Efficient Synchronization of Linux Memory Regions over a Network: A Comparative Study and Implementation

TODO: Add subtitle

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Abstract

TODO: Add abstract

Contents

1 Introduction

TODO: Add introduction

2 Technology

2.1 The Linux Kernel

The open-source Linux kernel, was created by Linus Torvalds in 1991. Developed primarily in the C programming language, it has recently seen the addition of Rust as an approved language for further expansion and development, esp. of drivers[1]. The powers millions of devices across the globe, including servers, desktop computers, mobile phones, and embedded devices. It serves as an intermediary between hardware and applications, as an abstraction layer that simplifies the interaction between them. It is engineered for compatibility with a wide array of architectures, such as ARM, x86, RISC-V, and others.

The kernel does not function as a standalone operating system. This role is fulfilled by distributions, which build upon the Linux kernel to create fully-fledged operating systems[2]. Distributions supplement the kernel with additional userspace tools, examples being GNU coreutils or BusyBox. Depending on their target audience, they further enhance functionality by integrating desktop environments and other software.

The open-source nature of the Linux kernel makes it especially interesting for academic exploration and usage. It offers transparency, allowing anyone to inspect the source code in depth. Furthermore, it encourages collaboration by enabling anyone to modify and contribute to the source code. This transparency, coupled with the potential for customization and improvement, makes developing for the Linux kernel a good choice for this thesis.

2.2 Linux Kernel Modules

Linux is a extensible, but not a microkernel. Despite it’s monolithic nature, it allows for the integration of kernel modules[2]. Kernel modules are small pieces of kernel-level code that can be dynamically incorporated into the kernel, presenting the advantage of extending kernel functionality without necessitating system reboots.

The dynamism of these modules comes from their ability to be loaded and unloaded into the running kernel as per user needs. This functionality aids in keeping the kernel size both manageable and maintainable, thereby promoting efficiency. Kernel modules are traditionally developed using the C programming language, like the kernel itself, ensuring compatibility and consistent performance.

Kernel modules interact with the kernel via APIs (Application Programming Interfaces). Despite their utility, since they run in kernel space, modules do carry a potential risk. If not written with careful attention to detail, they can introduce significant instability into the kernel, negatively affecting the overall system performance and reliability.

Modules can be managed and controlled at different stages, starting from boot time, and be manipulated dynamically when the system is already running. This is facilitated by utilities like `modprobe` and `rmmod`[3].

In the lifecycle of a kernel module, two key functions are of significance: initialization and cleanup. The initialization function is responsible for setting up the module when it's loaded into the kernel. Conversely, the cleanup function is used to safely remove the module from the kernel, freeing up any resources it previously consumed. These lifecycle functions, along with other such hooks, provide a more structured approach to module development.

2.3 UNIX Signals and Handlers

UNIX signals are an integral component of UNIX-like systems, including Linux. They function as software interrupts, notifying a process of significant occurrences, such as exceptions. Signals may be generated from various sources, including the kernel, user input, or other processes, making them a versatile tool for inter-process notifications.

Aside from this notification role, signals also serve as an asynchronous communication mechanism between processes or between the kernel and a process. As such, they have an inherent ability to deliver important notifications without requiring the recipient process to be in a specific state of readiness[4]. Each signal has a default action associated with it, the most common of which are terminating the process or simply ignoring the signal.

To customize how a process should react upon receiving a specific signal, handlers can be utilized. Handlers dictate the course of action a process should take when a signal is received. Using the `sigaction()` function, a handler can be installed for a specific signal, enabling a custom response to that signal such as reloading configuration, cleaning up ressources before exiting or enabling verbose logging [5].

It is however important to note that signals are not typically utilized as a primary inter-process communication (IPC) mechanism. This is primarily due to their limitation in carrying additional data. While signals effectively alert a process of an event, they are not designed to convey forther information related to that event; consequently, they are best used in scenarios where simple event-based notifications are sufficient, rather than for more complex data exchange requirements.

2.4 Principle of Locality

The principle of locality, or locality of reference, refers to the tendency of a processor in a computer system to recurrently access the same set of memory locations within a brief span of time. This principle forms the basis of a predictable pattern of behavior that is evident across computer systems, and can be divided into two distinct types: temporal locality and spatial locality[6].

Temporal locality revolves around the frequent use of particular data within a limited time period. Essentially, if a memory location is accessed once, it is probable that this same location will be accessed again in the near future. To leverage this pattern and improve performance, computer systems are designed to maintain a copy of this frequently accessed data in a faster memory storage, which in turn, significantly reduces the latency in subsequent references.

Spatial locality, on the other hand, refers to the use of data elements that are stored in nearby locations. That is, once a particular memory location is accessed, the system assumes that other nearby locations are also likely to be accessed shortly. Therefore, to optimize performance, the system tries to anticipate these subsequent accesses by preparing for faster access to these nearby memory locations. Temporal locality is considered a unique instance of spatial locality, demonstrating how the two types are closely interlinked.

A specific instance of spatial locality, termed sequential locality, occurs when the data elements are organized and accessed in a linear sequence. An example of this is when elements in a one-dimensional array are traversed systematically, accessing the elements one by one in their sequential order.

Locality of reference can be instrumental in improving the overall performance of a system. To achieve this, a variety of optimization techniques are deployed, such as caching, which stores copies of frequently accessed data in quick-access memory, and prefetching for memory, which involves loading potential future data into cache before it’s actually needed.

2.5 Memory Hierarchy

The memory hierarchy in computers is an organized structure based on factors such as size, speed, cost, and proximity to the Central Processing Unit (CPU). It follows the principle of locality, which suggests that data and instructions that are accessed frequently should be stored as close to the CPU as possible[7]. This principle is crucial primarily due to the limitations of “the speed of the cable”, where both throughput and latency decrease as distance increases due to factors like signal dampening and the finite speed of light.

TODO: Add graphic of the memory hierarchy

At the top of the hierarchy are registers, which are closest to the CPU. They offer very high speed, but provide limited storage space, typically accommodating 32-64 bits of data. These registers are used by the CPU to perform operations.

Following registers in the hierarchy is cache memory, typically divided into L1, L2, and L3 levels. As the level increases, each layer becomes larger and less expensive. Cache memory serves as a buffer for frequently accessed data, with predictive algorithms typically optimizing its usage.

Main Memory, i.e. Random Access Memory (RAM), provides larger storage capacity than cache but operates at a slower speed. It typically stores running programs and open files.

Below main memory, we find secondary storage devices such as Solid State Drives (SSD) or Hard Disk Drives (HDD). Although slower than RAM, these devices can store larger amounts of data and typically contain the operating system and application binary fies. Importantly, they are persistent, meaning they retain data even after power is cut.

Tertiary storage, including optical disks and tape, is slow but very cost-effective. Tape storage can store very large amounts of data for long periods of time. These types of storage are typically used for archiving or physically transporting data, such as importing data from personal infrastructure to a service like AWS[8].

The memory hierarchy is not static but evolves with technological advancements, leading to some blurring of these distinct layers[9]. For instance, Non-Volatile Memory Express (NVMe) storage technologies can rival the speed of RAM while offering greater storage capacities. Similarly, some research, such as the work presented in this thesis, further challenges traditional hierarchies by exposing tertiary or secondary storage with the same interface as main memory.

2.6 Memory Management in Linux

Memory management forms a cornerstone of any operating system, serving as a critical buffer between applications and physical memory. Arguably, it can be considered one of the fundamental purposes of an operating system itself. This system helps maintain system stability and provides security guarantees, such as ensuring that only a specific process can access its allocated memory.

Within the context of the Linux operating system, memory management is divided into two major segments: kernel space and user space.

Kernel space is where the kernel itself and kernel modules operate. The kernel memory module is responsible for managing this segment. Slab allocation is a technique employed in kernel space management; this technique groups objects of the same size into caches, enhancing memory allocation speed and reducing fragmentation of memory[10].

User space is the memory segment where applications and certain drivers store their memory[11]. User space memory management involves a paging system, offering each application its unique private virtual address space.

This virtual address space is divided into units known as pages, each typically 4 KB in size. These pages can be mapped to any location in physical memory, providing flexibility and optimizing memory utilization. The use of this virtual address space further adds a layer of abstraction between the application and the physical memory, enhancing the security and isolation of processes.

2.7 Swap Space

Swap space refers to a designated portion of the secondary storage utilized as virtual memory in a computer system[11]. This feature plays a crucial role in systems that run multiple applications simultaneously. When memory resources are strained, swap space comes into play, relocating inactive parts of the RAM to secondary storage. This action frees up space in primary memory for other processes, enabling smoother operation and preventing a potential system crash.

In the case of Linux, swap space implementation aligns with a demand paging system. This means that memory is allocated only when required. The swap space in Linux can be a swap partition, which is a distinct area within the secondary storage, or it can take the form of a swap file, which is a standard file that can be expanded or truncated based on need. The usage of swap partitions and files is transparent to the user.

The Linux kernel employs a Least Recently Used (LRU) algorithm to determine which memory pages should be moved to swap space. This algorithm effectively prioritizes pages based on their usage, transferring those that have not been recently used to swap space.

Swap space also plays a significant role in system hibernation. Before the system enters hibernation, the content of RAM is stored in the swap space, where it remains persistent even without power. When the system is resumed, the memory content is read back from swap space, restoring the system to its pre-hibernation state[12].

However, the use of swap space can impact system performance. Since secondary storage devices are usually slower than primary memory, heavy reliance on swap space can cause significant system slowdowns. To mitigate this, Linux allows for the adjustment of “swappiness”, a parameter that controls the system’s propensity to swap memory pages. Adjusting this setting can balance the use of swap space to maintain system performance while still preserving the benefits of virtual memory management.

2.8 Page Faults

Page faults are instances in which a process attempts to access a page that is not currently available in primary memory. This situation triggers the operating system to swap the necessary page from secondary storage into primary memory. These are significant events in memory management, as they determine how efficiently an operating system utilizes its resources.

Page faults can be broadly categorized into two types: minor and major. Minor page faults occur when the desired page resides in memory but isn't linked to the process that requires it. On the other hand, a major page fault takes place when the page has to be loaded from secondary storage, a process that typically takes more time and resources[3].

To minimize the occurrence of page faults, memory management algorithms such as the aforementioned Least Recently Used (LRU) and the more straightforward clock algorithm are often employed. These algorithms effectively manage the order and priority of memory pages, helping to ensure that frequently used pages are readily available in primary memory.

Handling page faults involves certain techniques to ensure smooth operation. One such technique is prefetching, which anticipates future page requests and proactively loads these pages into memory. Another approach involves page compression, where inactive pages are compressed and stored in memory preemptively[13]. This reduces the likelihood of major page faults by conserving memory space, allowing more pages to reside in primary memory.

In general, handling page faults is a task delegated to the kernel. This critical balance between resource availability and system performance is part of the kernel's memory management duties, ensuring that processes can access the pages they require while maintaining efficient use of system memory.

2.9 `mmap`

`mmap` is a versatile UNIX system call, used for mapping files or devices into memory, enabling a variety of core tasks like shared memory, file I/O, and fine-grained memory allocation. Due to its powerful nature, it is commonly harnessed in applications like databases.

One standout feature of `mmap` is its ability to create what is essentially a direct memory mapping between a file and a region of memory[14]. This connection means that read operations performed on the mapped memory region directly correspond to reading the file and vice versa, enhancing efficiency by reducing the overhead as the necessity for context switches (compared to i.e. the `read` or `write` system calls) diminishes.

The key advantage that `mmap` provides is the capacity to facilitate zero-copy operations. In practical terms, this signifies data can be accessed directly as if it were positioned in memory, eliminating the need to copy it from the disk first. This direct memory access saves time and reduces processing requirements, offering substantial performance improvements.

`mmap` is also proficient in sharing memory between processes without having to pass through the kernel with system calls[4]. With this feature, `mmap` can create shared memory spaces where multiple processes can read and write, enhancing interprocess communication and data transfer efficiency.

The potential speed improvement does however come with a notable drawback: It bypasses the file system cache, which can potentially result in stale data when multiple processes are reading and writing simultaneously. This bypass may lead to a scenario where one process modifies data in the `mmap` region, and another process that is not monitoring for changes might remain unaware and continue to work with outdated data.

2.10 `inotify`

The `inotify` is an event-driven notification system of the Linux kernel, designed to monitor the file system for different events, such as modifications and accesses, among others[15]. Its particularly useful because it can be configured to watch only write operations on certain files, i.e. only `write` operations. This level of control can offer considerable benefits in cases where there is a need to focus system resources on certain file system events, and not on others.

Naturally, `inotify` comes with some recognizable advantages. Significantly, it diminishes overhead and resource use when compared to polling strategies. Polling is an operation-heavy approach as it continuously checks the status of the file system, regardless of whether any changes have occurred. In contrast, `inotify` works in a more event-driven way, where it only takes action when a specific event actually occurs. This is usually more efficient, reducing overhead especially where there are infrequent changes to the file system.

Thanks to its efficiency and flexibility, `inotify` has found its utilization across many applications, especially in file synchronization services. In this usecase, the ability to instantly notify the system of file changes aids in instant synchronization of files, demonstrating how critical its role can be in real-time or near real-time systems that are dependent on keeping data up-to-date.

However, as is the case with many system calls, there is a limit to its scalability. `inotify` is constrained by a limit on how many watches can be established. This limitation can pose challenges in intricate systems where there is a high quantity of files or directories to watch for, and might warrant additional management or fallback to heavier polling mechanisms for some parts of the system.

2.11 Linux Kernel Caching

Caching is a key feature of the Linux kernel that work to boost efficiency and performance. Within this framework, there are two broad categories: disk caching and file caching.

Disk caching in Linux is a strategic method that temporarily stores frequently accessed data in RAM. It is implemented through the page cache subsystem, and operates under the assumption that data situated near data that has already been accessed will be needed soon. By retaining data close to the CPU where it may be swiftly accessed without costly disk reads can greatly reduce overall access time. The data within the cache is also managed using the LRU algorithm, which prunes the least recently used items first when space is needed.

Linux also caches file system metadata in specialized structures known as the `dentry` and `inode` caches. This metadata encompasses varied information such as file names, attributes, and locations. The key benefit of this is that it expedites the resolution of path names and file attributes, such as tracking when files were last changed for polling. Notably, file read/write operations are also channeled through the disk cache, further illustrating the intricate interconnectedness of disk and file caching mechanisms in the Linux Kernel.

While such caching mechanisms can improve performance, they also introduce complexities. One such complexity involves maintaining data consistency between the disk and cache through the process known as writebacks; aggressive writebacks, where data is copied back to disk frequently, can lead to reduced performance, while excessive delays may risk data loss if the system crashes before data has been saved.

Another complexity arises from the necessity to release cached data under memory pressure, known as cache eviction. This requires sophisticated algorithms, such as LRU, to ensure effective utilization of available cache space[3]. Prioritizing what to keep in cache when memory pressure builds does directly impact the overall system performance.

2.12 TCP, UDP and QUIC

TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and QUIC (Quick UDP Internet Connections) are three key communication protocols utilized in the internet today.

TCP has long been the reliable backbone for internet communication due to its connection-oriented nature [16]. It ensures the guaranteed delivery of data packets and their correct order, rendering it a highly dependable means for data transmission. Significantly, TCP incorporates error checking, allowing the detection and subsequent retransmission of lost packets. TCP also includes a congestion control mechanism to manage data transmission seamlessly during high traffic. Due to to these features and it's long legacy, TCP is widely used to power the majority of the web where reliable, ordered, and error-checked data transmission is required.

UDP is a connectionless protocol that does not make the same guarantees about the reliability or ordered delivery of data packets [17]. This lends UDP a speed advantage over TCP, resulting in less communication overhead. Although it lacks TCP's robustness in handling errors and maintaining data order, UDP finds use in applications where speed and latency take precedence over reliability. This includes online gaming, video calls, and other real-time communication modes where quick data transmission is crucial even if temporary packet loss occurs.

QUIC, a modern UDP-base transport layer protocol, was originally created by Google and standardized by the IETF in 2021[18]. It aspires to combine the best qualities of TCP and UDP [19]. Unlike raw UDP, QUIC ensures the reliability of data transmission and guarantees the ordered delivery of data packets similarly to TCP, while intending to keep UDP's speed advantages. One of QUIC's standout features is its ability to reduce connection establishment times, which effectively lowers initial latency. It achieves this by merging the typically separate connection and security handshakes, reducing the time taken for a connection to be established. Additionally, QUIC is designed to prevent the issue of "head-of-line blocking", allowing for the independent delivery of separate data streams. This means it can handle the delivery of separate data streams without one stream blocking another, resulting in smoother and more efficient transmission, a feature which is especially important for applications with lots of concurrent transmissions.

2.13 Delta Synchronization

Delta synchronization is a technique that allows for efficient synchronization of files between hosts, aiming to transfer only those parts of the file that have undergone changes instead of the entire file in order to reduce network and I/O overhead. Perhaps the most recognized tool employing this method of synchronization is `rsync`, an open-source data synchronization utility in Unix-like operating systems[20].

TODO: Add sequence diagram of the delta sync protocol from https://blog.acolyer.org/2018/03/02/towards-web-based-delta-synchronization-for-cloud-storage-systems/

While there are many applications of such an algorithm, it typically starts on file block division, dissecting the file on the destination side into fixed-size blocks. For each of these blocks, a quick albeit weak checksum calculation is performed, and these checksums are transferred to the source system.

The source initiates the same checksum calculation process. These checksums are then compared to those received from the destination (matching block identification). The outcome of this comparison allows the source to detect the blocks which have transformed since the last synchronization.

Once the altered blocks are identified, the source proceeds to send the offset of each block alongside the data of the changed block to the destination. Upon receiving a block, the destination writes it to the specific offset in the file. This process results in the reconstruction of the file in accordance with the modifications undertaken at the source, after which the next synchronization cycle can start.

2.14 File Systems In Userspace (FUSE)

File Systems in Userspace (FUSE) is a software interface that enables the creation of custom file systems in the userspace, as opposed to developing them as kernel modules. This reduces the need for the low-level kernel development skills that are usually associated with creating new file systems.

The FUSE APIs are available on various platforms; though mostly deployed on Linux, it can also be found on macOS and FreeBSD. In FUSE, a userspace program registers itself with the FUSE kernel module and provides callbacks for the file system operations. A simple read-only FUSE can for example implement the following callbacks:

The `getattr` function is responsible for getting the attributes of a file. For a real file system, this would include things like the file's size, its permissions, when it was last accessed or modified, and so forth:

```c
static int example_getattr(const char *path, struct stat *stbuf,
                           struct fuse_file_info *fi);
```

The `readdir` function is used when a process wants to list the files in a directory. It's responsible for filling in the entries for that directory:

```c
static int example_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                           off_t offset, struct fuse_file_info *fi,
                           enum fuse_readdir_flags flags);
```

The `open` function is called when a process opens a file. It's responsible for checking that the operation is permitted (i.e. the file exists and the process has the necessary permissions), and for doing any necessary setup:

```c
static int example_open(const char *path, struct fuse_file_info *fi);
```

Finally, the `read` function is used when a process wants to read data from a file. It's responsible for copying the requested data into the provided buffer:

```c
static int example_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi);
```

These callbacks would then be added to the FUSE operations struct and passed to `fuse_main`, which takes care of registering the operations with the FUSE kernel module and mounts the FUSE to a directory. Similarly to this, callbacks for handling writes etc. can be provided to the operation struct for a read-write capable FUSE[21].

When a user then performs a file system operation on a mounted FUSE file system, the kernel module sends a request for executing that operation to the userspace program. This is followed by the userspace program returning a response, which the FUSE kernel module conveys back to the user. As such, FUSE circumvents the complexity of coding the file system implementation directly in the kernel. This approach enhances safety, preventing entire kernel crashes due to errors within the implementation being limited to user instead of kernel space.

TODO: Add graphic from https://en.wikipedia.org/wiki/Filesystem_in_Userspace#/media/File:FUSE_structure.svg

Another benefit of a file system implemented as a FUSE is its inherent portability. Unlike a file system created as a kernel module, its interaction with the FUSE module rather than the kernel itself creates a stronger contract between the two, and allows shipping the file system as a plain binary instead of a binary kernel module, which typically need to be built from source on the target machine unless they are vendored by a distribution. Despite these benefits of FUSE, there is a noticeable performance overhead associated with it. This is largely due to the context switching between the kernel and the userspace that occurs during its operation[22].

Today, FUSE is widely utilized to mount high-level external services as file systems. For instance, it can be used to mount remote AWS S3 buckets with `s3fs`[23] or to mount a remote system's disk via Secure Shell (SSH) with SSHFS [24].

2.15 Network Block Device (NBD)

Network Block Device (NBD) is a protocol for connecting to a remote Linux block device. It typically works by communicating between a user space-provided server and a Kernel-provided client. Though potentially deployable over Wide Area Networks (WAN), it is primarily designed for Local Area Networks (LAN) or localhost usage. The protocol is divided into two phases: the handshake and the transmission[25].

TODO: Add sequence diagram of the NBD protocol

The NBD protocol involves multiple participants, notably one or several clients, a server, and the concept of an export. It starts with a client establishing a connection with the server. The server reciprocates by delivering a greeting message highlighting various server flags. The client responds by transmitting its own flags along with the name of an export to use; a single NBD server can expose multiple devices.

After receiving this, the server sends the size of the export and other metadata. The client acknowledges this data, completing the handshake. Post handshake, the client and server exchange commands and replies. A command can correspond to any of the basic actions needed to access a block device, for instance read, write or flush. These commands might also contain data such as a chunk for writing, offsets, and lengths among other elements. Replies may contain error messages, success status, or data contingent on the reply type.

While powerful in many regards, NBD has some limitations. Its maximum message size is capped at 32 MB[26], and the maximum block or chunk size supported by the Kernel's NBD client is a mere 4KB[27]. Thus, it might not be the most optimal protocol for WAN usage, especially in scenarios with high latency.

NBD, being a protocol with a long legacy, comes with its own set of operational quirks such as multiple different handshake versions and legacy features. As a result, it is advisable to only implement the latest recommended versions and the foundational feature set when considering it NBD for a narrow usecase.

Despite the simplicity of the protocol, there are certain scenarios where NBD falls short. Compared to FUSE, it has limitations when dealing with backing devices that operate drastically different from random-access storage devices like a tape drive, since it lacks the ability to work with high-level abstractions such as files or directories. For example, it does not support shared access to the same file for multiple clients. However, this shortcoming can be considered as an advantage for narrow usecases like memory synchronization, given that it operates on a block level, where such features are not needed or implemented at a higher layer.

2.16 Virtual Machine Live Migration

Virtual machine live migration involves the shifting of a virtual machine, its state, and its connected devices from one host to another, with the objective to minimize disrupted service by minimizing downtime during data transfer processes.

Algorithms that intent to implement this usecase can be categorized into two broad types: pre-copy migration and post-copy migration.

2.16.1 Pre-Copy

The primary characteristic of pre-copy migration is its "run-while-copy" nature, meaning that the copying of data from the source to the destination occurs concurrently while the VM continues to operate. This method is also applicable in a generic migration context where an application or another data state is being updated.

In the case of a VM, the pre-copy migration procedure starts with transfering the initial state of VM's memory to the destination host. During this operation, if modifications occur to any chunks of data, they are flagged as "dirty". These modified or "dirty" chunks of data are then transferred to the destination until only a small number remain - an amount small enough to stay within the allowable maximum downtime criteria.

Following this, the VM is suspended at the source, enabling the synchronization of the remaining chunks of data to the destination without having to continue tracking dirty chunks. Once this synchronization process is completed, the VM is resumed at the destination host.

The pre-copy migration process is fairly robust, especially in instances where there might be network disruption during synchronization. This is because of fact that, at any given point during migration, the VM is readily available in full either at the source or the destination. A limitation to the approach however is that, if the VM or application alters too many chunks on the source during migration, it may not be possible to meet the maximum acceptable downtime criteria. Maximum permissible downtime is also inherently restricted by the available round-trip time (RTT)[28].

2.16.2 Post-Copy

Post-copy migration is an alternative live migration approach. While pre-copy migration operates by copying data before the VM halt, post-copy migration opts for another strategy: it immediately suspends the VM operation on the source and resumes it on the destination – all with only a minimal subset of the VM's data.

During this resumed operation, whenever the VM attempts to access a chunk of data not initially transferred during the move, a page fault arises. A page fault, in this context, is a type of interrupt generated when the VM tries to read or write a chunk that is not currently present on the destination. This triggers the system to retrieve the missing chunk from the source host, enabling the VM to continue its operations[28].

The main advantage of post-copy migration centers around the fact that it eliminates the necessity of re-transmitting chunks of "dirty" or changed data before hitting the maximum tolerable downtime. This process can thus decrease the necessary downtime and also reduces the amount of network traffic between source and destination.

However, this approach is also not without its drawbacks. Post-copy migration could potentially lead to extended migration times, as a consequence of its "fetch-on-demand" model for retrieving chunks. This model is highly sensitive to network latency and round-trip time (RTT). Unlike the pre-copy model, this also means that the VM is not available in full on either the source or the destination during migration, requiring potential recovery solutions if network connectivity is lost during the migration.

2.16.3 Workload Analysis

Recent studies have explored different strategies to determine the most suitable timing for virtual machine migration. Even though these mostly focus on virtual machines, the methodologies proposed could be adapted for use with various other applications or migration circumstances, too.

One method[29] proposed identifies cyclical workload patterns of VMs and leverages this knowledge to delay migration when it is beneficial. This is achieved by analyzing recurring patterns that may unnecessarily postpone VM migration, and then constructing a model of optimal cycles within which VMs can be migrated. In the context of VM migration, such cycles could for example be triggered by a large application's garbage collector that results in numerous changes to VM memory.

When migration is proposed, the system verifies whether it is in an optimal cycle for migration. If it is, the migration proceeds; if not, the migration is postponed until the next cycle. The proposed process employs a Bayesian classifier to distinguish between favorable and unfavorable cycles.

Compared to the popular alternative method which usually involves waiting for a significant amount of unchanged chunks to synchronize first, the proposed pattern recognition-based approach potentially offers substantial improvements. The study found that this method yielded an enhancement of up to 74% in terms of live migration time/downtime and a 43% reduction concerning the volume of data transferred over the network.

2.17 Streams and Pipelines

Streams and pipelines are fundamental constructs in computer science, enabling efficient, sequential processing of large datasets without the need for loading an entire dataset into memory. They form the backbone of modular and efficient data processing techniques, with each concept having its unique characteristics and use cases.

A stream represents a continuous sequence of data, serving as a connector between different points in a system. Streams can be either a source or a destination for data. Examples include files, network connections, and standard input/output devices and many others. The power of streams comes from their ability to process data as it becomes available; this aspect allows for minimization of memory consumption, making streams particularly impactful for scenarios involving long-running processes where data is streamed over extended periods of time[30].

Pipelines comprise a series of data processing stages, wherein the output of one stage directly serves as the input to the next. It's this chain of processing stages that forms a "pipeline". Often, these stages can run concurrently; this parallel execution can result in a significant performance improvement due to a higher degree of concurrency.

One of the classic examples of pipelines is the instruction pipeline in CPUs, where different stages of instruction execution - fetch, decode, execute, and writeback - are performed in parallel. This design increases the instruction throughput of the CPU, allowing it to process multiple instructions simultaneously at different stages of the pipeline.

Another familiar implementation is observed in UNIX pipes, a fundamental part of shells such as GNU Bash or POSIX `sh`. Here, the output of a command can be "piped" into another for further processing; for instance, the results from a `curl` command fetching data from an API could be piped into the `jq` tool for JSON manipulation[31].

2.18 gRPC

gRPC is an open-source, high-performance remote procedure call (RPC) framework developed by Google in 2015. It is recognized for its cross-platform compatibility, supporting a variety of languages including Go, Rust, JavaScript and more. gRPC is being maintained by the Cloud Native Computing Foundation (CNCF), which ensures vendor neutrality.

One of the notable features of the gRPC is its usage of HTTP/2 as the transport protocol. This allows it to exploit features of HTTP/2 such as header compression, which minimizes bandwidth usage, and request multiplexing, enabling multiple requests to be sent concurrently over a single connection. In addition to HTTP/2, gRPC utilizes Protocol Buffers (protobuf) as the Interface Definition Language (IDL) and wire format. Protobuf is a compact, high-performance, and language-neutral mechanism for data serialization. This makes it preferable over the more dynamic, but more verbose and slower JSON format often used in REST APIs.

One of the strengths of the gRPC framework is its support for various types of RPCs. Not only does it support unary RPCs where the client sends a single request to the server and receives a single response in return, mirroring the functionality of a traditional function call, but also server-streaming RPCs, wherein the client sends a request, and the server responds with a stream of messages. Conversely, in client-streaming RPCs, the client sends a stream of messages to a server in response to a request. It also supports bidirectional RPCs, wherein both client and server can send messages to each other.

What distinguishes gRPC is its pluggable structure that allows for added functionalities such as load balancing, tracing, health checking, and authentication, which make it a comprehensive solution for developing distributed systems[32].

2.19 Redis

Redis (Remote Dictionary Server) is an in-memory data structure store, primarily utilized as an ephemeral database, cache, and message broker introduced by Salvatore Sanfilippo in 2009. Compared to other key-value stores and NoSQL databases, Redis supports a multitude of data structures, including lists, sets, hashes, and bitmaps, making it a good choice for caching or storing data that does not fit well into a traditional SQL architecture[33].

One of the primary reasons for Redis's speed is its reliance on in-memory data storage rather than on disk, enabling very low-latency reads and writes. While the primary usecase of Redis is in in-memory operations, it also supports persistence by flushing data to disk. This feature broadens the use cases for Redis, allowing it to handle applications that require longer-term data storage in addition to a caching mechanism. In addition to it being mostly in-memory, Redis also supports quick concurrent reads/writes thanks to its non-blocking I/O model, making it a good choice for systems that require the store to be available to many workers or clients.

Redis also includes a publish-subscribe (pub-sub) system. This enables it to function as a message broker, where messages are published to channels and delivered to all the subscribers interested in those channels. This makes it a particularly compelling choice for systems that require both caching and a memory broker, such as queue systems[34].

2.20 S3 and Minio

S3 is a scalable object storage service, especially designed for large-scale applications with frequent reads and writes. It is one of the prominent services offered by Amazon Web Services. S3's design allows for global distribution, which means the data can be stored across multiple geographically diverse servers. This permits fast access times from virtually any location on the globe, crucial for globally distributed services or applications with users spread across different continents.

S3 offers a variety of storage classes for to different needs, i.e. for whether the requirement is for frequent data access, infrequent data retrieval, or long-term archival. This ensures that it can meet a wide array of demands through the same API. S3 also comes equipped with comprehensive security features, including authentication and authorization mechanisms.

Communication with S3 is done through a HTTP API. Users and applications can interact with the stored data - including files and folders - via this API.[35].

Minio is an open-source storage server that is compatible Amazon S3's API. Due to it being written in the Go programming language, Minio is very lightweight and even ships as single static binary. Unlike with AWS S3, which is only offered as a service, Minio's open-source nature means that users have the ability to view, modify, and distribute Minio's source code, allowing community-driven development and innovation.

A critical distinction of Minio is its suitability for on-premises hosting, making it a good fit for organizations with specific security regulations, those preferring to maintain direct control over their data and developers prefering to work on the local system. It also supports horizontal scalability, designed to distribute large quantities of data across multiple nodes, meaning that it can be used in large-scale deployments similarly to AWS S3[36].

2.21 Cassandra and ScylllaDB

Apache Cassandra is a wide-column NoSQL database tailored for large-scale, distributed data management tasks. It blends the distributed nature of Amazon's Dynamo model with the structure of Google's Bigtable model, leading to a highly available database system. It is known for its scalability, designed to handle vast amounts of data spread across numerous servers. Unique to Cassandra is the absence of a single point of failure, thus ensuring continuous availability and robustness, which is critical for systems requiring high uptime.

Cassandra's consistency model is tunable according to needs, ranging from eventual to strong consistency. It distinguishes itself by not employing master nodes due to its usage of a peer-to-peer protocol and a distributed hash ring design. These design choices eradicate the bottleneck and failure risks associated with master nodes[37].

Despite these robust capabilities, Cassandra does come with certain limitations. Under heavy load, it experiences high latency that can negatively affect system performance. Besides this, it also demands complex configuration and fine-tuning to peform optimally.

In response to the perceived shortcomings of Cassandra, ScyllaDB was launched in 2015. It shares design principles with Cassandra, such as compatibility with Cassandra's API and data model, but has architectural differences intended to overcome Cassandra's limitations. It's primarily written in C++, contrary to Cassandra's Java-based code. This contributes to ScyllaDB's shared-nothing architecture, a design that aims to minimize contention and enhance performance.

ScyllaDB was particularly engineered to address one shortcoming of Cassandra - issues around latency, specifically the 99th percentile latency that impacts system reliability and predictability. ScyllaDB's design improvements and performance gains over Cassandra have been endorsed by various benchmarking studies[38].

TODO: Add graph of the Cassandra vs. ScyllaDB benchmark from the benchmarking study

3 Planning

3.1 Pull-Based Synchronization With `userfaultfd`

`userfaultfd` allows the implementation of a post-copy migration scenario. In this setup, a memory region is created on the destination host. When the migrated application starts to read from this remote region after it was resumed, it triggers a page fault, which we want to resolve by fetching the relevant offset from the remote.

Typically, page faults are resolved by the kernel. While this makes sense for use cases where they can be resolved by loading a local ressource into memory, here we want to handle the page faults using a user space program instead. Traditionally, this was possible by registering a signal handler for the `SIGSEGV` handler, and then responding to fault from the program. This however is a fairly complicated and inefficient process. Instead, we can now use the `userfaultfd` system to register a page fault handler directly without having to go through a signal first.

With `userfaultfd`, we first register the memory region that we want to handle page faults in and start a handler in user space that fetches the missing offsets from the source host in-demand whenever a page fault occurs. This handler is connected to the registered region's `userfaultfd` API through a file descriptor. To enable sharing the file descriptor between processes, a UNIX socket can be used.

3.2 Push-Based Synchronization With mmap and Hashing

As mentioned before, `mmap` allows mapping a memory region to a file. Similarly to how we used a region registered with `userfaultfd` before to store the state or application that is being migrated, we can use this region to do the same. Because the region is linked to a file, when writes happen to the region, they will also be written to the corresponding file. If we're able to detect these writes and copy the changes to the destination host, we can use this setup to implement a pre-copy migration system.

While writes done to a `mmap`ed region are eventually being written back to the underlying file, this is not the case immediately, since the kernel still uses caching on an `mmap`ed region in order to speed up reads/writes. As a workaround, we can use the `msync` syscall, which works similarly to the `sync` syscall by flushing any remaining changes from the cache to the backing file.

In order to actually detect the changes to the underlying file, an obvious solution might be to use `inotify`. This however isn't possible for `mmap`ed files, as the file corresponds to a memory region, and traditional `write` etc. evens are not emitted. Instead of using `inotify` or a similar event-based system to track changes, we can instead use a polling system. This has drawbacks - namely latency and computation load - that were attempted to be worked around in the following implementation, but are inherent to this approach.

3.3 Push-Pull Synchronization with FUSE

Using a file system in user space (FUSE) can serve as the basis for implementing either a pre- or a post-copy live migration system. Similarly to the file-based pre-copy approach, we can use `mmap` to map the migrated resource's memory region to a file. Instead of storing this file on the system's default filesystem however, a custom file system is implemented, which allows dropping the expensive polling system. Since a custom file system allows us to catch reads (for a post-copy migration scenario, were reads would be responded to by fetching from the remote), writes (for a pre-copy scenario, where writes would be forwarded to the destination) and other operations by the kernel, we no longer need to use `inotify`.

While implementing such a custom file system in the kernel is possible, it is a complex task that requires writing a custom kernel module, using a supported language by the kernel (mostly C or a limited subset of Rust), and in general having significant knowledge of kernel internals. Furthermore, since networking would be required to resolve reads/forward writes from/to the source/destination host, a job that would usually be done by user space applications, a user space component would probably also need to be developed in order to support this part of the synchronization system. Instead of implementing it in the kernel, we can use the FUSE API. This makes it possible to write the entire file system in user space, can significantly reduce the complexity of this approach.

3.4 Mounts with NBD

Another `mmap`-based approach for both pre- and post-copy migration is to `mmap` a block device instead of a file. This block device can be provided through a variety of APIs, for example NBD.

By providing a NBD device through the kernel's NBD client, we can connect the device to a remote NBD server, which in turn hosts the migratable resource as a memory region. Any reads/writes from/to the `mmap`ed memory region are resolved by the NBD device, which forwards it to the client, which then resolves them using the remote server; as such, this approach is less so a synchronization (as the memory region is never actually copied to the destination hist), but rather a mount of a remote memory region over the NBD protocol.

From an initial overview, the biggest benefit of `mmap`ing such a block device instead of a file on a custom file system is the reduced complexity. For the narrow usecase of memory synchronization, not all of the features provided by a full file system are be required, which means that the implementation of a NBD server and client, as well as the accompanying protocols, is significantly less complex and can also reduce the overhead of the system as a whole.

3.5 Push-Pull Synchronization with Mounts

3.5.1 Overview

This approach also leverages `mmap` and NBD to handle reads and writes to the migratable ressource's memory region, similar to the prior approaches, but differs from mounts with NBD in a few significant ways.

Usually, the NBD server and client don't run on the same system, but are instead separated over a network. This network commonly is LAN, and the NBD protocol was designed to access a remote hard drive in this network. As a result of the protocol being designed for this low-latency, high-throughput type of network, there are a few limitations of the NBD protocol when it is being used in a WAN that can not guarantee the same.

While most wire security issues with the protocol can be worked around by simply using TLS, the big issue of it's latency sensitivity remains. Usually, individual blocks would only be fetched as they are being accessed, resulting in a ready latency per block that is at least the RTT. In order to work around this issue, instead of directly connecting a NBD client to a remote NBD server, a layer of indirection (called "Mount") is created. This component consists of both a client and a server, both of which are running on the local system instead of being split into a separate remote and local component.

By combining the NBD server and client into this reusable uit, we can connect the server to a new backend component with a protocol which is better suited for WAN usage than NBD. This also allows the implementation of smart, asynchronous background push/pull strategies instead of simpliy directly writing to/from the network (called "Managed Mounts"). The simplest form of the mount API is the direct mount API; it simply swaps out NBD for a transport-independent RPC framework, but does not do additional optimizations. It has two simple actors: The client and the server. Only unidirectional RPCs from the client to the server are required for this to work, and the required backend service's interface is simple:

```go
type BackendRemote struct {
    ReadAt  func(context context.Context, length int, off int64) (r ReadAtResponse, err error)
    WriteAt func(context context.Context, p []byte, off int64) (n int, err error)
    Size    func(context context.Context) (int64, error)
    Sync    func(context context.Context) error
}
```

The protocol is stateless, as there is only a simple remote reader and writer interface; there are no distinct protocol phases, either.

TODO: Add protocol sequence diagram TODO: Add state machine diagram

3.5.2 Chunking

And additional issue that was mentioned before that this approach can approve upon is better chunking support. While it is possible to specify the NBD protocol's chunk size by configuring the NBD client and server, this is limited to only 4KB in the case of Linux's implementation. If the RTT between the backend and the NBD server however is large, it might be preferable to use a much larger chunk size;

this used to not be possible by using NBD directly, but thanks to this layer of indirection it can be implemented.

Similarly to the Linux kernel's NBD client, backends themselves might also have constraints that prevent them from working without a specific chunk size, or otherwise require aligned reads. This is for example the case for tape drives, where reads and writes must occur with a fixed block size and on aligned offsets; furthermore, these linear storage devices work best if chunks are multiple MBs instead KBs.

It is possible to do this chunking in two places: On the mount API's side (meaning the NBD server), or on the (potentially remote) backend's side. While this will be discussed further in the results section, chunking on the backend's side is usually preferred as doing it client-side can significantly increase latency due to a read being required if a non-aligned write occurs, esp. in the case of a WAN deployment with high RTT.

But even if the backend does not require any kind of chunking to be accessed - i.e. if it is a remote file - it might still make sense to limit the maximum supported message size between the NBD server and the backend, simply to prevent DoS attacks that would require the backend to allocate large chunks of memory, were such a limit provided by a chunking system not in place.

3.5.3 Background Pull and Push

A pre-copy migration system for the managed API is realized in the form of pre-emptive pulls that run asynchronously in the background. In order to optimize for sequential locality, a pull priority heuristic was introduced; this is used to determine the order in which chunks should be pulled. Many applications and other migratable resources commonly access certain parts of their memory first, so if a ressources should be accessible locally as quickly as possible (so that reads go to the local cache filled by the pre-emptive pulls, instead of having to wait at least one RTT to fetch it from the remote), knowing this access pattern and fetching these sections first can improve latency and throughput signficantly.

And example of this can be data that consists of one or multiple headers followed by raw data. If this structure is known, rather than fetching everything linearly in the background, the headers can be fetched first in order to allow for i.e. metadata to be displayed before the rest of the data has been fetched. Similarly so, if a file system is being synchronized, and the superblocks of a file system are being stored in a known pattern or known fixed locations, these can be pulled first, significantly speeding up operations such as directory listings that don't require the actual inode's data to be available.

Post-copy migration conversly is implemented using asynchronous background push. This push system is started in parallel with the pull system. It keeps track of which chunks were written to, deduplicates remote writes, and periodically writes back these dirty chunks to the remote backend. This can significantly improve write performance compared to forwarding writes directly to the remote by being able to catch multiple writes without having to block for at least the RTT until the remote write has finished before continuing to the next write.

For the managed mount API, the pre- and post-copy live migration paradigms are combined to form a hybrid solution. Due to reasons elaborated on in more detail in the discussion section, the managed mount API however is primarily intended for efficiently reading from a remote resource and synching back changes eventually, rather than migrating a resource between two hosts. For the migration usecase, the migration API, which will be introduced in the following section, provides a better solution by building on similar concepts as the managed mounts API.

3.6 Pull-Based Synchronization with Migrations

3.6.1 Overview

Similarly to the managed mount API, this migration API again tracks changes to the memory of the migratable resource using NBD. As mentioned before however, the managed mount API is not optimized for the migration usecase, but rather for efficiently accessing a remote resource. For live migration, one metric is very important: maximum acceptable downtime. This refers to the time that a application, VM etc. must be suspended or otherwise prevented from writing to or reading from the resource that is being synchronized; the higher this value is, the more noticable the downtime becomes.

To improve on this the pull-based migration API, the migration process is split into two distinct phases. This is required due the constraint mentioned earlier; the mount API does not allow for safe concurrent access of a remote resource by two readers or writers at the same time. This poses a signficant problem for the migration scenario, as the app that is writing to the source device would need to be suspended before the transfer could even begin, as starting the destination node would already violate the single-reader, single-writer constraint of the mount API. This adds significant latency, and is complicated further by the backend for the managed mount API not exposing a block itself but rather just serving as a remote that can be mounted. The migration API on the other hand doesn't have this hierarchical system; both the source and destination are peers that expose block devices on either end.

3.6.2 Migration Protocol and Critical Phases

The migration protocol that allows for this defines two new actors: The seeder and the leecher. A seeder represents a resource that can be migrated from or a host that exposes a migrabtable resource, while the leecher represents a client that intents to migrate a resource to itself. The protocol starts by running an application with the application's state on the region `mmap`ed to the seeder's block device, similarly to the managed mount API. Once a leecher connects to the seeder, the seeder starts tracking any writes to it's mount, effectively keeping a list of dirty chunks. Once tracking has started, the leecher starts pulling chunks from the seeder to it's local cache. Once it has received a satisfactory level of locally available chunks, it asks the seeder to finalize. This then causes the seeder to suspend the app accessing the memory region on it's block device, `msync`/flushes the it, and returns a list of chunks that were changed between the point where it started tracking and the flush has occured. Upon receiving this list, the leecher marks these chunks are remotes, immediately resumes the application (which is now accessing the leecher's block device), and queues the dirty chunks to be pulled in the background.

TODO: Add protocol sequence diagram TODO: Add state machine diagram

By splitting the migration into these two distinct phases, the overhead of having to start the deivce can be skipped and additional app initialization that doesn't depend on the app's state (i.e. memory allocation, connecting to databases, loading models etc.) can be performed before the application needs to be suspended. This combines both the pre-copy algorithm (by pulling the chunks from the seeder ahead of time) and the post-copy algorithm (by resolving dirtyc chunsk from the seeder after the VM has been migrated) into one coherent protocol. As will be discussed further in the results section, the maximum tolerable downtime can be drastically reduced, and dirty chunks don't need to be re-transmitted multiple times. Effectively, it allows dropping this downtime to the time it takes to `msync` the seeder's app state, the RTT and, if they are being accessed immediately, how long it takes to fetch the chunks that were written in between the start of it tracking and finalizing. The migration API can use the same preemptive pull system as the managed mount API and benefit from it's optimizations, but does not use the background push system.

An interesting question to ask with this two-step migration API is when to start the finalization step. The finalization phase in the protocol is critical, and it is hard or impossible to recover from depending on the specific implementation. While the synchronization itself could be safely recovered from by simply calling `Finalize` multiple times to restart it. But since `Finalize` needs to return a list of dirty chunks, it requires the app on the seeder to be suspended before `Finalize` can return, an operation that might not be idempotent.

4 Implementation

4.1 Userfaults in Go with `userfaultfd`

4.1.1 Registration and Handlers

By listening to page faults, we can know when a process wants to access a specific offset of memory that is not yet available. As mentioned before, we can use this event to then fetch this chunk of memory from the remote, mapping it to the offset on which the page fault occured, thus effectively only fetching data when it is required. Instead of registering signal handlers, we can use the `userfaultfd` system introduced with Linux 4.3[39] to handle these faults in userspace in a more idiomatic way.

In the Go implementation created for this thesis, `userfaultfd-go`, `userfaultfd` works by first creating a region of memory, e.g. by using `mmap`, which is then registered with the `userfaultfd` API:

```go
// Creating the `userfaultfd` API
uffd, _, errno := syscall.Syscall(constants.NR_userfaultfd, 0, 0, 0)

uffdioAPI := constants.NewUffdioAPI(
    constants.UFFD_API,
    0,
)
// ...

// Allocating the region
l := int(math.Ceil(float64(length)/float64(pagesize)) * float64(pagesize))
b, err := syscall.Mmap(
    -1,
    0,
    l,
    syscall.PROT_READ|syscall.PROT_WRITE,
    syscall.MAP_PRIVATE|syscall.MAP_ANONYMOUS,
)
// ...

// Registering the region
uffdioRegister := constants.NewUffdioRegister(
    constants.CULong(start),
    constants.CULong(l),
    constants.UFFDIO_REGISTER_MODE_MISSING,
)
// ...

syscall.Syscall(
  syscall.SYS_IOCTL,
  uffd,
  constants.UFFDIO_REGISTER,
  uintptr(unsafe.Pointer(&uffdioRegister))
)
```

This is abstracted into a single `Register(length int)([]byte, UFFD, uintptr, error)` function. Once this region has been registered, the `userfaultfd` API's file descriptor and the offset is passed over a UNIX socket:

```
syscall.Sendmsg(int(f.Fd()), nil, syscall.UnixRights(b...), nil, 0)
```

Where it can then be received by the handler:

```
buf := make([]byte, syscall.CmsgSpace(num*4)) // See https://github.com/ftrvxmtrx/fd/blob/master/fd.go#L51
syscall.Recvmsg(int(f.Fd()), nil, buf, 0)
// ..
msgs, err := syscall.ParseSocketControlMessage(buf)
```

The handler itself receives the address that has triggered the page fault by polling the transferred file descriptor, which is then responded to by fetching the relevant chunk from a provided reader and sending it to the faulting memory region over the same socket:

```
// Receiving the fage fault address
unix.Poll(
    []unix.PollFd{{
        Fd:     int32(uffd),
        Events: unix.POLLIN,
    }},
    -1,
)
// ...
pagefault := (*(*constants.UffdPagefault)(unsafe.Pointer(&arg[0])))
addr := constants.GetPagefaultAddress(&pagefault)

// Fetching the missing chunk from the provided backend
p := make([]byte, pagesize)
n, err := src.ReadAt(p, int64(uintptr(addr)-start))

// Sending the missing chunk to the faulting memory region's `userfaultfd` API:
cpy := constants.NewUffdioCopy(
    p,
    addr&^constants.CULong(pagesize-1),
    constants.CULong(pagesize),
    0,
    0,
)
```

```go

syscall.Syscall(
    syscall.SYS_IOCTL,
    uintptr(uffd),
    constants.UFFDIO_COPY,
    uintptr(unsafe.Pointer(&cpy)),
)
```

Similarly to the registration API, this is also wrapped into a reusable `func Handle(uffd UFFD, start uintptr, src io.ReaderAt)error` function.

4.1.2 `userfaultfd` Backends

Thanks to `userfaultfd` being mostly useful for post-copy migration, the backend can be simplifed to a simple pull-only reader interface (`ReadAt(p []byte, off int64)(n int, err error)`). This means that almost any `io.ReaderAt` can be used to provide chunks to a `userfaultfd`-registered memory region, and access to this reader is guaranteed to be aligned to system's page size, which is typically 4KB. By having this simple backend interface, and thus only requiring read-only access, it is possible to implement the migration backend in many different ways. A simple backend can for example return a pattern to the memory region:

```go
func (a abcReader) ReadAt(p []byte, off int64) (n int, err error) {
    n = copy(p, bytes.Repeat([]byte{'A' + byte(off%20)}, len(p)))

    return n, nil
}
```

In Go specifically, many objects can be exposed as an `io.ReaderAt`, including a file. This makes it possible to simply pass in any file as a backend, essentially mimicking a call to `mmap` with `MAP_SHARED`:

```go
f, err := os.OpenFile(*file, os.O_RDONLY, os.ModePerm)
// ...

b, uffd, start, err := mapper.Register(int(s.Size()))

mapper.Handle(uffd, start, f)
```

Similarly so, a remote file, i.e. one that is being stored in S3, can be used as a `userfaultfd` backend as well; here, HTTP range requests allow for fetching only the chunks that are being required by the application accessing the registered memory region, effectively making it possible to map a remote S3 object into memory:

```go
mc, err := minio.New(*s3Endpoint, /* ... */)

```

```
f, err := mc.GetObject(ctx, *s3BucketName, *s3ObjectName, minio.
    GetObjectOptions{})
// ...

b, uffd, start, err := mapper.Register(int(s.Size()))

mapper.Handle(uffd, start, f)
```

4.2 File-Based Synchronization

4.2.1 Caching Restrictions

As mentioned earlier, this approach uses `mmap` to map a memory region to a file. By default however, `mmap` doesn't write back changes to memory; instead, it simply makes the backing file available as a memory region, keeping changes to the region in memory, no matter whether the file was opened as read-only or read-writable. To work around this, Linux provides the `MAP_SHARED` flag; this tells the kernel to eventually write back changes to the memory region to the corresponding regions of the backing file.

Linux caches reads to the backing file similarly to how it does if `read` etc. are being used, meaning that only the first page fault would be responded to by reading from disk; this means that any future changes to the backing file would not be represented in the `mmap`ed region, similarly to how `userfaultfd` handles it. The same applies to writes, meaning that in the same way that files need to be `sync`ed in order for them to be flushed to disk, `mmap`ed regions need to be `msync`ed in order to flush changes to the backing file. This is particularly important for a memory usecase, since reading from the backing file without flushing first would result in the synchronization of potentially stale data, and is different to how traditional file synchronization can handle this usecase, where the Linux file cache would respond with the changes if the file is read from disk even if `sync` was not called beforehand. For file I/O, it is possible to skip the kernel cache and read/write directly from/to the disk by passing the `O_DIRECT` flag to `open`, but this flag is ignored by `mmap`.

4.2.2 Detecting File Changes

In order to actually watch for changes, at first glance, the obvious choice would be to use `inotify`, which would allow the registration of `write` or `sync` even handlers to catch writes to the memory region by registering them on the backing file. As mentioned earlier however, Linux doesn't emit these events on `mmap`ed files, so an alternative must be used; the best option here is to instead poll for either attribute changes (i.e. the "Last Modified" attribute of the backing file), or by continously hashing the file to check if it has changed. Hashing continously with this pollig method can have significant downsides, especially in a migration scenario, where it raises the guaranteed minimum latency by having to wait for at least the next polling cycle. Hashing the entire file is also a an I/O- and CPU-intensive process, because in order to compute the hash, the entire file needs to be read at some point. Within the context of the file-based synchronization approach however, it is the only option available.

To speed up the process of hashing, instead of hashing the entire file, we can instead hash individual chunks of the file, in effect implementing a delta synchronization algorithm. This can be implemented by opening the file multiple times, hashing individual offsets using each of the opened files, and aggregating the chunks that have been changed. When picking algorithms for this chunk-based hashing algorithm, two metrics are of relevance: the algorithm's throughput with which it can calculate hashes, and the prevalence of hash collisions, where two different inputs produce the same hashes, leading to a chunk change not being detected. Furthermore, if the underlying algorithm is CPU- and not I/O-bound, using multiple open files can increase throughput substantially by allowing for better concurrent processing. Not only does this decrease the time spent on each individual hashing iteration of the polling process, but dividing the file into smaller chunks that all have their own hashes to compare with the remote's hashes can also decrease the amount of network traffic that is required to sync the changes, since a small change in the backing file leads to the transfer of a smaller chunk.

4.2.3 Synchronization Protocol

The delta synchronization protocol for this approach is similar to the one used by `rsync`, but simplifie. It supports synchronizing multiple files at the same time by using the file names as IDs, and also supports a central forwarding hub instead of requiring peer-to-peer connectivity between all hosts, which also reduces network traffic since this central hub could also be used to forward one stream to all other peers instead of having to send it multiple times. The protocol defines three actors: The multiplexer, file advertiser and file receiver.

TODO: Add sequence diagram for the protocol

4.2.4 Multiplexer Hub

The multiplexer hub accepts mTLS connections from peers. When a peer connects, the client certificate is parsed to read the common name, which is then being used as the synchronization ID. The multiplexer spawns a goroutine to allow for more peers to connection. In the goroutine, it reads the type of the peer. If the type is `src-control`, it starts by reading a file name from the connection, and registers the connection as the one providing a file with this name, after which it broadcasts the file as now being available. For the `dst-control` peer type, it listens to the broadcasted files from the src-control peers, and relays and newly advertised and previously registered file names to the dst-control peers so that it can start receiving them:

```
case "src-control":
  // Decoding the file name
    file := ""
    utils.DecodeJSONFixedLength(conn, &file)
  // ...

    syncerSrcControlConns[file] = conn

    syncerSrcControlConnsBroadcaster.Broadcast(file)
  // ...
case "dst-control":
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
    // Subscription to send all future file names
        l := syncerSrcControlConnsBroadcaster.Listener(0)

        for file := range l.Ch() {
            utils.EncodeJSONFixedLength(conn, file)
      // ...
        }
    }()

  // Sending the previously known file names
    for file := range syncerSrcControlConns {
        utils.EncodeJSONFixedLength(conn, file)
    // ...
    }

    wg.Wait()
```

For the dst type, the multiplexer hub decodes a file name from the connection, looks for a corresponding src-control peer, and if it has found a matching one, it creates and sends a new ID for this connection to the src-control peer. After this, it waits until a src-control peer has connected to the hub with this ID as well as a new src-data peer by listening for broadcasts of src-data peer IDs. After this has occured, it spawns two new goroutines that copy data to and from this newly created synchronization connection and the connection of the dst peer, effectively relaying all packets between the two. For the src-data peer type, it decodes the ID for the peer, and broadcasts the ID, which allows the dst peer to continue operating:

```
case "dst":
  // Decoding the file name
  file := ""
  utils.DecodeJSONFixedLength(conn, &file)
```

```go

  // Finding the `src-control` peer
  controlConn, ok := syncerSrcControlConns[file]
  // ...

  // Sending the ID and waiting for a data connection
  id := uuid.NewString()
  utils.EncodeJSONFixedLength(controlConn, id)

  l := dataConnsBroadcaster.Listener(0)

  var dataConn net.Conn
  for candidate := range l.Ch() {
    if candidate.id == id {
        dataConn = candidate.conn

        l.Close()

        break
    }
  }

  // Copying data between both connections
  go func() {
    io.Copy(dataConn, conn)
    // ...
  }()

  go func() {
    io.Copy(conn, dataConn)
    // ...
  }()

case "src-data":
  // Receiving the ID
  id := ""
  utils.DecodeJSONFixedLength(conn, &id)

  // Broadcasting the ID
  dataConnsBroadcaster.Broadcast(connWithID{id, conn})
  // ...
```

4.2.5 File Advertisement

The file advertisement system connects to the multiplexer hub and registers itself a `src-control` peer, after which it sends the advertised file name. It starts a loop that handles `dst` peer types, which, as mentioned earlier, send an ID. Once such an ID is received, it spawns a new goroutine, which connects to the hub again and registers itself as a `src-data` peer, and sends the ID it has received earlier to allow connecting it to the matching dst peer:

```go
// ...
f, err := os.OpenFile(src, os.O_RDONLY, os.ModePerm)

utils.EncodeJSONFixedLength(dataConn, "src-data")

utils.EncodeJSONFixedLength(dataConn, id)
// ...
```

After this initial handshake is complete, the main synchronization loop is started, which initiates the file transmission to the dst peer through the multiplexer hub. In order to allow for termination, it checks if the syncStopped flag has been set by a context cancellation which case it returns:

```go
syncStopped := false
go func() {
    <-ctx.Done() // Context is supplied by the caller

    syncStopped = true
}()
```

If this is not the case, it waits for the specified polling interval, after which it restarts the transmission:

```go
done := false
for {
    if done {
        // ...
        return
    }

    if syncStopped {
    // Finish the synchronization
        SendFile(parallel, f, src, blocksize, dataConn, verbose)
    //  ...
        done = true
    }

  // Start the synchronization
    SendFile(parallel, f, src, blocksize, dataConn, verbose)

  // Wait until the next polling cycle starts
    time.Sleep(pollDuration)
}
```

4.2.6 File Receiver

The file receiver also connects to the multiplexer hub, this time registering itself as a `dst-control` peer. After it has received a file name from the multiplexer hub, it connects to the multiplexer hub again - this time registering itself as a `dst` peer, which creates leading directories, opens up the destination file and registers itself:

```go
// Connection and registration
syncerConn, err := d.DialContext(ctx, "tcp", syncerRaddr)
// ...
utils.EncodeJSONFixedLength(syncerConn, "dst-control")

for {
  file := ""
  utils.DecodeJSONFixedLength(syncerConn, &file)

  go func() {
    // Connection and registration
    dataConn, err := d.DialContext(ctx, "tcp", syncerRaddr)
    // ..
    utils.EncodeJSONFixedLength(dataConn, "dst")

    // Destination directory and file setup
    dst := getDstPath(file)
    os.MkdirAll(filepath.Dir(dst), os.ModePerm)

    f, err := os.OpenFile(dst, os.O_WRONLY|os.O_CREATE, os.ModePerm)
    // ...
  }()
}
```

The file name is now sent to the multiplexer again, causing it to look for a peer that advertises the requested file. If such a peer is found, it starts the file receiver process in a loop, exiting only once the file has been completely synced:

```go
// Sending the file name
utils.EncodeJSONFixedLength(dataConn, file)

for {
  // Receiving one set of delta changes
  ReceiveFile(parallel, f, dst, blocksize, dataConn, verbose)

  // Termination
  if once(file) {
    return
  }
}
```

4.2.7 File Transmission

This component does the actual transmission in each iteration of the delta synchronization algorithm. It receives the remote hashes from the multiplexer hub, calculates the matching local hashes and compares them, which it sends the hashes that don't match back to the file receiver via the multiplexer hub:

```go
// Receiving remote hashes
remoteHashes := []string{}
utils.DecodeJSONFixedLength(conn, &remoteHashes)
// ...

// Calculating the hashes
localHashes, cutoff, err := GetHashesForBlocks(parallel, path,
    blocksize)

// Comparing the hashes
blocksToSend := []int64{}
for i, localHash := range localHashes {
    j := int64(i)

    if len(remoteHashes) <= i {
        blocksToSend = append(blocksToSend, j)

        continue
    }

    if localHash != remoteHashes[i] {
        blocksToSend = append(blocksToSend, j)

        continue
    }
}

// Sending the non-matching hashes
utils.EncodeJSONFixedLength(conn, blocksToSend)
```

If the remote has sent less hashes than were calculated locally, it asks the remote to truncate it's file to the size of the local file that is being synchronized, after which it sends the updated data for the file in the order that the changed hashes were sent:

```go
// Local file is empty, truncate remote file to zero
if len(remoteHashes) > 0 && len(localHashes) <= 0 {
    utils.EncodeJSONFixedLength(conn, -1)

  return // ...
}

// Grow/shrink remote file
```

```
utils.EncodeJSONFixedLength(conn, cutoff)

// Read changed chunks from file and copy them to the remote connection
for i, blockToSend := range blocksToSend {
    backset := int64(0)
    if i == len(blocksToSend)-1 {
        backset = cutoff
    }

    b := make([]byte, blocksize-backset)
    file.ReadAt(b, blockToSend*(blocksize))

    m, err := conn.Write(b)
  // ...
}
```

4.2.8 Hash Calculation

The hash calculation implements the concurrent hashing of both the file transmitter and receiver. It uses a semaphore to limit the amount of concurrent access to the file that is being hashed, and a wait group to detect that the calculation has finished:

```
// The lock and semaphore
var wg sync.WaitGroup
wg.Add(int(blocks))

lock := semaphore.NewWeighted(parallel)

// ...

// Concurrent hash calculation
for i := int64(0); i < blocks; i++ {
    j := i

    go calculateHash(j)
}
wg.Wait()
```

Worker goroutines acquire a lock of this semaphore and calculate a CRC32 hash, which is a weak but fast hashing algorithm. For easier transmission, the hashes are hex-encoded and collected:

```
calculateHash := func(j int64) {
    _ = lock.Acquire(context.Background(), 1)
    // ...

    checkFile, err := os.Open(file)


```

```
  // Hash calculation of the specific chunk
    hash := crc32.NewIEEE()
    io.CopyN(hash, io.NewSectionReader(checkFile, j*(blocksize),
       blocksize), blocksize)
    // ...

  // Hash encoding
    hashes[j] = hex.EncodeToString(hash.Sum(nil))
  // ...
}
```

4.2.9 File Reception

This is the receiving component of one delta synchronization iteration. It starts by calculating hashes for the existing local copy of the file, which it then sends to the remote before it waits to receive the remote's hashes and potential truncation request:

```
// Local hash calculation
localHashes, _, err := GetHashesForBlocks(parallel, path, blocksize)

// Sending the hashes to the remote
utils.EncodeJSONFixedLength(conn, localHashes)

// Receiving the remote hashes and the truncation request
blocksToFetch := []int64{}
utils.DecodeJSONFixedLength(conn, &blocksToFetch)
// ...

cutoff := int64(0)
utils.DecodeJSONFixedLength(conn, &cutoff)
```

If the remote detected that the file needs to be cleared (by sending a negative cutoff value), the receiver truncates the file; similarly so, if it has detected that the file has grown or shrunk since the last synchronization cycle, it shortens or extends it, after which the chunks are read from the connection and written to the local file:

```
// Clearing the file
if cutoff == -1 {
    file.Truncate(0)

    return nil
}
/// ...

// Shrinking or growing the file
s, err := os.Stat(path)

```

```
newSize := (((blocksToFetch[len(blocksToFetch)-1] + 1) * blocksize) - cutoff)
diff := s.Size() - newSize

if diff > 0 {
    // If the file on the server got smaller, truncate the local file accordingly
    file.Truncate(newSize)
} else {
    // If the file on the server grew, grow the local file accordingly
    file.Seek(0, 2)

    io.CopyN(file, nopReader{}, -diff)
}

for i, blockToFetch := range blocksToFetch {
    backset := int64(0)
    if i == len(blocksToFetch)-1 {
        backset = cutoff
    }

  // Receiving the chunk
    b := make([]byte, blocksize-backset)
    io.ReadFull(conn, b)

  // Writing the chunk to the local file
    file.WriteAt(b, blockToFetch*(blocksize))
}
```

4.3 FUSE Implementation in Go

Implementing a FUSE in Go can be split into two separate tasks: Creating a backend for a file abstraction API and creating an adapter between this API and a FUSE library.

Developing a backend for a file system abstraction API such as `afero.Fs` instead of implementing it to work with FUSE bindings directly offers several advantages. This layer of indirection allows splitting the FUSE implementation from the actual `inode` structure of the system, which makes it unit testable[40]. This is a high priority due to the complexities and edge cases involved with creating a file system. A standard API also offers the ability to implement things such as caching by simply nesting multiple `afero.Fs` interfaces, and the required interface is rather minimal[41]:

```
type Fs interface {
    Create(name string) (File, error)
    Mkdir(name string, perm os.FileMode) error
    MkdirAll(path string, perm os.FileMode) error
    Open(name string) (File, error)
    OpenFile(name string, flag int, perm os.FileMode) (File, error)
```

```go
    Remove(name string) error
    RemoveAll(path string) error
    Rename(oldname, newname string) error
    Stat(name string) (os.FileInfo, error)
    Name() string
    Chmod(name string, mode os.FileMode) error
    Chown(name string, uid, gid int) error
    Chtimes(name string, atime time.Time, mtime time.Time) error
}
```

The STFS project[42] has shown that by using this abstraction layer, seemingly incompatible, non-linear backends can still be mapped to a file system. The project is backend by a tape drive, which is inherently append-only and optimized for linear access. Thanks to the inclusion of an on-disk index and various optimization methods, the resulting file system was still performant enough for standard use, while also supporting most of the features required by the average user such as symlinks, file updates and more.

By using a project like sile-fystem[43], it is also possible to use any `afero.Fs` filesystem as a FUSE backend; this can signficantly reduce the required implementation overhead, as it doesn't require writing a custom adapter:

```go
// Creating the file system
serve := filesystem.NewFileSystem(
  // ..
  afero.NewOsFs(),  // afero.Fs implementation here
  // ...
)
// ...

// Mounting the file system
fuse.Mount(viper.GetString(mountpoint), serve, cfg)
// ...
```

While the FUSE approach to synchronization is interesting, even with these available libraries the required overhead of implementing it (as shown by prior projects like STFS) as well as other factors that will be mentioned later led to this approach not being pursued further.

4.4 NBD with go-nbd

4.4.1 Overview

Due to a lack of existing, lean and maintained NBD libraries for Go, a custom pure Go NBD library was implemented. Most NBD libraries also only provide a server and no the client component, but both are needed for the NBD-based migration approach to work. By not having to rely on CGo or a pre-existing

NBD library like nbdkit, this custom library can also skip a significant amount of the overhead that is typically associated with C interoperability, particularly in the context of concurrency in Go with CGo [44].

4.4.2 Server

The NBD server is implemented completely in userspace, and there are no kernel components involved. The backend interface that is expected by the server is very simple and only requires four methods to be implemented; ReadAt, WriteAt, Size and Sync:

```go
type Backend interface {
    ReadAt(p []byte, off int64) (n int, err error)
    WriteAt(p []byte, off int64) (n int, err error)
    Size() (int64, error)
    Sync() error
}
```

The key difference between this backend design and the one used for userfaultfd-go is that they also support writes and other operations that would typically be expected for a complete block device, such as flushing data with Sync(). An example implementation of this backend is the file backend; since a file is conceptually similar to a block device, the overhead of creating the backend is minimal:

```go
func (b *FileBackend) ReadAt(p []byte, off int64) (n int, err error) {
    n, err = b.file.ReadAt(p, off)
}

func (b *FileBackend) WriteAt(p []byte, off int64) (n int, err error) {
    n, err = b.file.WriteAt(p, off)
}

func (b *FileBackend) Size() (int64, error) {
    stat, err := b.file.Stat()
    // ..
    return stat.Size(), nil
}

func (b *FileBackend) Sync() error {
    return b.file.Sync()
}
```

In order to serve such a backend, go-nbd exposes Handle function:

```go
func Handle(conn net.Conn, exports []Export, options *Options) error
```

By not depending on a specific transport layer and instead only depending on a generic net.Conn, it is possible to easily integrate go-nbd in existing client/server systems or to switch out the typical TCP transport layer with i.e. QUIC. By not requiring dial/accept semantics it is also possible to use a P2P communication layer for peer-to-peer NBD such as WebRTC with weron[45], which also provides the necessary net.Conn interface.

In addition to this net.Conn, options can be provided to the server; these include the ability to make the server read-only by blocking write operations, or to set the preferred block size. The actual backend is linked to the server through the concept of an export:

```go
type Export struct {
    Name        string
    Description string

    Backend backend.Backend
}
```

This allows a single server to expose multiple backends that are identified with a name and description, which can, in the memory synchronization scenario, be used to identify multiple shared memory regions.

To make the implementation of the NBD protocol easier, negotation and transmission phase headers and and other structured data is modelled using Go structs:

```go
// ...
type NegotiationOptionHeader struct {
    OptionMagic uint64
    ID          uint32
    Length      uint32
}

type NegotiationReplyHeader struct {
    ReplyMagic uint64
    ID         uint32
    Type       uint32
    Length     uint32
}
// ...
```

In order to keep the actual handshake as simple as possible, only the fixed newstyle handshake is implemented, which also makes the implementation compliant with the baseline specification as defined by the protocol[25]. The negotiation starts by the server sending the negotiation header to the NBD client and ignoring the client's flags:

```go
// Sending the negotiation header
binary.Write(conn, binary.BigEndian, protocol.NegotiationNewstyleHeader{
    OldstyleMagic:  protocol.NEGOTIATION_MAGIC_OLDSTYLE,
    OptionMagic:    protocol.NEGOTIATION_MAGIC_OPTION,
    HandshakeFlags: protocol.NEGOTIATION_HANDSHAKE_FLAG_FIXED_NEWSTYLE,
})

// Discard client flags (uint32)
_, err := io.CopyN(io.Discard, conn, 4)
// ...
```

The option negotiation phase is implemented using a simple loop, which either breaks on success or returns in the case of an error. For the Go implementation, it is possible to use the `binary` package to correctly encode and decode the NBD packets and then switching on the encoded option ID:

```
for {
    // Read the header
    var optionHeader protocol.NegotiationOptionHeader
    binary.Read(conn, binary.BigEndian, &optionHeader)

    //  Validate the packet
    if optionHeader.OptionMagic != protocol.NEGOTIATION_MAGIC_OPTION {
        return ErrInvalidMagic
    }

    // Handle the option by it's ID
    switch optionHeader.ID {
        // ...
    }
}
```

In this handshake, the `NEGOTIATION_ID_OPTION_INFO` and `NEGOTIATION_ID_OPTION_GO` options exchange information about the chosen export (i.e. block size, export size, name and description), and if `GO` is specified, immediately continue on to the transmission phase. If an export is not found, the server aborts the connection. In order to allow for enumeration of available exports, the `NEGOTIATION_ID_OPTION_LIST` allows for returning the list of exports to the client, and `NEGOTIATION_ID_OPTION_ABORT` allows aborting handshake, which can be necessary if i.e. the `NEGOTIATION_ID_OPTION_INFO` was chosen but the client can't handle the exposed export, i.e. due to it not supporting the advertised block size.

The actual transmission phase is implemented in a similar way, by reading headers in a loop, switchin on the message type and handling it accordingly. `TRANSMISSION_TYPE_REQUEST_READ` forwards a read request to the selected export's backend and sends the relevant chunk to the client, `TRANSMISSION_TYPE_REQUEST_WRITE` reads the offset and chunk from the client, and writes it to the export's backend:

```
// Reading the chunk to be written from the client's connection
n, err := io.ReadAtLeast(conn, b[:requestHeader.Length], int(requestHeader.Length))
// ...

// Writing it to the backend
export.Backend.WriteAt(b[:n], int64(requestHeader.Offset))

// Acknowledging the write to the client
binary.Write(conn, binary.BigEndian, protocol.TransmissionReplyHeader{
    ReplyMagic: protocol.TRANSMISSION_MAGIC_REPLY,
    Error:      0,
    Handle:     requestHeader.Handle,
})
```

It is here that the read-only option is implemented by sending a permission error in case of writes:

```
if options.ReadOnly {
    // Discard the write command's data
    _, err := io.CopyN(io.Discard, conn, int64(requestHeader.Length))

    // Sending the error reply
    binary.Write(conn, binary.BigEndian, protocol.
       TransmissionReplyHeader{
        ReplyMagic: protocol.TRANSMISSION_MAGIC_REPLY,
        Error:      protocol.TRANSMISSION_ERROR_EPERM,
        Handle:     requestHeader.Handle,
    })

    break
}
```

Finally, the TRANSMISSION_TYPE_REQUEST_DISC transmission message type gracefully disconnects the client from the server and causes the backend to sync, i.e. to flush and outstanding writes to disk. This is especially important in order to support the lifecycle of the migration API.

4.4.3 Client

Unlike the server, the client is implemented by using both the kernel's NBD client and a userspace component. In order to use the kernel NBD client, it is necessary to first find a free NBD device (/dev/nbd*); these devices are allocated by the kernel NBD module and can be specified with the nbds_max parameter[46]. In order to find a free device, we can either specify it manually, or check sysfs for a NBD device that reports a zero size:

```
// Using a glob on sysfs for the NBD device size
statPaths, err := filepath.Glob(path.Join("/sys", "block", "nbd*", "
    size"))
// ...

// Finding the first device that reports a zero zsize
for _, statPath := range statPaths {
    rsize, err := os.ReadFile(statPath)
```

```go
    // ...

    size, err := strconv.ParseInt(strings.TrimSpace(string(rsize)), 10,
         64)
    // ...

    if size == 0 {
        return filepath.Join("/dev", filepath.Base(filepath.Dir(
            statPath))), nil
    }
}
```

After a free NBD device has been found, the client can be started by calling `Connect` with a `net.Conn` and options, similarly to the server:

```go
func Connect(conn net.Conn, device *os.File, options *Options) error
```

This time, the options can define additonal information such as the client's preferred blocksize, connection timeouts or requested export name, which, in this scenario, can be used to refer to a specific memory region. The kernel's NBD device is then configured to use the connection; the relevant `ioctl` constants are extracted by using CGo, or hard-coded values if CGo is not available:

```go
// Only use CGo if it is available
//go:build linux && cgo

// Importing the kernel headers
/*
#include <sys/ioctl.h>
#include <linux/nbd.h>
*/
import "C"

const (
    // Extracting the `ioctl` numbers with `CGo`
    NEGOTIATION_IOCTL_SET_SOCK        = C.NBD_SET_SOCK
    // ...
)
```

The handshake for the NBD cleint is negotiated in userspace by Go. Similarly to the server, the client only supports the "fixed newstyle" negotiatiation and aborts otherwise. The negotiation is once again implemented as a simple loop similarly to the server with it switching on the type; on `NEGOTIATION_TYPE_REPLY_INFO`, the client receives the export size, and with `NEGOTIATION_TYPE_INFO_BLOCKSIZE` it receives the used block size, which it then valides to be within the specified bounds and as a valid power of two, falling back to the preffered block size supplied by the options if possible:

```go
// Falling back to the client's prefered block size if none is provided
```

```go
, and checking if the server's advertised size is within the clients's bounds as supplied by the options
if options.BlockSize == 0 {
    chosenBlockSize = info.PreferredBlockSize
} else if options.BlockSize >= info.MinimumBlockSize && options.BlockSize <= info.MaximumBlockSize {
    chosenBlockSize = options.BlockSize
} else {
    return ErrUnsupportedServerBlockSize
}

// Validating upper and lower supported bounds set by the kernel's NBD client
if chosenBlockSize > MaximumBlockSize {
    return ErrMaximumBlockSize
} else if chosenBlockSize < MinimumBlockSize {
    return ErrMinimumBlockSize
}

// Validating that the chosen block size is a power of two
if !((chosenBlockSize > 0) && ((chosenBlockSize & (chosenBlockSize - 1)) == 0)) {
    return ErrBlockSizeNotPowerOfTwo
}
```

After this relevant metadata has been fetched from the server, the kernel NBD client is further configured with these values using `ioctl`, after which the `DO_IT` `ioctl` number is used to asynchronously start the kernel's NBD client:

```go
syscall.Syscall(
    syscall.SYS_IOCTL,
    device.Fd(),
    ioctl.NEGOTIATION_IOCTL_SET_BLOCKSIZE,
    uintptr(chosenBlockSize),
)

// ...

go func() {
    syscall.Syscall(
        syscall.SYS_IOCTL,
        device.Fd(),
        ioctl.NEGOTIATION_IOCTL_DO_IT,
        0,
    )
    // ...
}()
```

In addition to being able to configure the client itself, the client library can also be used to list the exports of a server; for this, another handshake is initiated, but this time the `NEGOTIATION_ID_OPTION_LIST` option is provided, after which the client reads the export information from the server and disconnects:

```go
exportNames := []string{}
for {
    // Reading the export name's length
    var exportNameLength uint32
    if err := binary.Read(info, binary.BigEndian, &exportNameLength); err != nil {
        // All the exports have been received
        if errors.Is(err, io.EOF) {
            break
        }
        // ...
    }

    // Reading the export name
    exportName := make([]byte, exportNameLength)
    io.ReadFull(info, exportName)

    exportNames = append(exportNames, string(exportName))
}
```

4.4.4 Client Lifecycle

The final `DO_IT` `ioctl` never returns until it is disconnected, meaning that an external system must be used to detect whether the device is actually ready. There are two fundamental ways of doing this: By polling `sysfs` for the size parameter as it was done for finding an unused NBD device, or by using `udev`.

`udev` manages devices in Linux, and as a device becomes available, the kernel sends an event using this subsystem. By subscribing to this system with the expected NBD device name to catch when it becomes available, it is possible to have a reliable and idiomatic way of detecting the ready state:

```go
// Connecting to `udev`
udevConn.Connect(netlink.UdevEvent)

// Subscribing to events for the device name
udevConn.Monitor(udevReadyCh, udevErrCh, &netlink.RuleDefinitions{
    Rules: []netlink.RuleDefinition{
        {
            Env: map[string]string{
                "DEVNAME": device.Name(),
            },
        },
    },
})
```

```go

// Waiting for the device to become available
go func() {
    // ...
    <-udevReadyCh

    options.OnConnected()
}()
```

In reality however, due to overheads in `udev`, it can be faster to use polling instead of the even system, which is why it is possible to set the `ReadyCheckUdev` option in the NBD client to **`false`**, which uses polling instead:

```go
// Opening up the NBD device's size file in `sysfs`
sizeFile, err := os.Open(filepath.Join("/sys", "block", filepath.Base(
    device.Name()), "size"))

for {
    // Reading the size file and parsing it's contents as a number
    sizeFile.Seek(0, io.SeekStart)
    io.ReadAll(sizeFile)
    size, err := strconv.ParseInt(strings.TrimSpace(string(rsize)), 10,
         64)

    // If the size is no longer reported as zero, return
    if size > 0 {
        options.OnConnected()

        return
    }

    // If the device is not yet ready, sleep until the next polling
        cycle
    time.Sleep(options.ReadyCheckPollInterval)
}
```

Similarly to the setup lifecycle, the teardown lifecycle is also as an asynchronous operation. It works by calling three `ioctl`s on the NBD device's file descriptor, causing it to disconnect from the server and causing the prior `DO_IT` syscall to return, which in turn causes the prior call to `Connect` to return:

```go
// Complete any remaining reads/writes
syscall.Syscall(
    syscall.SYS_IOCTL,
    device.Fd(),
    ioctl.TRANSMISSION_IOCTL_CLEAR_QUE,
    0,
)

```

```
// Disconnect from the NBD server
syscall.Syscall(
    syscall.SYS_IOCTL,
    device.Fd(),
    ioctl.TRANSMISSION_IOCTL_DISCONNECT,
    0,
)

// Disassociate the socket from the NBD device so that it can be used
   again
syscall.Syscall(
    syscall.SYS_IOCTL,
    device.Fd(),
    ioctl.TRANSMISSION_IOCTL_CLEAR_SOCK,
    0,
)
```

4.4.5 Optimizing Access to the Block Device

When opening the block devie that the client is connected to, the kernel usually provides a caching/buffer mechanism, requiring an expensive sync syscall to flush outstanding changes to the NBD client. As mentioend earlier, by using O_DIRECT it is possible to skip this caching layer and write all changes directly to the NBD client and thus the server, which is particularly useful in a case where both the client and server are on the same host, and the amount of time for syncing should be minimal, as is the case for a migration scenario. Using O_DIRECT however does come with the downside of requiring reads/writes that are aligned to the system's page size, which is possible to implement in the specific application using the device to access a resource, but not in a generic way.

4.4.6 Combining the NBD Client and Server to a Mount

When both the client and server are started on the same host, it is possible to connect them in an efficient way by creating a connected UNIX socket pair, returning a file descriptor for both the server and the client respectively, after which both components can be started in a new goroutine:

```
// Creating the socket pair
fds, err := syscall.Socketpair(syscall.AF_UNIX, syscall.SOCK_STREAM, 0)
// ..

// Starting the server on file descriptor 1
go func() {
    sf := os.NewFile(uintptr(fds[0]), "server")

    c, err := net.FileConn(sf)
```

```go

    sc := c.(*net.UnixConn)

    server.Handle(
        d.sc,
        // ...
    )
}()

// Starting the client on file descriptor 1
go func() {
    cf := os.NewFile(uintptr(fds[1]), "client")

    c, err := net.FileConn(cf)

    cc := c.(*net.UnixConn)
    // ...

    client.Connect(d.cc, /* ... */)
}()
```

This once again shows the benefit of not requiring a specific transport layer or `accept` semantics for the NBD library, as this makes it possible to skip the usually required TCP handshake for NBD.

This form of a combined client and server on the local device, with the server's backend providing the actual ressource, forms a direct path mount - where the path to the block device can be passed to the application consuming or providing the ressource, which can then choose to `open`, `mmap` etc. it. In addition to this simple path-based mount, a file mount is provided. This simply opens up the path as a file, so that it can be accessed with the common `read`/`write` syscalls; the benefit over simply using the path mount and handling the access in the application consuming the ressource is that common pitfalls around the lifecycle (`Close` and `Sync`) can be handled within the mount API directly.

The direct slice mount works similarly to the file mount, with the difference being that it `mmap`s the NBD device, bringing a variety of benefits such as not requiring syscalls to read/write from the memory region as mentioned before. The benefit of using the slice API over simply using the direct path mount API letting the application `mmap` the block device itself is once again the lifecycle, where `Close` and `Sync` handle the complexities of managing `mmap`ed regions, esp. around garbage collection and flushing, in the mount directly. As for the API design, another aspect however is critical; thanks to it providing a standard Go slice instead of a file, it is possible to use this interface to provide streaming ability to applications that expect to work with a `[]byte`, without requiring changes to the application itself:

```go
func (d *DirectSliceMount) Open() ([]byte, error)
```

It is also possible to format the backend for a NBD server/mount with a filesystem and mount the underlying filesystem on the host that accesses a resource, where a file on this filesystem can then be `open`ed/`mmap`ed similarly to the FUSE approach. This is particularly useful if there are multiple memory regions which all belong to the same application to synchronize, as it removes the need to start multiple block devices and reduces the latency overhead associated with it. This solution can be implemented by i.e. calling `mkfs.ext4` on a block device directly or by formatting the NBD backend ahead of time, which does however come at the cost of storing and transferring the file system metadata as well as the potential latency overhead of mounting it.

TODO: Reference voltools and how it works as a fast way of formatting the mount if it becomes OSS

4.5 Managed Mounts

4.5.1 Stages

In order to implement a chunking system and related components, a pipeline of readers/writers is a useful abstraction layer; as a result, the mount API is based on a pipeline of multiple `ReadWriterAt` stages:

```go
type ReadWriterAt interface {
    ReadAt(p []byte, off int64) (n int, err error)
    WriteAt(p []byte, off int64) (n int, err error)
}
```

This way, it is possible to forward calls to the NBD backends like `Size` and `Sync` directly to the underlying backend, but can chain the `ReadAt` and `WriteAt` methods, which carry actual data, into a pipeline of other `ReadWriterAt`s.

4.5.2 Chunking

One such `ReadWriterAt` is the the `ArbitraryReadWriterAt`. This chunking component allows breaking down a larger data stream into smaller chunks at aligned offsets, effectively making every read and write an aligned operation. In `ReadAt`, it calculates the index of the chunk that the currently read offset falls into as well as the offset within the chunk, after which it reads the entire chunk from the backend into a buffer, copies the requested portion of the buffer into the input slice, and repeats the process until all requested data is read:

```go
totalRead := 0
remaining := len(p)

buf := make([]byte, a.chunkSize)
// Repeat until all chunks that need to be fetched have been fetched
for remaining > 0 {
    // Calculating the chunk and offset within the chunk
    chunkIndex := off / a.chunkSize
    indexedOffset := off % a.chunkSize
    readSize := int64(min(remaining, int(a.chunkSize-indexedOffset)))

    // Reading from the next `ReadWriterAt` in the pipeline
    _, err := a.backend.ReadAt(buf, chunkIndex*a.chunkSize)
    // ...

    copy(p[totalRead:], buf[indexedOffset:indexedOffset+readSize])
    // ...

    remaining -= int(readSize)
}
```

The writer is implemented in a similar way; it starts by calculating the chunk and offset within the chunk. If an entire chunk is being written to at an aligned offset, it completely bypasses the chunking system, and writes the data directly to the backend so as to prevent unnecessary copies:

```go
// Calculating the chunk and offset within the chunk
chunkIndex := off / a.chunkSize
indexedOffset := off % a.chunkSize
writeSize := int(min(remaining, int(a.chunkSize-indexedOffset)))

// Full chunk is covered by the write request, no need to read
if indexedOffset == 0 && writeSize == int(a.chunkSize) {
    _, err = a.backend.WriteAt(p[totalWritten:totalWritten+writeSize],
        chunkIndex*a.chunkSize)
}
// ...
```

If this is not the case, and only parts of a chunk need to be written, it first reads the complete chunk into a buffer, modifies the buffer with the data that was changed, and then writes the entire buffer back until all data has been written:

```go
// Read the existing chunk
_, err = a.backend.ReadAt(buf, chunkIndex*a.chunkSize)

// Modify the chunk with the provided data
copy(buf[indexedOffset:], p[totalWritten:totalWritten+writeSize])

// Write back the updated chunk
_, err = a.backend.WriteAt(buf, chunkIndex*a.chunkSize)
```

This simple implementation can be used to efficiently allow reading and writing data of arbitrary length at arbitrary offsets, even if the backend only supports aligned reads and writes.

In addition to this chunking system, there is also a `ChunkedReadWriterAt`, which ensures that the limits concerning a backend's maximum chunk size and aligned reads/writes are being respected. Some backends, i.e. a backend where each chunk is represented by a file, might only support writing to aligned offsets, but don't support checking for this behavior; in this example, if a chunk with a larger chunk size is written to the backend, depending on the implementation, this could result in this chunk file's size being extended, which could lead to a DoS attack vector. It can also be of relevance if a client instead of a server is expected to implement chunking, and the server should simply enforce that the aligned reads and writes are being provided.

In order to check if a read or write is aligned, this `ReadWriterAt` checks whether an operation is done to an offset that is multiples of the chunk size, and whether the length of the slice of data is a valid chunk size:

```go
// Check if provided data is valid
if off%c.chunkSize != 0 || int64(len(p)) != c.chunkSize {
    return 0, ErrInvalidOffset
}

// Check if offset is valid
if off < 0 || off >= int64(c.chunkSize*c.chunks) {
    return 0, ErrInvalidReadSize
}

// Continues with the operation
```

4.5.3 Background Pull

The `Puller` component asynchronously pulls chunks in the background. It starts by sorting the chunks with the pull heuristic mentioned earlier, after which it starts a fixed number of worker threads in the background, each which ask for a chunk to pull:

```go
// Sort the chunks according to the pull priority callback
sort.Slice(chunkIndexes, func(a, b int) bool {
    return pullPriority(chunkIndexes[a]) > pullPriority(chunkIndexes[b
        ])
})

// ...

for {
    // Get the next chunk
    chunk := p.getNextChunk()

    // Exit after all chunks have been pulled
    if chunk >= p.chunks {
        break
    }
    // ...

    // Reading the chunk from the backend
    _, err := p.backend.ReadAt(make([]byte, p.chunkSize), chunkIndex*p.chunkSize)
    // ...
}
```

Note that the puller itself does not copy any data from the destination; this is handled by a separate component. It simply reads from the next provided pipeline stage, which is expected to handle the actual copying process.

An implementation of this stage is the `SyncedReadWriterAt`, which takes both a remote and local `ReadWriterAt` pipeline stage as it's argument. If a chunk is read, i.e. by the puller component calling `ReadAt`, it is tracked and marked as remote by adding it to a local map. The chunk itself is then read from the remote reader and written to the local one, after which it is marked as locally available, meaning that on the second read it is fetched from the faster, local reader instead; a callback is used to make it possible to track the syncer's pull progress:

```go
// Track chunk
chk := c.getOrTrackChunk(off)

// If chunk is available locally, return it
if chk.local {
    return c.local.ReadAt(p, off)
}

// If chunk is not available locally, copy it from the remote, then mark the chunk as local
c.remote.ReadAt(p, off)
c.local.WriteAt(p, off)
chk.local = true

// Enable progress tracking
c.onChunkIsLocal(off)
```

Note that since this is a pipeline stage, this behavior also applies to reads that happen aside from those initiated by the `Puller`, meaning that any chunks that haven't been fetched asynchronously before they are being accessed will be scheduled to be pulled immediately. The `WriteAt` implementation of this stage immediately marks and reports the chunk as available locally no matter whether it has been pulled before or not.

The combination of the `SyncedReadWriterAt` stage and the `Puller` component implements a pre-copy migration system in an independently unit testable way, where the remote ressource is being pre-emptively copied to the destination system first. In addition to this however, since it can also schedule chunks to be available immediately, it has some of the characteristics of a post-copy migration system, too, where it is possible to fetch chunks as they become available, making it behave similarly to the `rr+` prefetching mechanism mentioned in "Remote Regions"[47]. Using this combination, it is possible to implement the full read-only managed mount API.

4.5.4 Background Push

In order to also allow for writes back to the remote source host, the background push component exists. Once it has been opened, it schedules recurring writebacks to the remote by calling `Sync`; once this is called by either the background worker system or another component, it launces writeback workers in the background. These wait to receive a chunk that needs to be written back; once they receive one, they read it from the local `ReadWriterAt` and copy it to the remote, after which the chunk is marked as no longer requiring writebacks:

```go
// Wait until the worker gets a slot from a semaphore
p.workerSem <- struct{}{}

// First fetch from local ReaderAt, then copy to remote one
b := make([]byte, p.chunkSize)
p.local.ReadAt(b, off)
p.remote.WriteAt(b, off)

// Remove the chunk from the writeback queue
delete(p.changedOffsets, off)
```

In order to prevent chunks from being pushed back to the remote before they have been been pulled first or written to locally, the background push system is integrated into the `SyncedReadWriterAt` component. This is made possible by intercepting the offset passed to the progress callback, and only then marking it as ready:

```go
chunks.NewSyncedReadWriterAt(m.remote, local, func(off int64) error {
    return local.(*chunks.Pusher).MarkOffsetPushable(off)
})
```

Unlike the puller component, the pusher also functions as a pipeline step, and as such provides a `ReadAt` and `WriteAt` implementation. While `ReadAt` is a simple proxy forwarding the call to the next stage, `WriteAt` marks a chunk as pushable, causing it to be written back to the remote on the next writeback cycle, before writing the chunk to the next stage. If a managed mount is intended to be read-only, the pusher is simply not included in the pipeline.

4.5.5 Pipeline

For the direct mount system, the NBD server was connected directly to the remote; managed mounts on the other hand have an internal pipeline of pullers, pushers, a syncer, local and remote backends as well as a chunking system:

TODO: Add graphic of the internal pipeline and how systems are connected to each other

Using such a pipeline system of independent stages and other components also makes the system very testable. To do so, instead of providing a remote and local `ReadWriterAt` at the source and drain of the pipeline respectively, a simple in-memory or on-disk backend can be used in the unit tests. This makes the individual components unit-testable on their own, as well as making it possible to test and benchmark edge cases (such as reads that are smaller than a chunk size) and optimizations (like different pull heuristics) without complicated setup or teardown procedures, and without having to initialize the complete pipeline.

4.5.6 Concurrent Device Initialization

The background push/pull components allow pulling from the remote pipeline stage before the NBD device itself is open. This is possible because the device doesn't need to start accessing the data in a post-copy sense to start the pull, and means that the pull process can be started as the NBD client and server are still initializing. Both components typically start quickly, but the initialization might still take multiple milliseconds. Often, this amounts to roughly one RTT, meaning that making this initialization procedure concurrent can signficantly reduce the initial read latency by pre-emptively pulling data. This is because even if the first chunks are being accessed right after the device has been started, they are already available to be read from the local backend instead of the remote, since they have been pulled during the initialization and thus before the mount has even been made available to application.

4.5.7 Device Lifecycles

Similarly to how the direct mount API used the basic path mount to build the file and slice mounts, the managed mount API provides the same interfaces. In the case of managed mounts however, this is even more important, since the synchronization lifecycle needs to be taken into account. For example, in order to allow the `Sync()` API to work, the `mmap`ed region must be `msync`ed before the `SyncedReadWriterAt`'s `Sync()` method is called. In order to support these flows without tightly coupling the individual pipeline stages, a hooks system exists that allows for such actions to be registered from the managed mount, which is also used to implement the correct lifecycle for closing/tearing down a mount:

```go
type ManagedMountHooks struct {
    OnBeforeSync func() error
    OnBeforeClose func() error
    OnChunkIsLocal func(off int64) error
}
```

4.5.8 WAN Optimization

While the managed mount system functions as a hybrid pre- and post-copy system, optimizations are implemented that make it more viable in a WAN scenario compared to a typical pre-copy system by using a unidirectional API. Usually, a pre-copy system pushes changes to the destination host. In many WAN scenarios however, NATs prevent a direct connection. Moreover, since the source host needs to keep track of which chunks have already been pulled, the system becomes stateful on the source host and events such as network outages need to be recoverable from.

By using the pull-only, unidirectional API to emulate the pre-copy setup, the destination can simply keep track of which chunks it still needs to pull itself, meaning that if there is a network outage, it can just resume pulling or decide to restart the pre-copy process. Unlike the pre-copy system used for the file synchronization/hashing approach, this also means that destination hosts don't need to subscribe to a central multiplexing hub, and adding clients to the topology is easy since their pull progress state does not need to be stored anywhere except the destination node.

4.6 Live Migration

4.6.1 Overview

As mentioned in Pull-Based Synchronization with Migrations earlier, the mount API is not optimal for a migration scenario. Splitting the migration into two discrete phases can help fix the biggest problem, the maximum guaranteed downtime; thanks to the flexible pipeline system of `ReadWriterAt`s, a lot of the code from the mount API can be reused for the migration, even if the API and corresponding wire protocol are different.

4.6.2 Seeder

The seeder defines a new read-only RPC API, which, in addition the known `ReadAt`, also adds new RPCs such as `Sync`, which is extended to return dirty chunks, as well as `Track()`, which triggers a new tracking stage:

```go
type SeederRemote struct {
    ReadAt func(context context.Context, length int, off int64) (r ReadAtResponse, err error)
    Size   func(context context.Context) (int64, error)
    Track  func(context context.Context) error
    Sync   func(context context.Context) ([]int64, error)
    Close  func(context context.Context) error
}
```

Unlike the remote backend, the seeder also exposes a mount through the familiar path, file or slice APIs, meaning that even as the migration is in progress, the underlying ressource can still be accessed by the application on the source host. This fixes the architectural constraint of the mount API when used for the migration, where only the destination is able to expose a mount, while the source simply serves data without accessing it.

The tracking support is imlement in the same modular and composable way as the syncer, by providing a new pipeline stage, the `TrackingReadWriter`. Once activated by the `Track` RPC, the tracker intecepts all `WriteAt` calls and adds them to a local map before calling the next stage.:

```go
// If tracking is enabled, mark the chunk as dirty
if c.tracking {
    c.dirtyOffsetsIndex[off] = struct{}{}
    // ...
}

// Call the next stage
return c.backend.WriteAt(p, off)
```

Once the `Sync` RPC is called by the destination host, these dirty offsets are returned and the map is cleared:

TODO: Add graphic of the pipeline's design

A benefit of the protocol being defined in such a way that only the client ever calls an RPC, thus making the protocol uni-directional, is that both the transport layer and RPC system are completely interchangeable. This works by returning a simple abstract `service` utility struct from `Open`, which can then be used as the implementation for any RPC framework, i.e. with the actual gRPC service simply functioning as an adapter:

```go
type SeederGrpc struct {
    v1.UnimplementedSeederServer

    svc *Seeder
}

// ...

func (s *SeederGrpc) ReadAt(ctx context.Context, args *v1.ReadAtArgs) (*v1.ReadAtReply, error) {
    res, err := s.svc.ReadAt(ctx, int(args.GetLength()), args.GetOff())
    // ...

    return &v1.ReadAtReply{
        N: int32(res.N),
        P: res.P,
    }, nil
}
```

4.6.3 Leecher

The leecher then takes this abstract service struct provided by the seeder, which is implemented by a RPC framework. Using this, as soon as the leecher is opened, it calls `Track()` in the background and starts the NBD device in parallel to achieve a similar reduction in initial read latency as the mount API. The leecher introduces a new pipeline stage, the `LockableReadWriterAt`:

TODO: Add graphic of pipeline design

This component simply blocks all read and write operations to/from the NBD device until `Finalize` has been called by using a `sync.Cond`. This is required becaused otherwise, stale data (before `Finalize` marked the chunks as dirty) could have poisoned the kernel's file cache if the application read data before finalization:

```go
// For `ReadAt/WriteAt`: Waits for finalization, then calls the actual read/write operation
a.lock.L.Lock()
if a.locked {
    a.lock.Wait()
}
a.lock.L.Unlock()

// Unlocks the `sync.Cond` and broadcasts the new unlocked state to all previously blocked `ReadAt`/`WriteAt` operations
func (a *LockableReadWriterAt) Unlock() {
    a.lock.L.Lock()
    a.locked = false
    a.lock.Broadcast()
    a.lock.L.Unlock()
}
```

Once the leecher has started the device, it sets up a syncer in the same way as the mount API. A callback can again be used to monitor the pull progress, and once the reported availability is satisfactory, `Finalize` can be called. This then handles the critical migration phase, in which the remote application consuming the resource must be suspended; to do this, `Finalize` calls `Sync` on the seeder, causing it to return the dirty chunks and suspending the remote application, while the leecher marks the dirty chunks as remote and schedules them to be pulled immediately in the background to optimize for temporal locality:

```go
// Suspends the remote application, flushes the mount and returns offsets that have been written too since `Track()`
dirtyOffsets, err := l.remote.Sync(l.ctx)

```

```
// Marks the chunks as remote, causing subsequent reads to pull them again
l.syncedReadWriter.MarkAsRemote(dirtyOffsets)

// Schedules the chunks to be pulled in the background immediately
l.puller.Finalize(dirtyOffsets)

// Unlocks the local ressource for reading
l.lockableReadWriterAt.Unlock()
```

As an additional measure aside from the lockable `ReadWriterAt` is to make accessing the mount too earlier than after finalization harder, since only `Finalize` returns the mount, meaning that the API can't easily lead to deadlocks between `Finalize` and accessing the mount.

After the leecher has successfully reached 100% availability, it calls `Close` on the seeder and disconnects the leecher, causing both to shut down, after which the leecher can re-use the mount to provide a new seeder which can allow further migrations to happen in the same way.

[1] T. kernel development community, "Quick start." https://www.kernel.org/doc/html/next/rust/quick-start.html, 2023.

[2] R. Love, *Linux kernel development*, 3rd ed. Pearson Education, Inc., 2010.

[3] W. Mauerer, *Professional linux kernel architecture*. Indianapolis, IN: Wiley Publishing, Inc., 2008.

[4] W. R. Stevens, *Advanced programming in the UNIX environment*. Delhi: Addison Wesley Logman (Singapore) Pte Ltd., Indian Branch, 2000.

[5] K. A. Robbins and S. Robbins, *Unix™ systems programming: Communication, concurrency, and threads*. Prentice Hall PTR, 2003.

[6] W. Stallings, *Computer organization and architecture: Designing for performance*. Upper Saddle River, New Jersey, 07458: Pearson Education, Inc., 2010.

[7] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982, doi: 10.1145/356887.356892.

[8] J. Barr, "New - offline tape migration using AWS snowball edge." https://aws.amazon.com/blogs/aws/new-offline-tape-migration-using-aws-snowball-edge/, 2021.

[9] H. A. Maruf and M. Chowdhury, "Memory disaggregation: Advances and open challenges." 2023.Available: https://arxiv.org/abs/2305.03943

[10] J. Bonwick, "The slab allocator: An Object-Caching kernel," Jun. 1994.Available: https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel

[11] M. Gorman, *Understanding the linux virtual memory manager*. Upper Saddle River, New Jersey 07458: Pearson Education, Inc. Publishing as Prentice Hall Professional Technical Reference, 2004.

[12] T. K. D. Community, "Swap suspend," 2023. https://www.kernel.org/doc/html/latest/power/swsusp.html (accessed Jul. 19, 2023).

[13] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, 10th ed. Hoboken, NJ: Wiley, 2018.Available: https://lccn.loc.gov/2017043464

[14] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file I/O for In-Memory file systems," Jul. 2017.Available: https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[15] M. Prokop, "Inotify: Efficient, real-time linux file system event monitoring," Apr. 2010. https://www.infoq.com/articles/inotify-linux-file-system-event-monitoring/

[16] "Transmission Control Protocol." RFC 793; J. Postel, Sep. 1981. doi: 10.17487/RFC0793.

[17] "User Datagram Protocol." RFC 768; J. Postel, Aug. 1980. doi: 10.17487/RFC0768.

[18] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000; RFC Editor, May 2021. doi: 10.17487/RFC9000.

[19] A. Langley *et al.*, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196. doi: 10.1145/3098822.3098842.

[20] H. Xiao *et al.*, "Towards web-based delta synchronization for cloud storage services," in *16th USENIX conference on file and storage technologies (FAST 18)*, Feb. 2018, pp. 155–168.Available: https://www.usenix.org/conference/fast18/presentation/xiao

[21] T. libfuse authors, "FUSE minimal example filesystem using high-level API." https://github.com/libfuse/libfuse/blob/master/example/hello.c, 2020.

[22] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of User-Space file systems," in *15th USENIX conference on file and storage technologies (FAST 17)*, Feb. 2017, pp. 59–72.Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[23] A. Gaul, T. Nakatani, and @rrizun, "s3fs: FUSE-based file system backed by amazon S3." https://github.com/s3fs-fuse/s3fs-fuse, 2023.

[24] T. libfuse authors, "SSHFS: A network filesystem client to connect to SSH servers." https://github.com/libfuse/sshfs, 2022.

[25] E. Blake, W. Verhelst, and other NBD maintainers, "The NBD protocol." https://github.com/NetworkBlockDevice/nbd/blob/master/doc/proto.md, Apr. 2023.

[26] P. Clements, "[PATCH] nbd: Increase default and max request sizes." https://lore.kernel.org/lkml/20130402194120.54043222C0@clements/, Apr. 02, 2013.

[27] W. Verhelst, *Nbd-client man page*. 2023.Available: https://manpages.ubuntu.com/manpages/lunar/en/man8/nbd-client.8.html

[28] S. He, C. Hu, B. Shi, T. Wo, and B. Li, "Optimizing virtual machine live migration without shared storage in hybrid clouds," in *2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS)*, 2016, pp. 921–928. doi: 10.1109/HPCC-SmartCity-DSS.2016.0132.

[29] A. Baruchi, E. Toshimi Midorikawa, and L. Matsumoto Sato, "Reducing virtual machine live migration overhead via workload analysis," *IEEE Latin America Transactions*, vol. 13, no. 4, pp. 1178–1186, 2015, doi: 10.1109/TLA.2015.7106373.

[30] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems*. Sebastopol, CA: O'Reilly Media, Inc., 2018.

[31] J. D. Peek, *UNIX power tools*. Sebastopol, CA; New York: O'Reilly Associates; Bantam Books, 1994.

[32] gRPC Authors, "Introduction to gRPC." 2023.Available: https://grpc.io/docs/what-is-grpc/introduction/

[33] Redis Ltd, "Introduction to redis." https://redis.io/docs/about/, 2023.

[34] Redis Ltd, "Redis pub/sub." https://redis.io/docs/interact/pubsub/, 2023.

[35] Amazon Web Services, Inc, "What is amazon S3?" https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html, 2023.

[36] MinIO, Inc, "Core administration concepts." https://min.io/docs/minio/kubernetes/upstream/administration/concepts.html, 2023.

[37] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010, doi: 10.1145/1773912.1773922.

[38] P. Grabowski, J. Stasiewicz, and K. Baryla, "Apache cassandra 4.0 performance benchmark: Comparing cassandra 4.0, cassandra 3.11 and scylla open source 4.4," ScyllaDB Inc, 2021.Available: https://www.scylladb.com/wp-content/uploads/wp-apache-cassandra-4-performance-benchmark-3.pdf

[39] J. Corbet, "4.3 merge window, part 2." https://lwn.net/Articles/656731/, 2015.Available: https://lwn.net/Articles/656731/

[40] F. Pojtinger, "STFS file system unit tests." https://github.com/pojntfx/stfs/blob/main/pkg/fs/filesystem_test.go, 2022.

[41] S. Francia, "Fs interface - afero: A FileSystem abstraction system for go." https://pkg.go.dev/github.com/spf13/afero#Fs, 2023.

[42] F. Pojtinger, "STFS: Simple Tape File System, a file system for tapes and tar files." https://github.com/pojntfx/stfs, 2022.

[43] J. Waibel and F. Pojtinger, "sile-fystem: A generic FUSE implementation." https://github.com/jakWai01/sile-fystem, 2022.

[44] T. Grieger, "The cost and complexity of cgo." Dec. 2015.Available: https://www.cockroachlabs.com/blog/the-cost-and-complexity-of-cgo/

[45] F. Pojtinger, "Weron: Overlay networks based on WebRTC." https://github.com/pojntfx/weron, 2023.

[46] T. L. K. Contributors, "NBD kernel module in the linux kernel source tree." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/block/nbd.c#n2592, 2023.

[47] M. K. Aguilera *et al.*, "Remote regions: A simple abstraction for remote memory," in *Proceedings of the 2018 USENIX conference on usenix annual technical conference*, 2018, pp. 775–787.