nd ed.). Apress
3. https://medium.com/@pm74367/types-of-vcs-version-control-system-f7af5594bd04
4. https://estuary.dev/blog/distributed-architecture
5. Spolsky, J. (2010, March 17). Distributed Version Control Is Here to Stay, Baby. Joel on Software. [https://www.joelonsoftware.com/2010/03/17/distributed-version-control-is-here-to-stay-baby]
6. Loeliger, J., & McCullough, M. (2012). Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development (2nd ed.). O'Reilly Media.
7. Eastlake, D., & Jones, P. (2001). US Secure Hash Algorithm 1 (SHA1) (RFC 3174). Internet Engineering Task Force. [https://tools.ietf.org/html/rfc3174]
8. https://softwareengineering.stackexchange.com/questions/301400/why-is-the-git-git-objects-folder-subdivided-in-many-sha-prefix-folders
9. Benet, J. (2014). IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561. [https://arxiv.org/abs/1407.3561]
10. https://en.wikipedia.org/wiki/Z-Library
11. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb
12. Leonard, A. (2024). Java Coding Problems (2nd ed.). Packt Publishing
13. https://min.io/docs/minio/linux/index.html
14. Bassem D. (2023). Understanding the git internals, a deep dive into the git folder [https://www.youtube.com/watch?v=VJB-TYo9_DY]
15. Jon G. (2024) Implementing (parts of) git from scratch in Rust [https://www.youtube.com/watch?v=u0VotuGzD_w]
16. Stefan S. (2013) Reimplementing “git clone” in Haskell from the bottom up [https://stefan.saasen.me/articles/git-clone-in-haskell-from-the-bottom-up]