Bachelorarbeit im Studiengang Medieninformatik

Efficient Synchronization of Linux Memory Regions over a Network

vorgelegt von

an der

am **04.08.2023**

zur Erlangung des akademischen Grades eines **Bachelor of Science**

Erstprüfer: **Prof. Dr. Martin Goik**

Zweitprüfer: **M.Sc. Philip Betzler**

Bachelor's Thesis

Efficient Synchronization of Linux Memory Regions over a Network

Author:

University:

Course of Study: **Media Informatics**

Date:

Academic Degree: **Bachelor of Science**

Primary Supervisor: **Prof. Dr. Martin Goik**

Secondary Supervisor: **M.Sc. Philip Betzler**

Contents

List of Acronyms

API Application Programming Interface

I/O Input/Output

OS Operating System

CPU Central Processing Unit

RAM Random Access Memory

SSD Solid State Drive

HDD Hard Disk Drive

CXL Compute Express Link

VFS Virtual File System

UUID Universally Unique Identifier

CRC32 Cyclic Redundancy Check 32-Bit

LRU Least Recently Used

WAN Wide Area Network

LAN Local Area Network

TCP Transmission Control Protocol

UDP User Datagram Protocol

P2P Peer-To-Peer

NATs Network Address Translators

IPC Inter-Process Communication

RTT Round-Trip Time

SRP SCSI RDMA Protocol

GNU GNU's Not Unix

UNIX UNIX Family of Operating Systems

macOS Apple Macintosh Operating System

FreeBSD Free Berkeley Software Distribution

NBD Network Block Device

S3fs S3 File System

NVMe Non-Volatile Memory Express

LTFS Linear Tape File System

LTO Linear Tape-Open

EXT4 Fourth Extended Filesystem

Btrfs B-Tree File System

C C Programming Language

Rust Rust Programming Language

Go Go Programming Language

C++ C++ Programming Language

ARM ARM RISC Computer Processor Architecture

x86 x86 CISC Computer Processor Architecture

RISC-V RISC-V RISC Computer Processor Architecture

LPDDR5 Low-Power Double Data Rate 5

HTTP Hypertext Transfer Protocol

HTTPS HTTP Secure

HTTP/2 HTTP Version 2

QUIC Quick UDP Internet Connections

WebRTC Web Real-Time Communication

Wasm WebAssembly

IETF Internet Engineering Task Force

OIDC OpenID Connect

AWS Amazon Web Services

CNCF Cloud Native Computing Foundation

S3 Simple Storage Service

TLS Transport Layer Security

mTLS Mutual TLS

SSH Secure Shell

DoS Denial of Service

JSON JavaScript Object Notation

JSONL JSON Lines

SQL Structured Query Language

NoSQL Not Only SQL

Protobuf Protocol Buffers

IDL Interface Definition Language

DSL Domain-Specific Language

KV Key-Value

Syscalls System Calls

VM Virtual Machine

RPC Remote Procedure Call

REST Representational State Transfer

FUSE File Systems in Userspace

1 Introduction

In today's technological landscape, numerous methods exist for accessing remote resources, such as via databases or custom APIs. The same applies to resource synchronization, which is typically addressed on a case-by-case basis via methods such as third-party databases, file synchronization services, or bespoke synchronization protocols. Resource migration is also a frequent challenge, solutions for which often rely on APIs better suited for long-term persistence, like storing the resource in a remote database. Today's solutions for resource synchronization are generally custom-built per application, despite the typical internal resource abstraction being a memory region or file.

What if, instead of applying application-specific protocols and abstractions for accessing, synchronizing, and migrating resources, these processes could be universally managed by directly operating on the memory region? While systems for interacting with remote memory exist, they primarily serve niche purposes, such as virtual machine live migration. They also suffer from the absence of a universal, generic API, often due to their design based on compatibility with a particular application's architecture such as a specific hypervisor, rather than them being intended for use as a library. This current state significantly diminishes developer experience, and represents a significant barrier for adoption.

In light of these limitations, this thesis explores alternative strategies to create a more universal approach to remote memory management. After looking at the current state of related technology, it details the implementation of selected methodologies using APIs like `userfaultfd` and NBD, discusses challenges and potential optimizations, and provides an outline for a universal API and related wire protocols. It also assesses the performance of various configurations, like background push and pull mechanisms, two-phase protocols and worker counts, to determine the optimal use case for each approach as well as their suitability for both WAN and LAN deployment contexts. Ultimately, it introduces a comprehensive, production-ready reference implementation of an NBD-based solution, which is able to cover most use cases in real-world applications today through the open-source r3map (remote mmap) library, before continuing to future research opportunities and possible improvements.

2 Technology

2.1 User Space and Kernel Space

The kernel represents the core of an operating system. It directly interacts with hardware, manages system resources such as CPU time, memory and others, and enforces security policies. In addition to this, it is also responsible for progress scheduling, memory management, drivers and many more responsibilities depending on the implementation. Kernel space refers to the memory region that this system is stored and executed in(**maurer2008professional?**).

User space on the other hand is the portion of system memory where user applications execute. Applications can't directly access hardware or kernel memory; instead they use APIs to access them(**tanenbaum2006operating?**). This API is provided in the form of syscalls, which serve as a bridge between user and kernel space. Well-known syscalls are `open()`, `read()`, `write()`, `close()` and `ioctl()`. While most syscalls have a specific purpose, `ioctl` serves as a more generic, universal one based on file descriptors and a data struct. Using it, it is possible to implement device-specific actions that can't be expressed with regular system calls. Despite their utility, they can be an implementation hurdle for language development due to their use of numerical constants and other typing issues(**devault2022hareioctl?**).

2.2 Linux Kernel

The Linux kernel was released by Linus Torvalds in 1991. Developed primarily in the C language, it has recently seen the addition of Rust as an approved option for further expansion and development, esp. for drivers(**linux2023docs?**). The kernel powers millions of devices across the globe, including servers, desktop computers, mobile phones, and embedded devices. As a kernel, it serves as an intermediary between hardware and applications. It is engineered for compatibility with a wide array of architectures, such as ARM, x86, RISC-V, and others. 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.

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

Linux is extensible, but not a microkernel. Despite its monolithic nature, it allows for the integration of kernel modules. These 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 reboot, helping to keep the kernel size both manageable and maintainable. Kernel modules are developed using the C or Rust programming languages, like the kernel itself, ensuring compatibility and consistent performance. They 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(**love2010linux?**).

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`. 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(**maurer2008professional?**).

2.3 UNIX Signals and Sockets

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(**stevens2000advanced?**). 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 resources before exiting or enabling verbose logging (**robbins2003unix?**).

It is however important to note that signals are not typically utilized as a primary inter-process communication (IPC) mechanism. This is due to their limitation in carrying additional data; while signals effectively alert a process of an event, they are not designed to convey further information related to that event, and as result they are best used in scenarios where simple event-based notifications are sufficient, rather than for more complex data exchange requirements. To work around this, sockets allow processes within the same host system to communicate with each other. Unlike UNIX signals, much like TCP sockets, they can easily be used for IPC by allowing not only to submit additional data for an event, and are particularly popular on Linux.

Stream sockets use TCP to provide reliable, two-way, connection-based byte streams, making them optimal for use in applications which require strong consistency guarantees. Datagram sockets on the other hand use UDP, which allows for fast, connection-less communication with fewer guarantees. In addition to these two different types of sockets, named and unnamed sockets exist. Named sockets are represented by a special file type on the file system and can be identified by a path, which allows for easy communication between unrelated processes. Unnamed sockets exist only in memory and disappear after the creating process terminates, making them a better choice for subsystems of applications to communicate with each other. In addition to this, UNIX sockets can pass a file descriptor between processes, which allows for interesting approaches to sharing resources with technologies such as `userfaultfd`(**stevens2003unixnet?**).

2.4 Memory Optimization

2.4.1 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.

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(**stallings2010architecture?**).

2.4.2 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(**smith1982cache?**). 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. While latency increases the further away a cache is from the CPU, the capacity of these caches typically also increases, which can be a worthwhile trade-off depending on the application.

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 files. Importantly, they are persistent, meaning they retain data even after power is cut(**stallings2010architecture?**).

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(**barr2021offline?**).

Depending on the technical choices for each of the hierarchy's layers, these latency differences can be quite significant, ranging from below a nanosecond for registers to multiple milliseconds for an HDD:

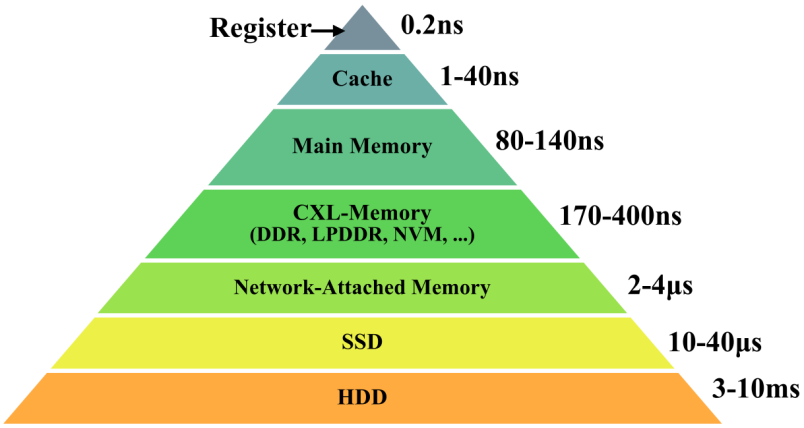

Figure 1: Latencies for different memory technologies showing, from lowest to highest latency, registers, cache, main memory, CXL memory, network-attached memory, SSDs and HDDs (**maruf2023memory?**)

The memory hierarchy is not static but evolves with technological advancements, leading to some blurring of these distinct layers. For instance, Non-Volatile Memory Express (NVMe) storage technologies can rival the speed of RAM while offering greater storage capacities(**maruf2023memory?**). 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.

2.5 Memory in Linux

2.5.1 Memory Management

Memory management forms a cornerstone of any kernel, serving as a critical buffer between applications and physical memory; it can be considered one of the fundamental purposes of a kernel 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 Linux, memory management is divided into two aforementioned major segments of kernel space and user space. The kernel memory module is responsible for managing kernel space. Slab allocation is a technique employed in managing this segment; the technique groups objects of the same size into caches, enhancing memory allocation speed and reducing fragmentation of

memory(**bonwick1994slaballoc?**). User space is the memory segment where applications and certain drivers store their memory in Linux. 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. 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 physical memory, enhancing the security and isolation of processes(**gorman2004linuxmem?**).

2.5.2 Swap Space

Swap space refers to a designated portion of the secondary storage utilized as virtual memory in a computer system. 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, the 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 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(**gorman2004linuxmem?**).

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(**kernel2023suspend?**).

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(**gorman2004linuxmem?**).

2.6 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.

They 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.

To minimize the occurrence of page faults, memory management algorithms such as the aforementioned LRU and the more straightforward clock algorithm are often used. These algorithms effectively manage the order and priority of memory pages, helping to ensure that frequently used pages are readily available in primary memory(**maurer2008professional?**).

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. This reduces the likelihood of major page faults by conserving memory space, allowing more pages to reside in primary memory(**silberschatz2018operating?**).

2.7 `mmap`

`mmap` is a 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 used 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(**choi2017mmap?**). This connection means that read operations performed on the mapped memory region directly correspond to reading the file and vice versa, enhancing efficiency as the amount of expensive context switches (compared to i.e. the `read` or `write` system calls) can be reduced.

The key advantage that `mmap` provides is its ability to do zero-copy operations. In practical terms, this means that 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.

This 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(**stevens2000advanced?**).

2.8 `inotify`

`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. 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 reduces overhead and resource use when compared to polling strategies. Polling is a I/O-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` is used across many applications, especially in file synchronization services. In this use case, 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 changes in, and might warrant additional management or fallback to heavier polling mechanisms for some parts of the system(**prokop2010inotify?**).

2.9 Linux Kernel 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 can be quickly accessed without expensive disk reads can significantly 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 contains information such as file names, attributes, and locations. The key benefit of this is that it speeds up the resolution of path names and file attributes, such as tracking when files were last changed for polling.

While such caching mechanisms can improve performance, they also introduce complexities. One such complexity is maintaining data consistency between the disk and cache through 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 stems from the necessity to release cached data under memory pressure, known as cache eviction. As mentioned before, this requires sophisticated algorithms, such as LRU, to ensure effective utilization of available cache space. Prioritizing what to keep in cache as the memory pressure increases directly impacts the overall system performance(**maurer2008professional?**).

2.10 Networking

2.10.1 RTT, LAN and WAN

Round-trip time (RTT) represents the time data takes to travel from a source to a destination and back. It provides a valuable insight into application latency, and can varying according to many factors such as network type, system load and physical distance. Local area networks (LAN) are geographically small networks characterized by having a low RTT, resulting in a low latency due to the short distance (typically no more than across an office or data center) that data needs to travel. As a result of their small geographical size and isolation, perimeter security is often applied to such networks, meaning that the LAN is viewed as a trusted network that doesn't necessarily require authentication or encryption between internal systems, resulting in a potentially lower overhead.

Wide area networks (WAN) on the other hand typically span a large geographical area, with the internet being an example that operates on a planetary scale. Due to the physical distance between source and destination, as well as the number of hops required for data to reach the destination, these networks typically have higher RTT and thus latency, and are also vulnerable to wire-tapping and packet inspection, meaning that in order to securely transmit data on them, encryption and authentication is required(**tanenbaum2003net?**).

2.10.2 TCP, UDP, TLS and QUIC

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

TCP has long been the reliable backbone for internet communication due to its connection-oriented nature. 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 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(**postel1981tcp?**).

UDP is a connectionless protocol that does not make the same guarantees about the reliability or ordered delivery of data packets. 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.(**postel1980udp?**)

TLS is an encryption protocol that intents to secure communication over a public network over the internet. It uses both symmetric and asymmetric encryption, and is used for most internet communication, esp. in combination with HTTP in the form of HTTPS. It consists of a handshake phase, in which the parameters necessary to establish a secure connection, as well as session keys and certificates are exchanged, before continuing on to the encrypted data transfer phase. Besides this use as a server authentication (through certificate authorities) and encryption method, it is also able to authenticate clients through the use of mutual TLS, where both the client and the server submit a certificate(**rescorla2018tls?**).

QUIC, a modern UDP-based transport layer protocol, was originally created by Google and standardized by the IETF in 2021(**rfc2021quic?**). It aspires to combine the best qualities of TCP and UDP; 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 (TLS) 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(**langley2017quic?**).

2.11 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.

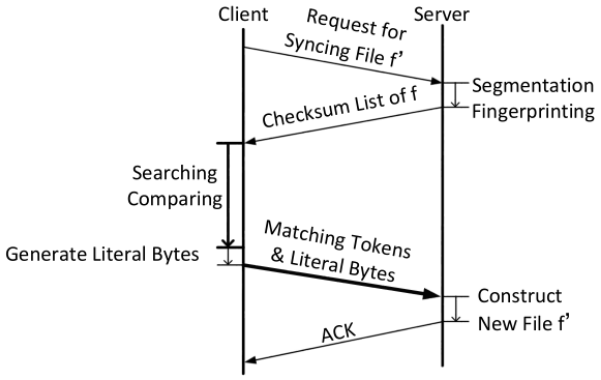

Figure 2: Design flow chart of WebRsync, showing the messages sent between and operations done for server and client in a single synchronization cycle(**xiao2018rsync?**)

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(**xiao2018rsync?**).

2.12 File Systems In User space (FUSE)

File Systems in User space (FUSE) is a software interface that enables the creation of custom file systems in the user space, 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 user space 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:

```
1 static int example_getattr(const char *path, struct stat *stbuf,
2                            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:

```
1 static int example_readdir(const char *path, void *buf, fuse_fill_dir_t
      filler,
2                            off_t offset, struct fuse_file_info *fi,
3                            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:

```
1 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:

```
1 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(**libfuse2020example?**).

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 user space program. This is followed by the user space program returning a response, which the FUSE kernel module conveys back to the user. As such, FUSE circumvents the complexity of implementing the file system implementation directly in the kernel. This architecture enhances safety, preventing entire kernel crashes due to errors within the implementation being limited to user instead of kernel space:

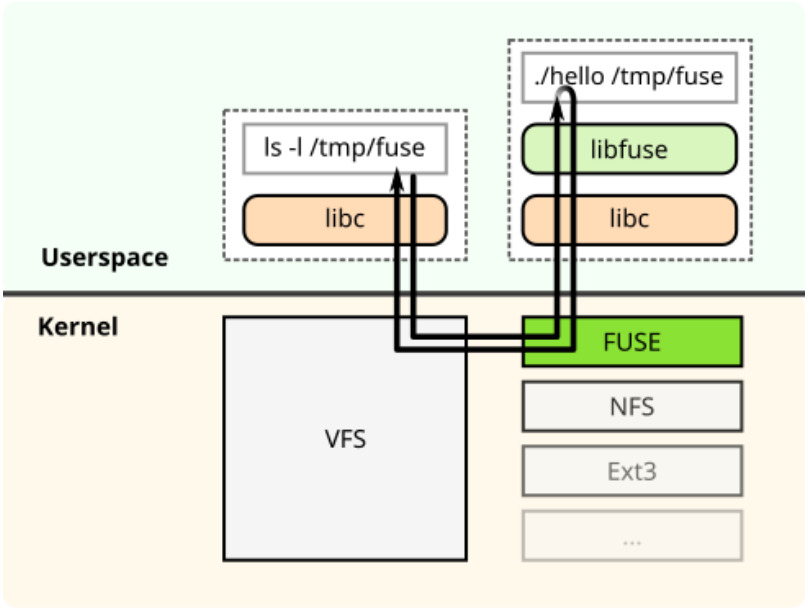

Figure 3: Structural diagram of FUSE, showing the user space components handled by the C library and the FUSE library as well as the kernel components such as the Linux VFS and the FUSE kernel module(**commons2023fusestructure?**)

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 user space that occurs during its operation(**vangoor2017fuse?**).

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`(**gaul2023s3fs?**) or to mount a remote system's disk via Secure Shell (SSH) with SSHFS (**libfuse2022sshfs?**).

2.13 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(**blake2023nbd?**):

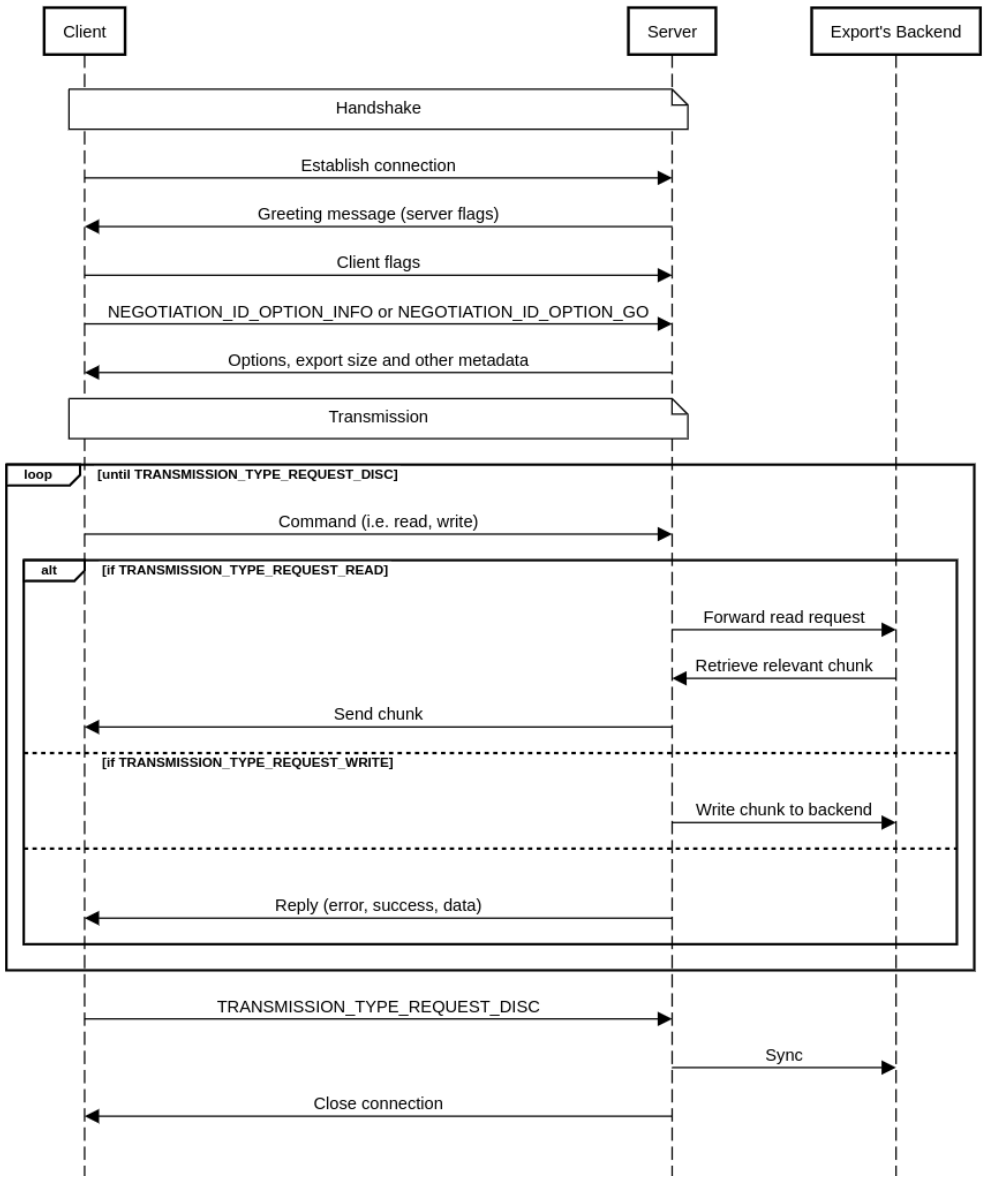

Figure 4: Sequence diagram of the baseline NBD protocol (simplified), showing the handshake, transmission and disconnect phases

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 responds 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(**clements2013nbd?**), and the maximum block or chunk size supported by the Kernel's NBD client is a mere 4 KB(**verhelst2023nbdclient?**). 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 NBD for a narrow use case.

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 use cases like memory synchronization, given that it operates on a block level, where synchronization features are not needed or implemented at a higher layer.

2.14 Virtual Machine Live Migration

Virtual machine live migration involves moving 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 the processes. Algorithms that implement this use case can be categorized into two broad types: pre-copy migration and post-copy migration.

2.14.1 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 transferring 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 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)(**he2016migration?**).

2.14.2 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.

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(**he2016migration?**).

2.14.3 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 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 a 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 alternative, 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(**baruchi2015workload?**).

2.15 Streams and Pipelines

Streams and pipelines are fundamental constructs that enable 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(**akidau2018streaming?**).

Pipelines are a series of data processing stages, where the output of one stage directly serves as the input to the next. 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(**peek1994unix?**); for instance, the results from a `curl` command fetching data from an API could be piped into the `jq` tool for JSON manipulation.

2.16 Go

Go is a statically typed, compiled open-source programming language released by Google in 2009. It is typically known for its simplicity, and was developed to address the unsuitability of many traditional languages for modern distributed systems development. Thanks to input from many people affiliated with UNIX, such as Rob Pike and Ken Thompson, as well as good support for concurrency, Go is particularly popular for the development of cloud services and other types of network programming. The headline feature of Go is “Goroutines”, a lightweight feature that allows for concurrent function execution that is similar to threads, but more scalable to support millions of Goroutines per program. Synchronization between different Goroutines is provided by using channels, which are type- and concurrency-safe conduits for data(**donovan2015go?**).

2.17 RPC Frameworks

2.17.1 gRPC and Protocol Buffers

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), more specifically proto3, 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(**google2023grpc?**).

2.17.2 fRPC and Polyglot

fRPC is an open-source RPC framework released by Loophole labs in 2022. It is proto3-compatible, meaning that it can be used as a drop-in replacement for gRPC, promising better performance characteristics. A unique feature is its ability to stop the RPC system to retrieve an underlying connection, which makes it possible to re-use connections for different purposes(**loopholelabs2023frpc?**). Internally, it uses Frisbee as it's messaging framework to implement the request-response semantics(**loopholelabs2022frisbee?**), and Polyglot, a high-performance serialization framework, as it's Protobuf equivalent. Polyglot achieves a similar goal as Protobuf, which is to encode data structures in a platform-independent way, but does so with less legacy code and a simpler wire format. It is also language-independent, with implementations for Go, Rust and TypeScript(**loopholelabs2023polyglot?**).

2.18 Data Stores

2.18.1 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(**redis2023introduction?**).

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 use case 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(**redis2023pubsub?**).

2.18.2 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.

It offers a variety of storage classes for 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 HTTP API. S3 also comes equipped with comprehensive security features, including authentication and authorization mechanisms. Access to objects and directories stored in S3 is done with HTTP(**aws2023s3?**).

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.

The main feature of Minio is its suitability for on-premises hosting, making it a good option for organizations with specific security regulations, those preferring to maintain direct control over their data and developers preferring 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(**minio2023coreadmin?**).

2.18.3 Cassandra and ScylllaDB

Apache Cassandra is a wide-column NoSQL database tailored for large-scale, distributed data management tasks. 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 does not require 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 other archictures(**lakshman2010cassandra?**).

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 perform optimally. In response to the perceived shortcomings of Cassandra, ScyllaDB was released 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.

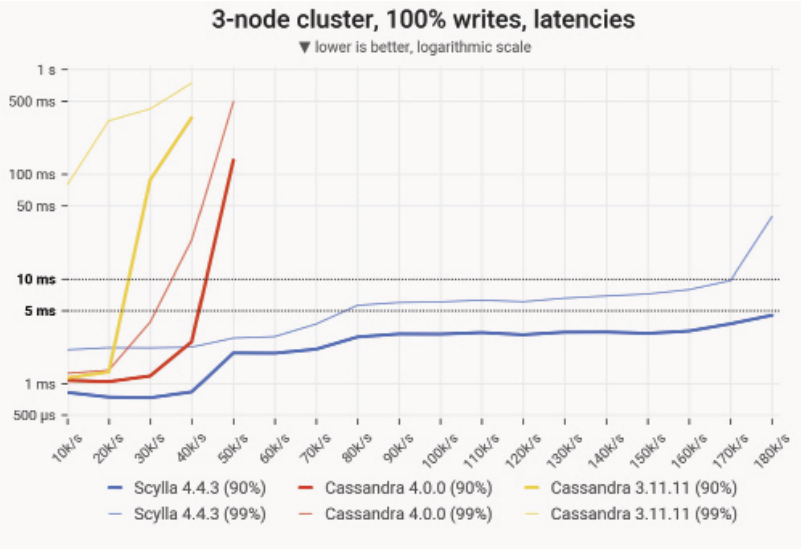

Figure 5: 90- and 99-percentile latency measurements of UPDATE queries for different load levels and different versions of Cassandra and ScyllaDB(**grabowski2021scylladb?**)

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 benchmarking studies(**grabowski2021scylladb?**).

3 Planning

3.1 Pull-Based Synchronization With `userfaultfd`

`userfaultfd` is a technology that allows for 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 resource 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 a `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 file system 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 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 use case of memory synchronization, not all the features provided by a full file system are 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 resource'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 its 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, referred to as a "mount", and is implemented as part of the r3map library (re**m**ote **mm**ap)(**pojtinger2023r3map?**).

By combining the NBD server and client into this reusable unit, 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 asynchronous background push/pull strategies instead of simply 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:

```
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.

3.5.2 Chunking

An 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 4 KB 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 the best if chunks are multiple MB instead of KB.

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 preemptive pulls that run asynchronously in the background. In order to optimize for spatial locality, a pull priority heuristic was introduced; this is used to determine the order in which chunks should be pulled. Many applications and other resources commonly access certain parts of their memory first, so if a resource should be accessible locally as quickly as possible (so that reads go to the local cache filled by the preemptive 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 significantly.

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 super blocks 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 conversely 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 use case, 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 use case, 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.

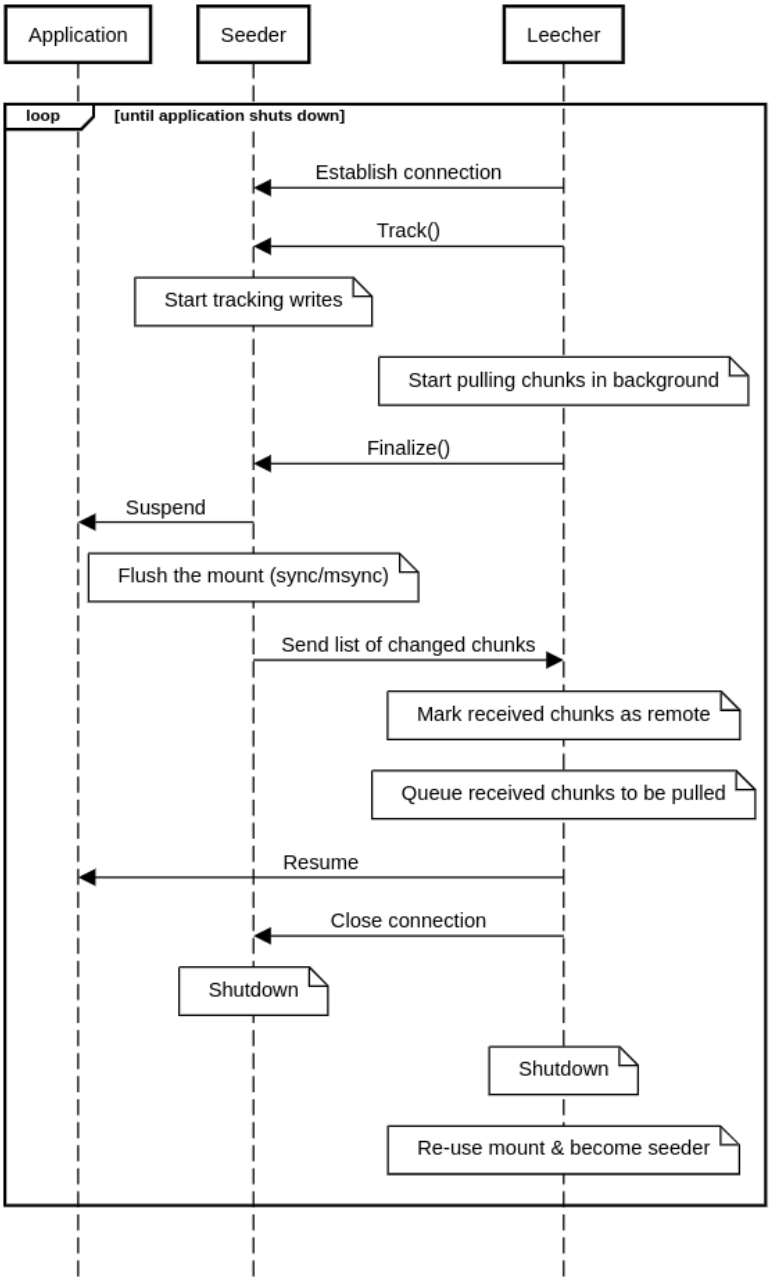

Figure 6: Sequence diagram of the migration protocol (simplified), showing the two protocol phases between the application that is being migrated, the seeder and the leecher components

By splitting the migration into these two distinct phases, the overhead of having to start the device does not count towards downtime, and additional app initialization that doesn't depend on the app's state (i.e. memory allocation, connecting to databases etc.) can happen 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(**corbet2015linux43?**) to handle these faults in user space in a more idiomatic way.

In the Go implementation created for this thesis, `userfaultfd-go`(**loopholelabs2022userfaultfdgo?**), `userfaultfd` works by first creating a region of memory, e.g. by using `mmap`, which is then registered with the `userfaultfd` API:

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

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

// Registering a 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, where it can then be received by the handler. 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 fault-

ing memory region over the same socket. 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 4 KB. 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:

```
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`:

```
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:

```
// ...
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 use case, 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 use case, 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 simplified. 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.

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.

4.2.5 File Advertisement and Receiver

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. 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 a flag has been set by a context cancellation which case it returns. If this is not the case, it waits for the specified polling interval, after which it restarts the transmission.

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. The file name is then 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.

4.2.6 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:

```
1 // Receiving remote hashes
2 remoteHashes := []string{}
3 utils.DecodeJSONFixedLength(conn, &remoteHashes)
4 // ...
5
6 // Calculating the hashes
7 localHashes, cutoff, err := GetHashesForBlocks(parallel, path,
      blocksize)
8
9 // Comparing the hashes
```

```
blocksToSend := []int64{}
for i, localHash := range localHashes {
    //  ...
    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.

4.2.7 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. 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:

```
// 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()
```

4.2.8 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
// 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.

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(**pojtinger2022stfstests?**). 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(**francia2023afero?**):

```
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)
    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(**pojtinger2022stfs?**) has shown that by using this abstraction layer, seemingly incompatible, non-linear backends can still be mapped to a file system. The project is backed 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(**waibel2022silefystem?**), 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:

```
// Creating the file system
serve := filesystem.NewFileSystem(
  afero.NewOsFs(),  // afero.Fs implementation here, followed by configuration
)
// 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(**pojtinger2023gonbd?**). 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 (**grieger2015cgo?**).

4.4.2 Server

The NBD server is implemented completely in user space, 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`:

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

```
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(**pojtinger2023weron?**), 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; 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:

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

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(**blake2023nbd?**) (see figure 4). The negotiation starts by the server sending the negotiation header to the NBD client and ignoring the client's flags. 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; 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, switching 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; it is here that the read-only option is implemented by sending a permission error in case of writes. 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 user space 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(**linux2023nbd?**). 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. 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.

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

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:

```
1  // Only use CGo if it is available
2  //go:build linux && cgo
3
4  // Importing the kernel headers
5  /*
6  #include <sys/ioctl.h>
7  #include <linux/nbd.h>
8  */
9  import "C"
10
11 const (
12     // Extracting the `ioctl` numbers with `CGo`
13     NEGOTIATION_IOCTL_SET_SOCK        = C.NBD_SET_SOCK
14     // ...
15 )
```

The handshake for the NBD client is negotiated in user space 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. 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. 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.

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:

```
// 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(),
            },
        },
    },
})

// 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. Similarly to the setup lifecycle, the teardown lifecycle is also as an asynchronous operation. It works by calling three `ioctl`s (`TRANSMISSION_IOCTL_CLEAR_QUE` to complete any remaining reads/writes, `TRANSMISSION_IOCTL_DISCONNECT` to disconnect from the NBD server and `TRANSMISSION_IOCTL_CLEAR_SOCK` to disassociate the socket from the NBD device so that it can be used again) 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.

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. This highlights the benefit of not requiring a specific transport layer or `accept` semantics for the NBD library, as it is possible to skip the usually required handshakes.

This form of a combined client and server on the local device, with the server's backend providing the actual resource, forms a direct path mount - where the path to the block device can be passed to the application consuming or providing the resource, 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 resource 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:

```
1 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.

4.5 Managed Mounts with r3map

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:

```
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:

```
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:

```
// 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:

```
// 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.

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:

```
// 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:

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

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

```
8
9  // If chunk is not available locally, copy it from the remote, then
       mark the chunk as local
10 c.remote.ReadAt(p, off)
11 c.local.WriteAt(p, off)
12 chk.local = true
13
14 // Enable progress tracking
15 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 resource 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. 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:

```
1  // Wait until the worker gets a slot from a semaphore
2  p.workerSem <- struct{}{}
3
4  // First fetch from local ReaderAt, then copy to remote one
5  b := make([]byte, p.chunkSize)
6  p.local.ReadAt(b, off)
7  p.remote.WriteAt(b, off)
8
9  // Remove the chunk from the writeback queue
10 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:

```
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.

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:

```
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 (see figure 6) 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:

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

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`. 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.

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:

```
1  // Suspends the remote application, flushes the mount and returns
       offsets that have been written too since `Track()`
2  dirtyOffsets, err := l.remote.Sync(l.ctx)
3
4  // Marks the chunks as remote, causing subsequent reads to pull them
       again
5  l.syncedReadWriter.MarkAsRemote(dirtyOffsets)
6
7  // Schedules the chunks to be pulled in the background immediately
8  l.puller.Finalize(dirtyOffsets)
9
10 // Unlocks the local resource for reading
11 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.

4.7 Pluggable Encryption, Authentication and Transport

Compared to existing remote memory and migration solutions, r3map is designed for a new field of application: WAN. Most existing systems that provide these solutions are intended to work in high-throughput, low-latency LAN, where assumptions concerning authentication and authorization as well as scalability can be made that are not valid in a WAN deployment. For example encryption: While in trusted LAN networks, it can be a viable option to assume that there are no bad actors in the local subnet, the same can not be assumed for WAN. While depending on i.e. TLS for the APIs would have been a viable option for r3map if it were to only support WAN deployments, it should still be functional and be able to take advantage of the guarantees if it is deployed in a LAN, which is why it is transport agnostic.

This makes adding guarantees such as encryption as simple as choosing the best solution depending on the network conditions. For low-latency, trusted networks, a protocol like the SCSI RDMA protocol (SRP) can be chosen, while for WAN, a standard internet protocol like TLS over TCP or QUIC can be used instead. Similarly to how the transport layer is interchangable, it is RPC-framework independent as well. This means that RPC frameworks such as dudirekta (which will be elaborated on later), which can work over P2P protocols like WebRTC data channels, are an option for environments with highly dynamic network topologies, where IP addresses rotate or there might be temporary loss of connectivity to recover from, as is the case with i.e. mobile networks, allowing live migration to work in completely new environments.

Since r3map makes no assumptions about them, authentication and authorization can be implemented in a similar way. For LAN deployments, the typical approach of simply trusting the local subnet can be used, for public deployments mTLS certificates or dedicated authentication protocols like OIDC can be an option. In networks with high RTT, QUIC allows the use of a 0-RTT handshake, which combines the connectivity and security handshake into one; this paired with mTLS can be an interesting option to decrease the initial read latency while still providing proper authentication.

4.8 Concurrent Backends

In high-RTT scenarios, the ability to fetch chunks concurrently is important. Without concurrent backgrounds pulls, latency can add up quickly, since every read to an offset of the memory region would have at least one RTT as it's latency, while concurrent pulls allow for multiple offsets' chunks to be pulled at the same time.

The first requirement for supporting this is that the remote backend has to be able to read from multiple regions without globally locking it. For the file backend for example, this is not the case, as a lock needs to be acquired for the entire file before an offset can be accessed:

```
1 func (b *FileBackend) ReadAt(p []byte, off int64) (n int, err error) {
2     b.lock.RLock()
3     defer b.lock.RUnlock()
4
5     n, err = b.file.ReadAt(p, off)
6     // ...
7 }
```

This can quickly become a bottleneck in the pipeline. One option that tries to solve this is the directory backend; instead of using just one backing file, the directory backend is a chunked backend that uses a directory of files, with each file representing a chunk. By using multiple files, this backend can lock each file (and thus chunk) individually, speeding up concurrent access. The same also applies to writees, where even concurrent writes to different chunks can safely be done at the same time as they are all backend by a separate file. This backend keeps track of these chunks by using an internal map of locks, and a queue to keep track of the order in which chunks' corresponding files were opened; when a chunk is first accessed, a new file is created for the chunk, and if the first operation is `ReadAt`, it is also truncated to exactly one chunk length. In order not to exceed the maximum number of open files for the backend process, a simple LRU algorithm is used to to close an open file if the limit would be exceeded.

4.9 Remote Stores as Backends

4.9.1 Overview

RPC backends provide a dynamic way to access a remote backend. This is useful for lots of use cases, esp. if the backend exposes a custom resource or requires custom authorization or caching. For the mount API specifically however, having access to a remote backend that doesn't require a custom RPC system can be useful, since the backend for a remote mount maps fairly well to the concept of a remote random-access storage device, for which many protocols and systems exist already.

4.9.2 Key-Value Stores with Redis

On such option is Redis, an in-memory key-value (KV) store with network access. To implement a mount backend, chunk offses can be mapped to keys, and since bytes are a valid key type, the chunk itself can be stored directly in the KV store; if keys don't exist, they are simply treated as empty chunks:

```
1  func (b *RedisBackend) ReadAt(p []byte, off int64) (n int, err error) {
2      // Retrieve a key corresponding to the chunk from Redis
3      val, err := b.client.Get(b.ctx, strconv.FormatInt(off, 10)).Bytes()
4      // If  a key does not exist, treat it as an empty chunk
5      if err == redis.Nil {
6          return len(p), nil
7      }
8      // ...
9  }
10
11 func (b *RedisBackend) WriteAt(p []byte, off int64) (n int, err error)
       {
12     // Store an offset as a key-value pair in Redis
13     b.client.Set(b.ctx, strconv.FormatInt(off, 10), p, 0)
14     // ...
15 }
```

Using Redis is particularly interesting because it is able to handle locking the indiviudal chunks server-side in an efficient way, and thanks to it's custom protocol and fast serialization it is well-suited for high-throughput deployment scenarios. Authentication and authorization for this backend can be handled using the Redis protocol, and hosting multiple memory regions can be implemented by using multiple databases or key prefixes.

4.9.3 Object Stores with S3

While Redis is interesting for high-throughput scenarios, when it comes to making a memory region available on the public internet, it might not be the best choice due to its low-level, custom protocol and (mostly) in-memory nature. This is where S3 can be used; a S3 backend can be a good choice for mounting public information, i.e. media assets, binaries, large filesystems and more into memory. While S3 has traditionally been mostly a AWS SaaS offering, projects such as Minio have helped it become the de facto standard for accessing files over HTTP. Similarly to the directory backend, the S3 backend is chunked, with one S3 object representing one chunk; if accessing a chunk returns a 404 error, it is treated as an empty chunk in the same way as the Redis backend, and multi-tenancy can once again be implemented either by using multiple S3 buckets or a prefix:

```
1  func (b *S3Backend) ReadAt(p []byte, off int64) (n int, err error) {
2      // Receiving a chunk using Minio's S3 client
3      obj, err := b.client.GetObject(b.bucket, b.prefix+"-"+strconv.
           FormatInt(off, 10), minio.GetObjectOptions{})
4      if err != nil {
5          // If an object is not found, it is treated as an empty chunk
6          if err.Error() == errNoSuchKey.Error() {
7              return len(p), nil
8          }
9
10         //  ...
11     }
12     // ...
13 }
```

4.9.4 Document Databases with ScylllaDB

Another option to access persistent remote resource is a NoSQL database such as Cassandra, specifically ScyllaDB, which improves on Cassandra's latency, a key metric for mounting remote resources. While this backend is more of a proof of concept rather than a real use case, it does show that even a database can be mapped to a memory region, which does allow for the interesting use case of making the databases' contents available directly in memory without having to use a database-specific client. Here, `ReadAt` and `WriteAt` are implemented by issues queries through ScylllaDB's DSL, where each row represents a chunk identified by it's offset as the primary, and as with Redis and S3, non-existing rows are treated as empty chunks:

```
1  func (b *CassandraBackend) ReadAt(p []byte, off int64) (n int, err
       error) {
2      // Executing a select query for a specific chunk, then scanning it
           into a byte slice
3      var val []byte
4      if err := b.session.Query(`select data from `+b.table+` where key =
           ? limit 1`, b.prefix+"-"+strconv.FormatInt(off, 10)).Scan(&val)
           ; err != nil {
5          if err == gocql.ErrNotFound {
6              return len(p), nil
7          }
8
9          return 0, err
10     }
11     // ...
12 }
13
14 func (b *CassandraBackend) WriteAt(p []byte, off int64) (n int, err
       error) {
15     // Upserting a row with a chunk's new content
16     b.session.Query(`insert into `+b.table+` (key, data) values (?, ?)`
           , b.prefix+"-"+strconv.FormatInt(off, 10), p).Exec()
17     // ...
18 }
```

Support for multiple regions can be implement by using a different table or key prefix, and migrations are used to create the table itself similarly to how it would be done in SQL.

4.10 Concurrent Bi-Directional RPCs with Dudirekta

4.10.1 Overview

Another aspect that plays an important role in performance for real-life deployments is the choice of RPC framework and transport protocol. As mentioned before, both the mount and migration APIs are transport-independent, and as a result almost any RPC framework can be used. A RPC framework developed as part of r3map is Dudireka(**pojtinger2023dudirekta?**). As such, it was designed specifically with the hybrid pre-and post-copy scenario in mind. To optimize for this, it has support for concurrent RPCs, which allows for efficient background pulls as multiple chunks can be pulled at the same time.

The framework also allows for defining functions on both the client and the server, which makes it possible to initiate pre-copy migratons and transfer chnks from the source host to the destination without having the latter be `dial`able; while making the destination host available by dialing it is possible in trusted LAN deployments, NATs and security concerns make it harder in WAN deployment. As part of this bi-directional support it is possible to also pass callbacks and closures as arguments to RPCs, which makes it possible to model remote generators with yields to easily report i.e. a migration's progress as it is running, while still modelling the migration with a single, transactional RPC and a return value. In addition to this, because dudirekta is also itself transport-agnostic, it is possible to use transport protocols like QUIC in WAN deployments, which can reduce the initial latency by using the 0-RTT handshake and thus makes calling an RPC less expensive. By not requiring TCP-style client-server semantics, Dudirekta can also be used to allow for P2P migrations over a protocol such as WebRTC(**pojtinger2023dudirektawebrtc?**).

4.10.2 Usage

Dudirekta is reflection based; both RPC definition and calling an RPC are completely transparent, which makes it optimal for prototyping the mount and migration APIs. To define the RPCs to be exposed, a simple implementation struct can be created for both the client and server. In this example, the server provides a simple counter with an increment RPC, while the client provides a simple `Println` RPC that can be called from the server. Due to protocol limitations, RPCs must have a context as their first argument, not have variadic arguments, and must return either a single value or an error:

```
1  // Server
2  type local struct { counter int64 }
3
4  func (s *local) Increment(ctx context.Context, delta int64) (int64,
       error) {
5      return atomic.AddInt64(&s.counter, delta), nil
6  }
7
8  // Client
9  type local struct{}
10
11 func (s *local) Println(ctx context.Context, msg string) error {
12     fmt.Println(msg)
13     // ...
14 }
```

In order to call these RPCs from the client/server respectively, the remote functions defined earlier are also created as placeholder structs that will be implemented by Dudirekta at runtime using reflection and are added to registry, which provides a handler that links a connection to the RPC implementation. This handler can then be linked to a any transport layer, i.e. TCP. After both registries have been linked to the transport, it is possible to call the RPCs exposed by the remote peers from both the server and the client, which makes bi-directional communication possible:

```
1  // Server: Calls the `Println` RPC exposed by the client
2  for _, peer := range registry.Peers() {
3      peer.Println(ctx, "Hello, world!")
4  }
5
6  // Client: Calls the `Increment` RPC exposed by the server
7  for _, peer := range registry.Peers() {
8      new, err := peer.Increment(ctx, 1)
9
10     log.Println(new) // Returns the value incremented by one
11 }
```

As mentioned earlier, Dudirekta also allows for passing in closures as arguments to RPCs; since this is also handled transparently, all that is necessary is to define the signature of the closure on the client and server, and it can be passed in as though the RPC were a local call, which also works on both the client and the server side:

```
// Server
func (s *local) Iterate(
    ctx context.Context,
    length int,
    onIteration func(i int, b string) (string, error), // Closure that is being passed in
) (int, error) {
    for i := 0; i < length; i++ {
        rv, err := onIteration(i, "This is from the callee") // Remote closure is being called
    }

    return length, nil
}

// Client
type remote struct {
    Iterate func(
        ctx context.Context,
        length int,
        onIteration func(i int, b string) (string, error), // The closure is defined in the placeholder struct
    ) (int, error)
}

for _, peer := range registry.Peers() {
    // `Iterate` RPC provided by the server is called
    length, err := Iterate(peer, ctx, 5, func(i int, b string) (string, error) {
        // Closure is transparently provided as a regular argumnt
        return "This is from the caller", nil
    })

    log.Println(length) // Despite having a closure as an argument, the RPC can still return values
}
```

4.10.3 Protocol

The protocol used for dudirekta is simple and based on JSONL, a for exchanging newline-delimited JSON data(**ward2013jsonl?**); a function call, i.e. to `Println` looks like this:

```
[true, "1", "Println", ["Hello, world!"]]
```

The first element marks the message as a function call, while the second one is the call ID, the third represents the name of the RPC that is being called followed by an array of arguments. A function return messages looks similar:

```
[false, "1", "", ""]
```

Here, the message is marked as a return value in the first element, the ID is passed with the second element and both the actual return value (third element) and error (fourth element) are nil and represented by an empty sring. Because it includes IDs, this protocol allows for concurrent RPCs through muxing and demuxing the JSONL messages.

4.10.4 RPC Providers

If an RPC (such as `ReadAt` in the case of the mount API) is called, a method with the provided RPC's name is looked up on the provided implementation struct and if it is found, the provided argument's types are validated against those of the implementation by unmarshalling them into their native natives. After the call has been validated by the RPC provider, the actual RPC implementation is executed in a new goroutine to allow for concurrent RPCs, the return and error value of which is then marshalled into JSON and sent back the caller.

In addition to the RPCs created by analyzing the implementation struct through reflection, to be able to support closures, a virtual `CallClosure` RPC is also exposed. This RPC is provided by a separate closure management component, which handles storing references to remote closure implementations; it also garbage collects those references after a RPC that has provided a closure has returned.

```
// If a closure is provided as an argument, handle it differently
if arg.Kind() == reflect.Func {
    closureID, freeClosure, err := registerClosure(r.local.wrapper, arg
        .Interface())
    // Freeing the closure after the RPC that has provided it is out of
        scope
    defer freeClosure()

    cmdArgs = append(cmdArgs, closureID)
}
```

4.10.5 RPC Calls

As mentioned earlier, on the caller's side, a placeholder struct representing the callee's available RPCs is provided to the registry. Once the registry is linked to a connection, the placeholder struct's methods are iterated over and the signatures are validated for compatibility with Dudirekta's limitations. They are then implemented using using reflection; these implementations simply marshal and unmarshal the function calls into Dudirekta's JSONL protocol upon being called, effectively functioning as transparent proxies to the remote implementations; it is at this point that unique call IDs are generated in order to be able to support concurrent RPCs:

```
// Creating the implementation method
reflect.MakeFunc(functionType, func(args []reflect.Value) (results []
    reflect.Value) {
        // Generating a unique call ID
        callID := uuid.NewString()

        cmdArgs := []any{}
        for i, arg := range args {
            // Collecting the function arguments
        }

        // Marshalling the JSONL for the call
        b, err := json.Marshal(cmd)

        // Writing the JSONL to the remote
        conn.Write(b)
        // ...
})
```

Once the remote has responded with a message containing the unique call ID, it unmarshals the return values, and returns from the implemented method. Closures are implemented similarly to this. If a closure is provided as a function argument, instead of marshing the argument and sending it as JSONL, the function is implemented by creating a "proxy" closure that calls the remote `CallClosure` RPC, while reusing the same marshalling/unmarshalling logic as regular RPCs. Calling this virtual `CallClosure` RPC is possible from both the client and the server because the protocol is bi-directional:

```
// If an argument is a function instead of a JSON-serializable value,
    handle it differently
if functionType.Kind() == reflect.Func {
    // Create a closure proxy
    arg := reflect.MakeFunc(functionType, func(args []reflect.Value) (
        results []reflect.Value) {
        // ...

        // A reference to the virtual `CallClosure` RPC
```

```
 8          rpc := r.makeRPC(
 9              "CallClosure",
10              // ...
11          )
12
13          // Continues with the same marshalling logic as regular
                function calls, then calls the virtual `CallClosure` RPC on
                the remote
14      })
15  }
```

As mentioned earlier, Dudirekta has a few limitations when it comes to the RPC signatures that are supported. This means that mount or migration backends can't be provided directly to the registry and need to be wrapped using an adapter. To not have to duplicate this translation for the different backends, a generic adapter between the Dudirekta API and the `go-nbd` backend (as well as the) interfaces exists.

4.11 Connection Pooling with gRPC

While the dudirekta RPC implementation serves as a good reference implementation of how RPC backends work, it has issues with scalability (see figure 23). This is mostly the case because of it's JSONL-based wire format, which, while simply and easy to analyze, is quite slow to marshal and unmarshal. The bi-directional RPCs do also come at a cost, since they prevent an effective use of connection pooling; since a client `dial`ing the server multiple times would mean that server could not reference multiple client connections as one composite client, it would not be able to differentiate two client connections from two separate clients. While implementing a future pooling mechanism based on a client ID is possible in the future, bi-directional RPCs can also be completely avoided entirely by implementing the pull- instead of push-based pre-copy solution described earlier where the destination host keeps track of the pull progress, effectively making unary RPC support the only requirement for a RPC framework.

Thanks to this narrower scope of requirements, alternative RPC frameworks can be used that do not have this limitation to their scalability. One such popular framework is gRPC, a high-performance system based on protocol buffers which is based on code generation and protocol buffers instead of reflection and JSONL. Thanks to it's support for unary RPCs, this protocol also supports connection pooling (which removes Dudirekta's main bottleneck) and is available in more language ecosystems (whereas Dudirekta currently only supports Go and TypeScript), making it possible to port the mount and migration APIs to other languages with wire protocol compatibility in the future. In order to implement the backend and seeder APIs for gRPC, they are defined in the `proto3` DSL:

```
// Fully-qualified package name
package com.pojtinger.felicitas.r3map.migration.v1;
// ...

// RPCs that are being provided, in this case for the migration API
service Seeder {
  rpc ReadAt(ReadAtArgs) returns (ReadAtReply) {};
  rpc Size(SizeArgs) returns (SizeReply) {};
  rpc Track(TrackArgs) returns (TrackReply) {};
  rpc Sync(SyncArgs) returns (SyncReply) {};
  rpc Close(CloseArgs) returns (CloseReply) {};
}

// Message definition for the `ReadAt` RPC
message ReadAtArgs {
  int32 Length = 1;
  int64 Off = 2;
}
// ... Rest of the message definitions
```

After generating the gRPC bindings from this DSL, the generated interface is implemented by using the Dudirekta RPC system's implementation struct as the abstract representation for the mount and migration gRPC adapters respectively, in order to reduce duplication.

4.12 Optimizing Throughput with fRPC

While gRPC tends to perform better than Dudirekta due to its support for connection pooling and more efficient binary serialization, it can be improved upon. This is particularly true for protocol buffers, which, while being faster than JSON, have issues with encoding large chunks of data, and can become a real bottleneck with large chunk sizes:

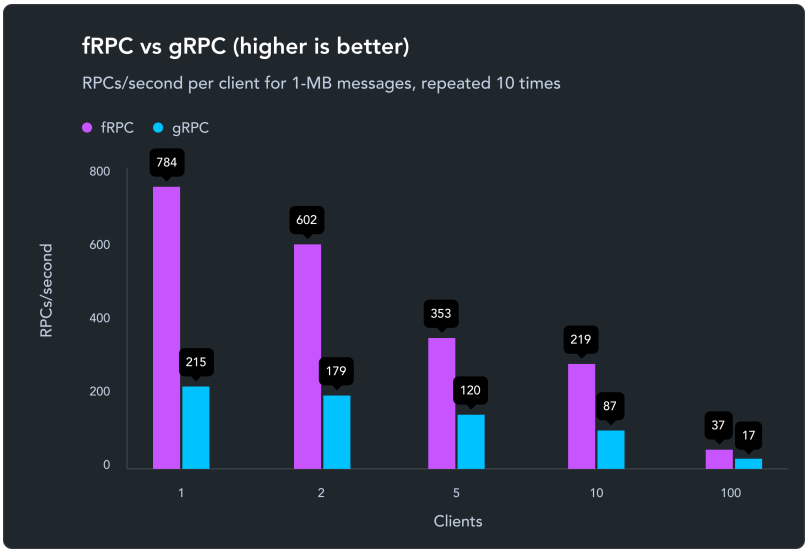

Figure 7: Amount of remote calls/second carrying 1 MB of data for fRPC and gRPC, repeated 10 times(**loopholelabs2023benchmarks?**)

fRPC(**loopholelabs2023frpc?**), a drop-in replacement for gRPC, can improve upon this by switching out the serialization layer with the faster Polyglot(**loopholelabs2023polyglot?**) library and a custom transport layer. It also uses the proto3 DSL and the same code generation framework as gRPC, which makes it easy to switch to by simply re-generating the code from the DSL. The implementation of the fRPC adapter functions in a very similar way as the gRPC adapter.

5 Results

5.1 Testing Environment

All benchmarks were conducted on a test machine with the following specifications:

Property	Value
Device Model	Dell XPS 9320
OS	Fedora release 38 (Thirty Eight) x86_64
Kernel	6.3.11-200.fc38.x86_64
CPU	12th Gen Intel i7-1280P (20) @ 4.700GHz
Memory	31687MiB LPDDR5, 6400 MT/s

To make the results reproducible, the benchmark scripts with additional configuration details and notebooks to plot the related visualizations can be found in the accompanying repository(**pojtinger2023memsync?**), and multiple runs have been conducted for each benchmark to ensure consistency.

5.2 Access Methods

5.2.1 Latency

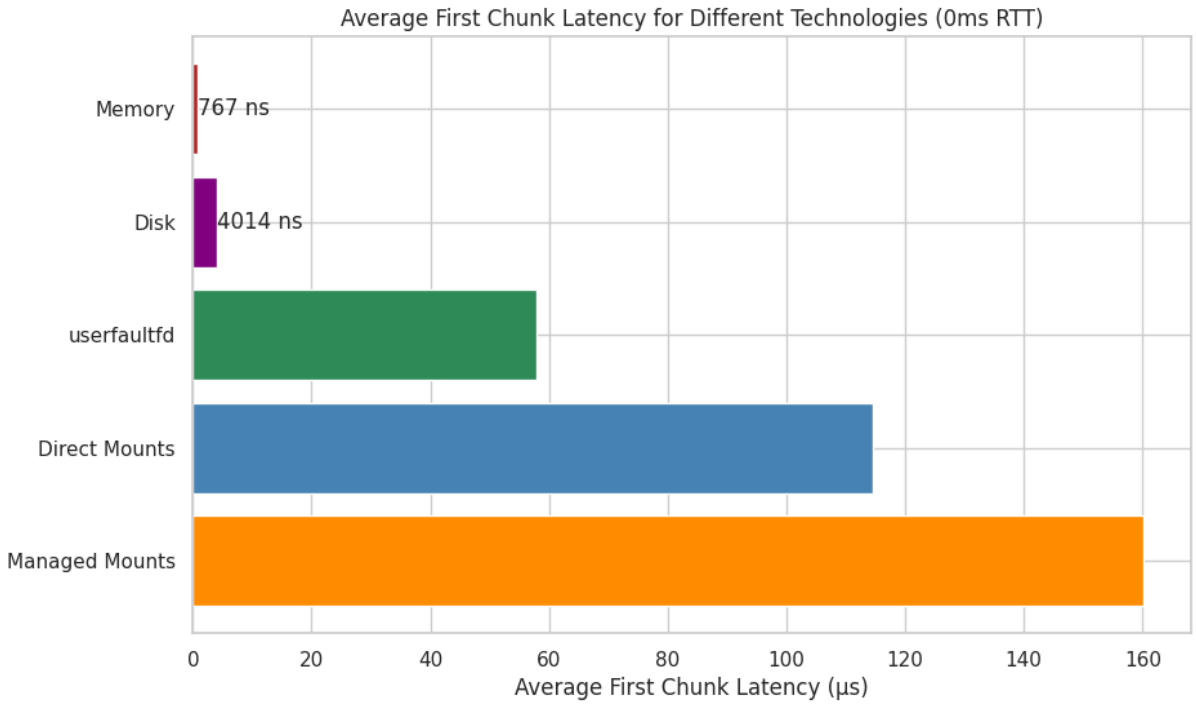

Figure 8: Average first chunk latency for different direct memory access, disk, userfaultfd, direct mounts and managed mounts (0ms RTT)

Compared to disk and memory, all other network-capable access methods (`userfaultfd`, direct mounts and managed mounts) have significantly higher latencies when accessing the first chunk. The latency of `userfaultfd` is 15 times slower than the disk access time, while direct mounts and managed mounts are 28 and 40 times slower respectively. Its is however important to consider that even despite these differences, the overall latency is still below 200 µs for all access methods.

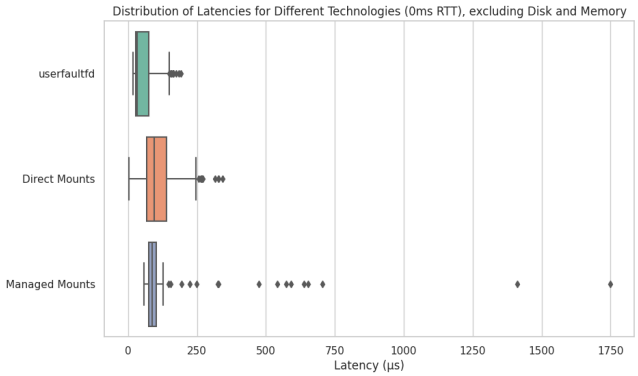

Figure 9: Box plot for the distribution of first chunk latency for userfaultfd, direct mounts and managed mounts (0ms RTT)

When looking at the latency distribution for the network-capable access methods, the spread for the managed mount is the smallest, but there are significant outliers until more than 1 ms; direct mounts have a significantly higher spread, but less outliers, while `userfaultfd`'s latency advantage is visible here too.

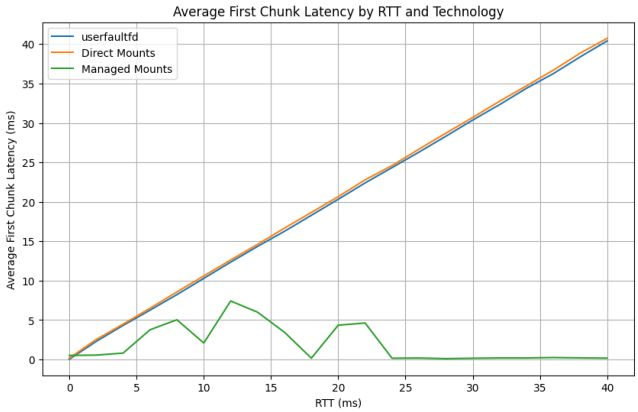

Figure 10: Average first chunk latency for userfaultfd, direct mounts and managed mounts by RTT

For the earlier measurements the backends were connected directly to the mount or `userfaultfd` handler respectively, resulting in an effictive 0ms RTT. If the RTT is increased, the backends behave differently; for `userfaultfd` and direct mounts, the first chunk latency grows linearly. For managed mounts however, the latency is higher than at a 0ms RTT, but even at this peak it is significantly lower than both `userfaultfd` and managed mounts; after the RTT reaches 25ms, the first chunk latency for managed mounts reach latency levels below the measured latency at 0ms RTT.

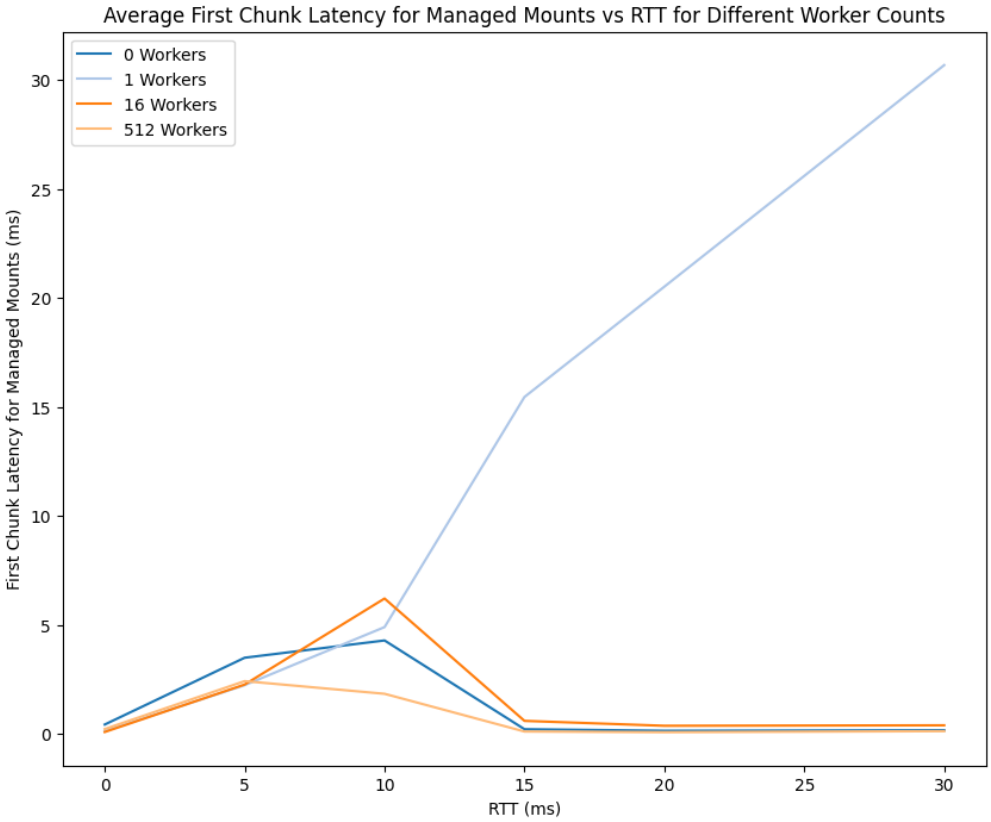

Figure 11: Average first chunk latency for managed workers with 0-512 workers by RTT

A similar pattern can be seen when analysing how different worker counts for managed mounts influence latency; for zero workers, the latency grows almost linearly, while if more than 1 worker is used, the latency first has a peak, but then continues to drop until it reaches a level close to or lower than direct mounts.

5.2.2 Read Throughput

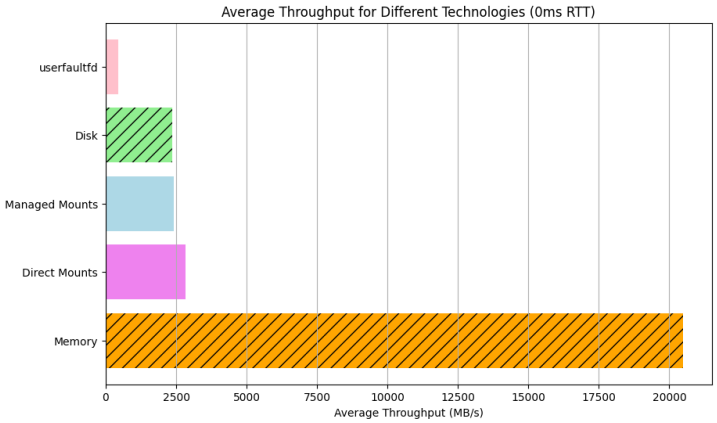

Figure 12: Average throughput for memory, disk, userfaultfd, direct mounts and managed mounts (0ms RTT)

When looking at throughput compared to latency, the trends for memory, disk, `userfaultfd` and the two mount types are similar, but less drastic. Direct memory access still is the fastest option at a 20 GB/s throughput, but unlike with latency is followed not by the disk, but rather by direct mounts at 2.8 GB/s and managed mounts at 2.4 GB/s. The disk is slower than both of these methods at 2.3 GB/s, while `userfaultfd` has the lowest throughput at 450 MB/s.

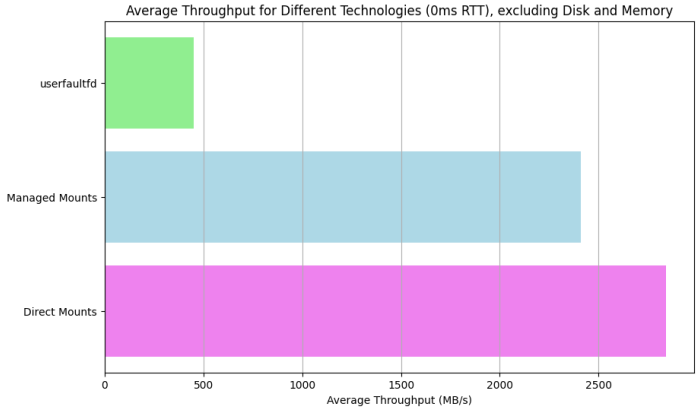

Figure 13: Average throughput for userfaultfd, direct mounts and managed mounts (0ms RTT)

When looking at just the throughput for network-capable access methods, `userfaultfd` falls significantly behind both mount types, meaning that while at 0 RTT, `userfaultfd` has lower latency, but also much lower throughput.

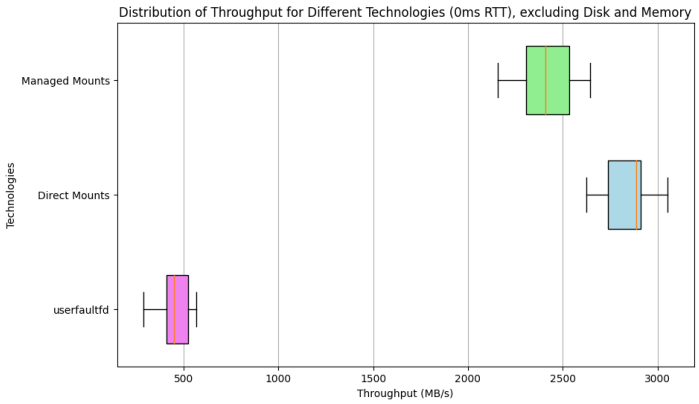

Figure 14: Box plot for the throughput distribuion for userfaultfd, direct mounts and managed mounts (0ms RTT)

When it comes to throughput distribution, `userfaultfd` has the lowest spread while managed mounts has the highest, closely followed by direct mounts. Interestingly, the median throughput of direct mounts is especially high.

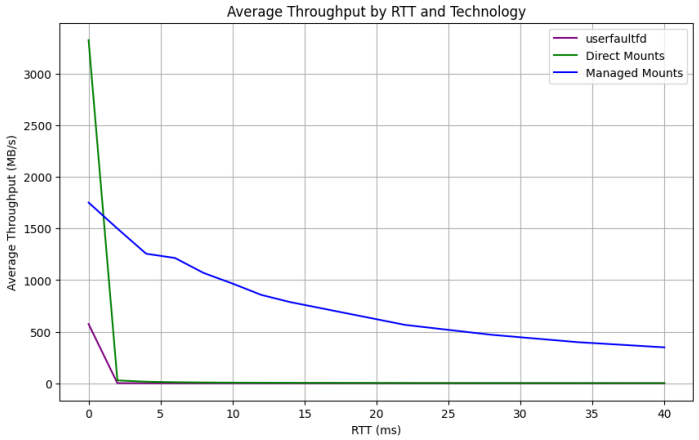

Figure 15: Average throughput for userfaultfd, direct mounts and managed mounts by RTT

As it is the case with latency, the access methods behave very differently as the RTT increases. Direct mounts and `userfaultfd` throughputs drop to below 10 MB/s and 1 MB/s as the RTT reaches 6 ms, and continues to drop as it increases further. The throughput for managed mounts also decreases as RTT increases, but much less drastically compared to the other methods; even at a RTT of 25ms, the throughput is still over 500 MB/s. If the RTT is lower than 10ms, a throughput of almost 1 GB/s can still be achieved with managed mounts.

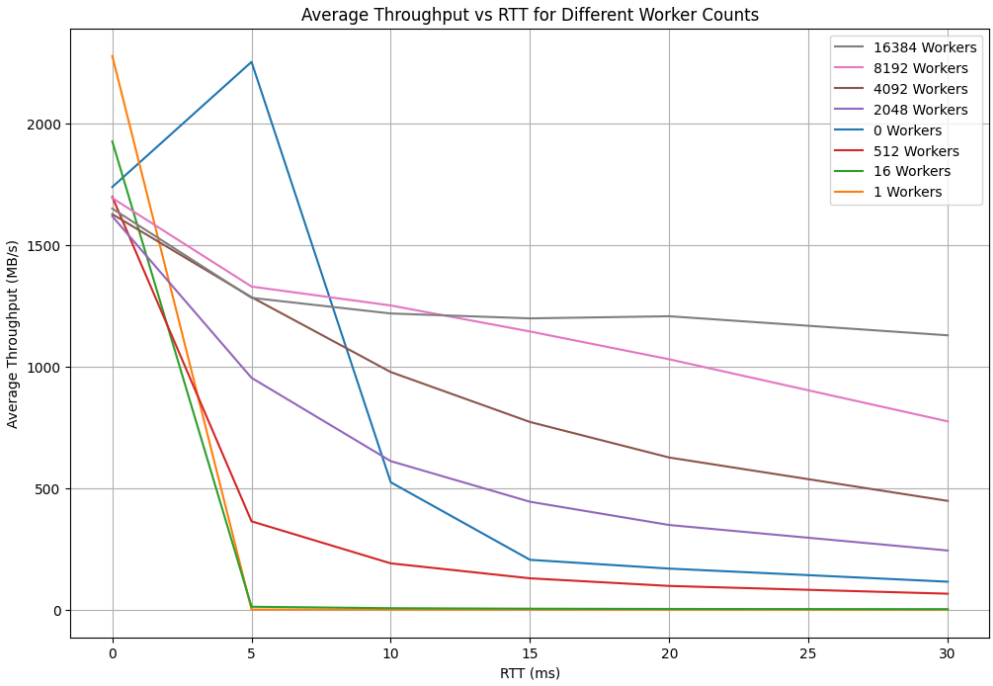

Figure 16: Average throughput for managed mounts with 0-16384 workers by RTT

Similar results the effects of worker counts on latency can be seen when measuring throughput with different configurations as RTT increases. While low worker counts result in good throughput at 0 ms RTT, generally, the higher the worker counts, the higher the achievable throughput. For 16384 workers, throughput can be consistently kept at over 1 GB/s, even at a latency of 30 ms.

5.2.3 Write Throughput

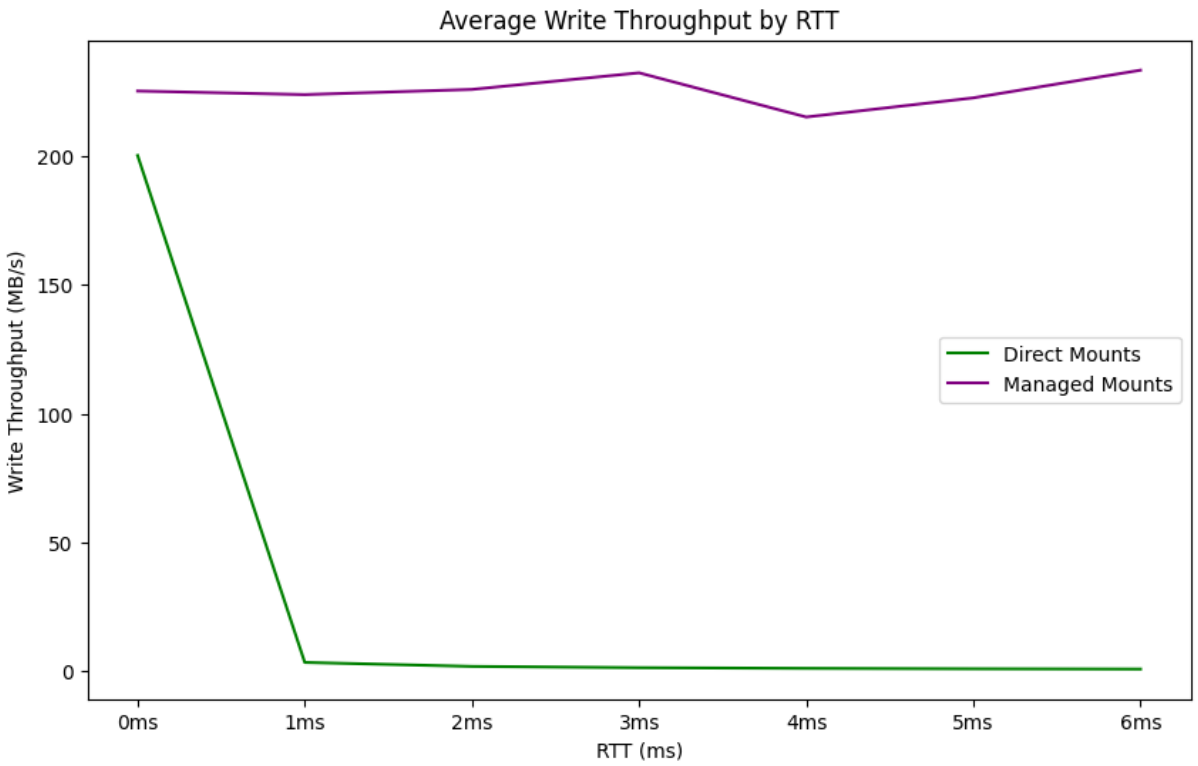

Figure 17: Average write throughput for direct and managed mounts by RTT

While for a migration or mount scenario read throughput is a critical metric, it is also interesting to compare the write throughput of the different access methods. Note that for this benchmark, the underlying block device is opened with `O_DIRECT`, which causes additional overhead when writing due to it skipping the kernel buffer, but is useful for this benchmark specifically as it doesn't require a `sync`/`msync` step to flush data to the disk. As RTT increases, managed mounts show a much better write performance. Write speeds for direct mounts drop to below 1 MB/s as soon as the RTT increases to 4ms, while they are consistenly above 230 MB/s for managed mounts, independent of RTT. `userfaultfd` was not measured, as it does not allow for tracking changes to the memory regions handled by it.

5.3 Initialization

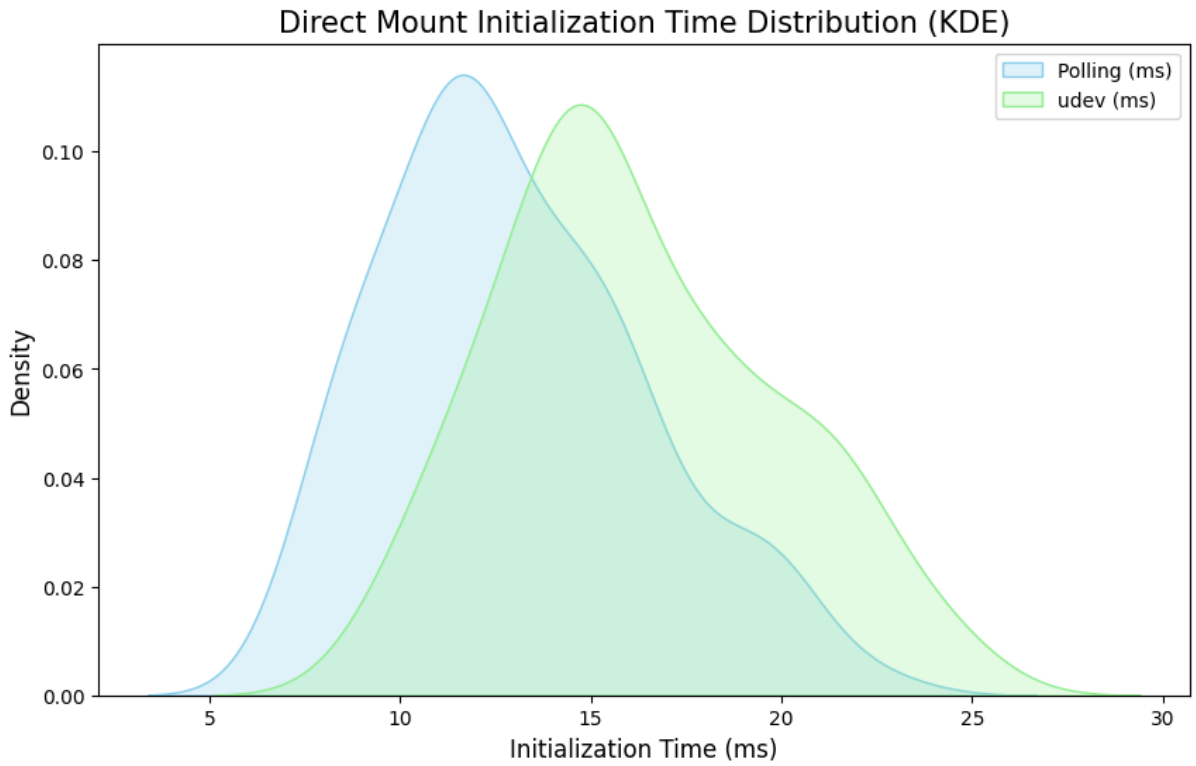

Figure 18: Kernel density estimation for the distribution of direct mount initialization time with polling vs. udev

When it comes to initialization for direct and managed mounts, as introduced earlier, there are two methods of detecting that a NBD device is ready: Polling, or subscribing to `udev` events. While spread for both methods is similarly high, the average initialization time for `udev` is higher than the polling method.

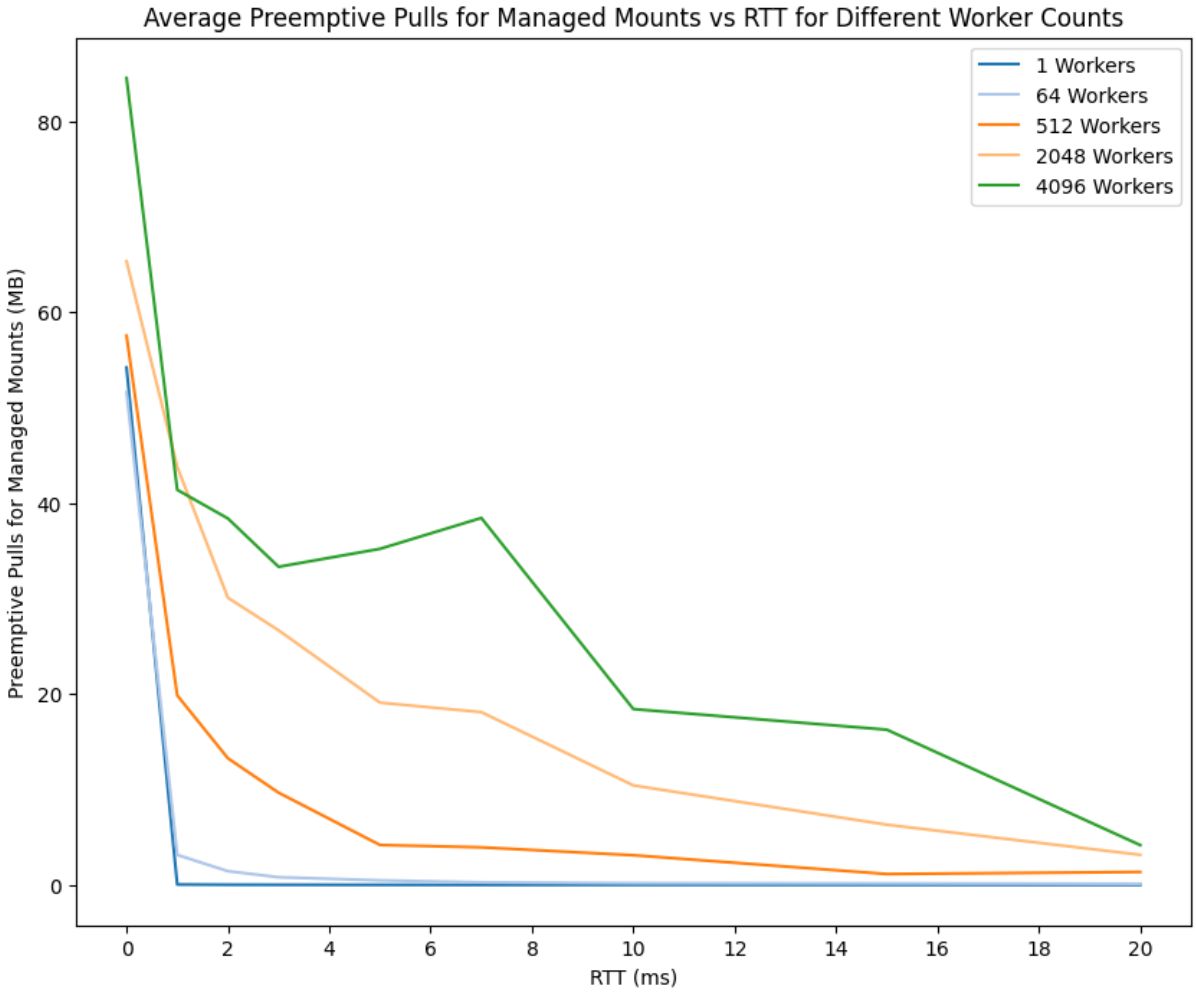

Figure 19: Amount of pre-emptively pulled data for managed mounts with 0-4096 workers by RTT

Another aspect of the device initialization process is the amount of data that can be pulled pre-emptively; here again, the importance of the worker count becomes apparent. The higher the worker count is, the more data can be pulled; while for 4096 workers, almost 40 MB of data can be pulled before the device is opened at a RTT of 7 ms, this drops to 20 MB for 2048 workers, 5 MB for 512 and continues to drop as the worker count decreases. Even for a 0 ms RTT, more background pull workers result in more pre-emptively pulled data.

5.4 Chunking

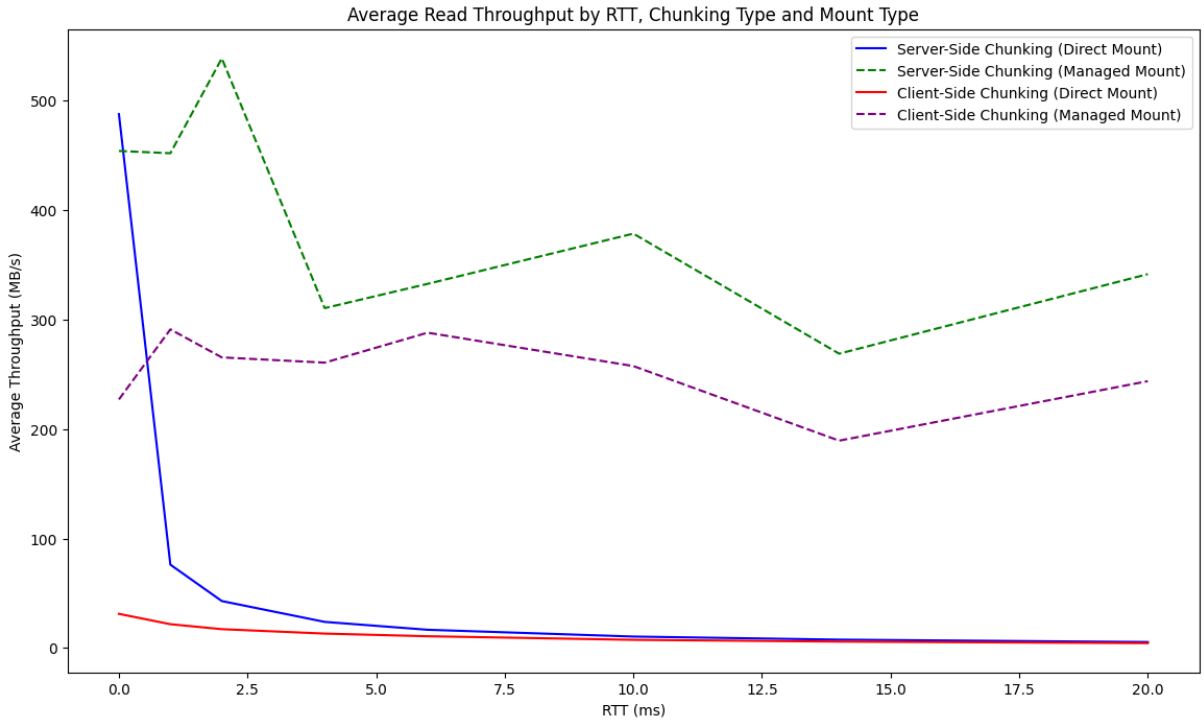

Figure 20: Average read throughput for server-side and client-side chunking, direct mounts and managed mounts by RTT

Chunking can be done on either client- or server-side for both the direct and the managed mounts; looking at throughput for both options, it is clear that unless the RTT is 0 ms, managed mounts yield significantly higher throughput compared to direct mounts for both client- and server-side chunking.

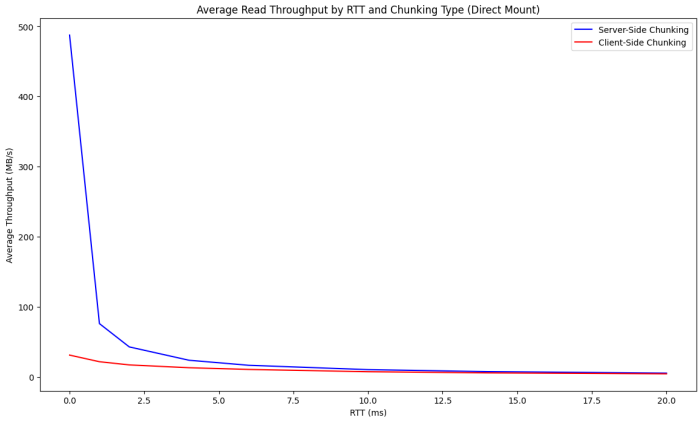

Figure 21: Average read throughput for server-side and client-side chunking with direct mounts by RTT

When looking at direct mounts specifically, server-side chunking is a very fast option for 0 ms RTT at almost 500 MB/s throughput, but drops to just 75 MB/s at a 1 ms RTT, 40 MB/s at 2 ms, and then continues to drop to just over 5 MB/s at 20 ms RTT. For client-side chunking, the throughput is much lower at just 30 MB/s even at 0 ms RTT, after which it continues to drop steadily until reaches just 4.5 MB/s at 20 ms RTT.

Figure 22: Average read throughput for server-side and client-side chunking with managed mounts by RTT

For managed mounts, the measured throughput is different; throughput also decreases as RTT increases, but much less drastically. Server-side throughput also yields higher throughputs for this solution at 450 MB/s at 0 ms RTT vs. 230 MB/s at 0 ms for client-side chunking. As RTT increases, throughput for both direct and managed backends drop to 300 MB/s for managed mounts and 240 MB/s for direct mounts at a RTT of 20 ms.

5.5 RPC Frameworks

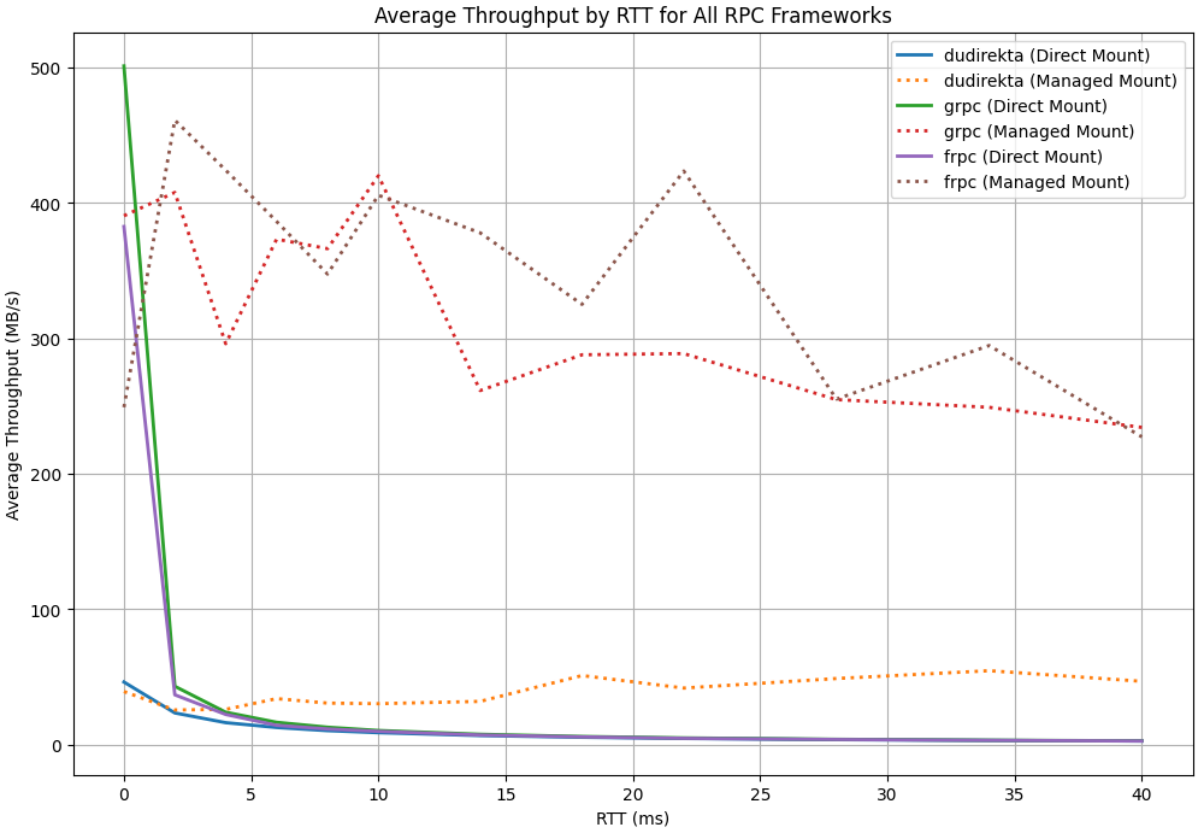

Figure 23: Average throughput by RTT for Dudirekta, gRPC and fRPC frameworks for direct and managed mounts

Looking at the performance for the Dudirekta, gRPC and fRPC frameworks for managed and direct mounts, throughput decreases as the RTT increases. While for 0 ms RTT, direct mounts provide the best throughput in line with the measurements for the different access technologies, it drops more drastically as RTT increases compared to managed mounts. Also apparent is that Dudirekta has a much lower throughput than both gRPC and fRPC.

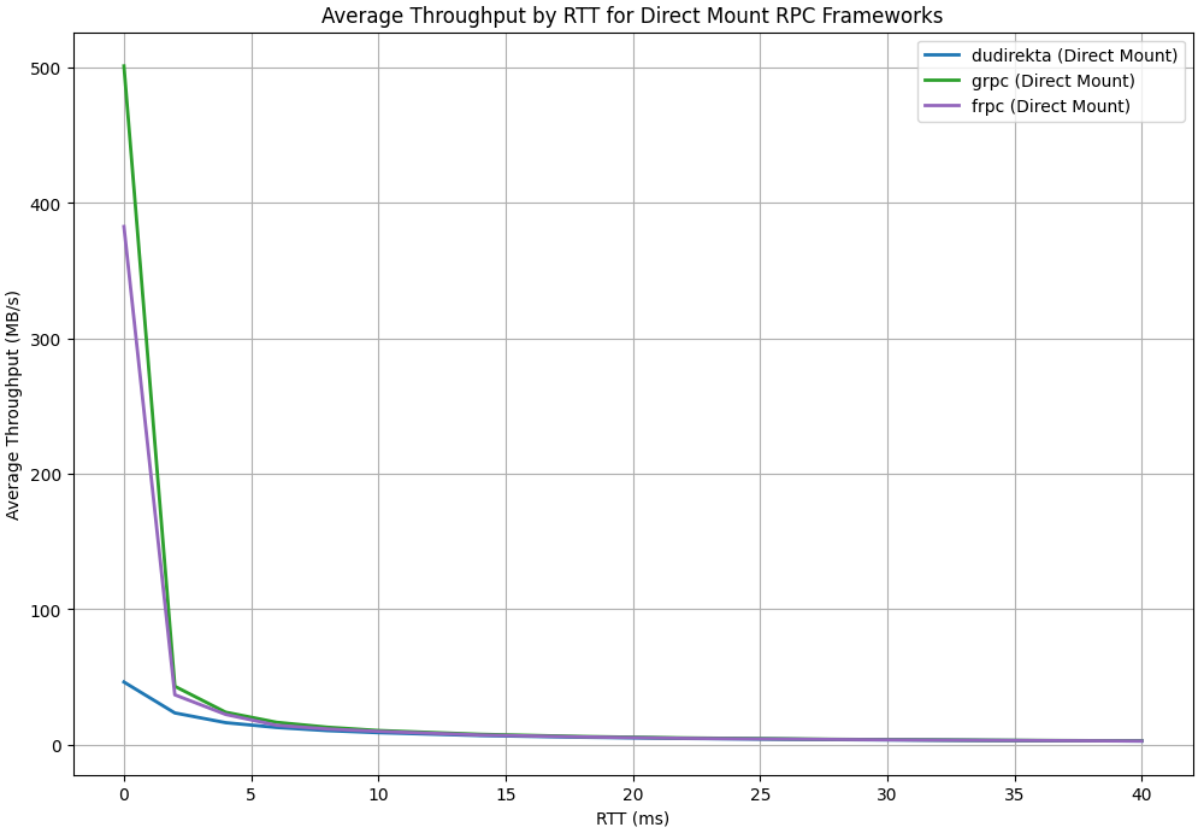

Figure 24: Average throughput by RTT for Dudirekta, gRPC and fRPC frameworks for direct mounts

When looking at direct mounts specifically, the sharp difference between measured throughputs for Dudirekta and gRPC/fRPC is apparent again. While 0 ms RTT, fRPC reaches a throughput of 390 MB/s and gRPC of 500 MB/s, Dudirekta reaches just 50 MB/s. At 2 ms, throughput for all RPC frameworks drop drastically as the RTT increases, with fRPC and gRPC both dropping to 40 MB/s, and Dudirekta dropping to just 20 MB/s. All RPC frameworks ultimately reach a throughput of just 7 MB/s at a RTT of 14 ms, and then continue to decrease until they reach 3 MB/s at 40 ms.

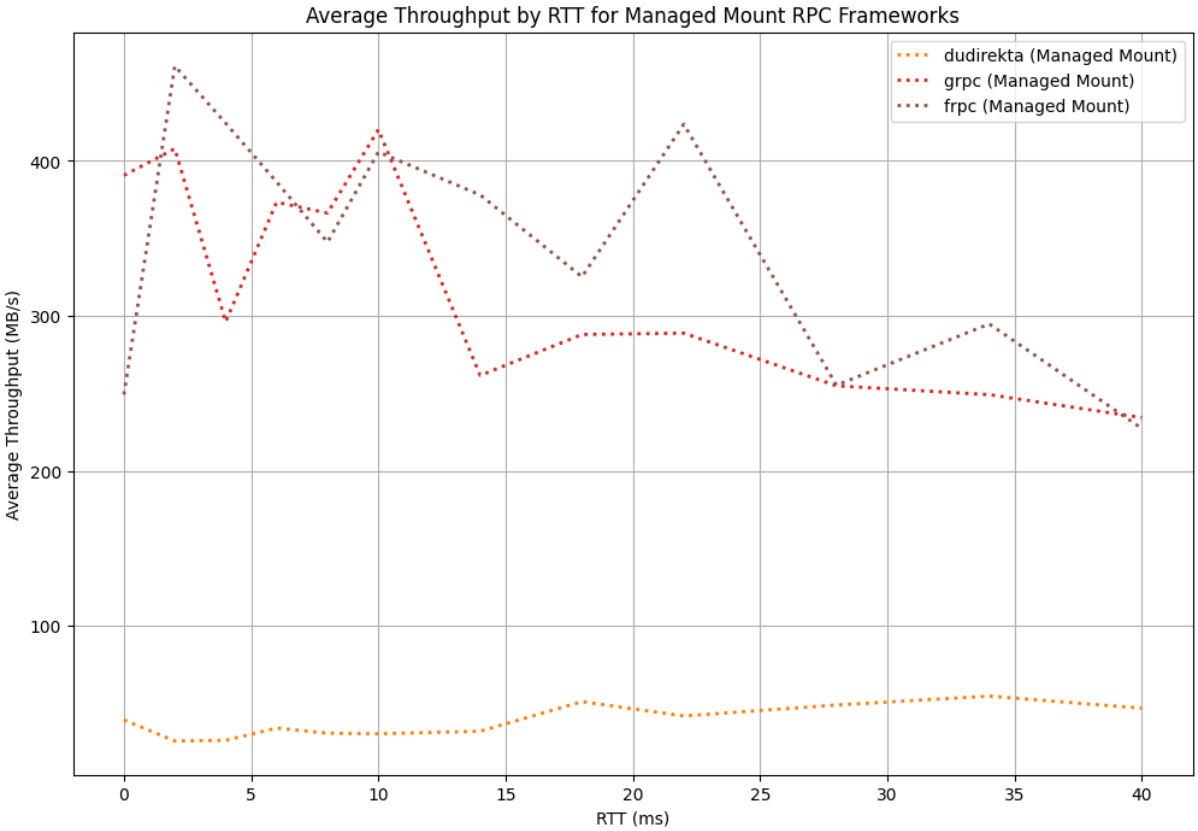

Figure 25: Average throughput by RTT for Dudirekta, gRPC and fRPC frameworks for managed mounts

For managed mounts, Dudirekta stays consistently low compared to the other options at an average of 45 MB/s, but doesn't decrease in throughput as RTT increases. Like for direct mounts, gRPC manages to outperform fRPC at 0 ms RTT at 395 MB/s vs. 250 MB/s respectively, but fRPC gets consistently higher throughput values starting at a 2 ms RTT. fRPC manages to keep a throughput above 300 MB/s until a RTT of 25 ms, while gRPC drops below this after 14 ms. While the average throughput of fRPC is higher, as the RTT reaches 40 ms this difference becomes less pronounced at 50 MB/s after a 28 ms RTT is reached.

5.6 Backends

5.6.1 Latency

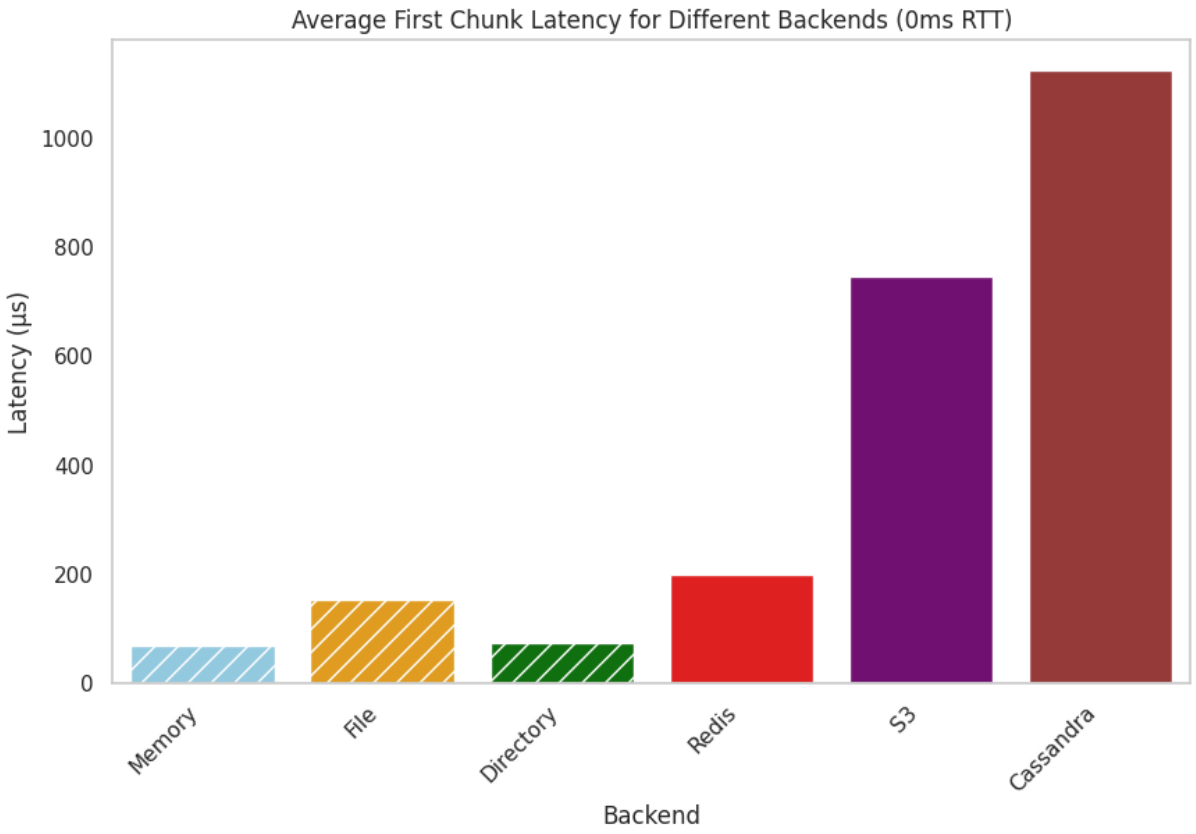

Figure 26: Average first chunk latency for memory, file, directory, Redis, S3 and ScylllaDB backends (0ms RTT)

When it comes to the average first chunk latency for the memory, file, directory, Redis, S3 and ScyllaDB backends, considerable differences between the different backends can be observed. While the overhead of the memory, file and directory backends over the raw access latency of the direct and managed mounts (see figure 8) is minimal, the network-capable backends have a much higher latency. While the values measured for Redis are only marginally higher than those of the file backend, S3 and Cassandra have significantly higher average read latencies, reaching more than a millisecond in the case of the latter.

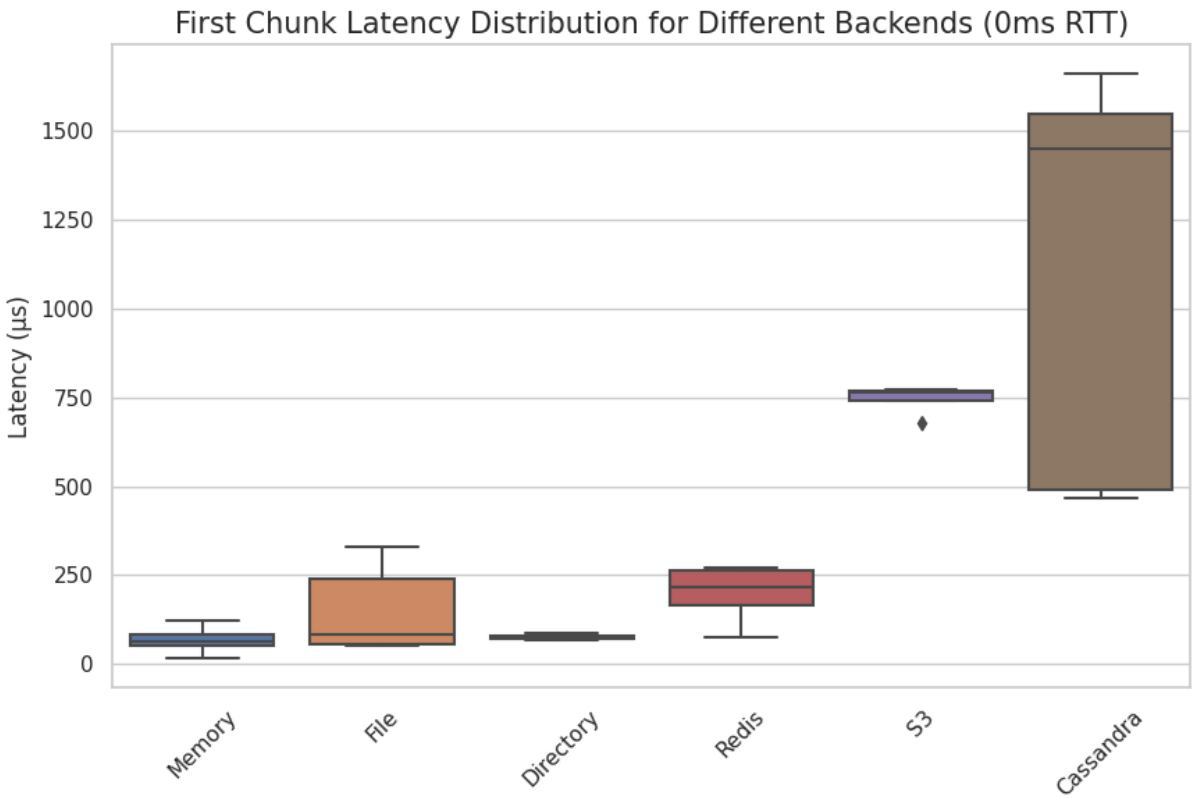

Figure 27: Box plot of first chunk latency distribution for memory, file, directory, Redis, S3 and ScylllaDB (0ms RTT)

When looking at the latency distribution, a significant difference in spread between the backends can be observed. While the memory, directory and S3 backends show only minimal spread albeit at different average latencies, the network-capable Redis backend shows less spread than the file backend. Compared to the other options, the Cassandra backend shows the most amount of spread by far, being characterized by a noticably high median latency, too.

5.6.2 Throughput

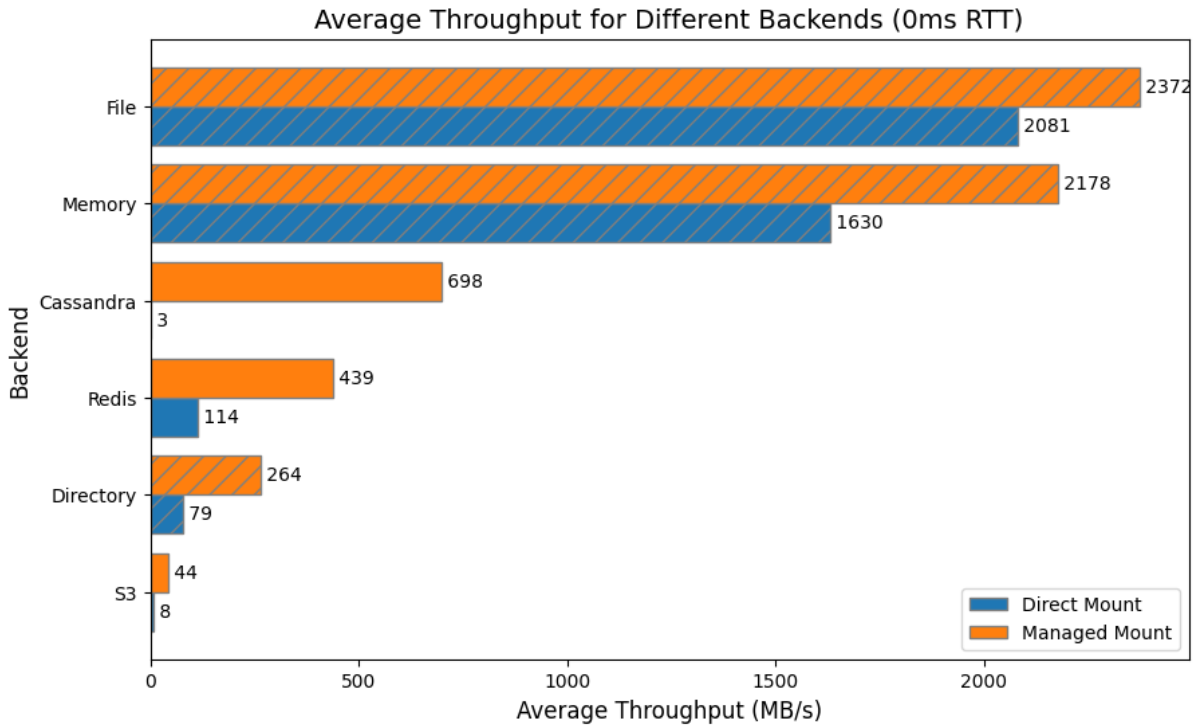

Figure 28: Average throughput for memory, file, directory, Redis, S3 and ScylllaDB backends for direct and managed mounts (0ms RTT)

When looking at throughputs, the backends behave significantly more different compared to latency. Both the file and memory backends have consistently high throughputs; for direct mounts, file throughput is higher than memory throughput at 2081 MB/s vs. 1630 MB/s on average respectively. For managed mounts, this increases to 2372 MB/s vs. 2178 MB/s. When comparing the direct vs. managed mount measurement, ScylllaDB and the network-capable backends in general show vast differences in throughput; while ScylllaDB manages to reach almost 700 MB/s for a managed mount scenario, it falls to only 3 MB/s for a direct mount. Similarly so, for Redis and the directory backend, direct mounts are 3.5 times slower than the managed mounts.

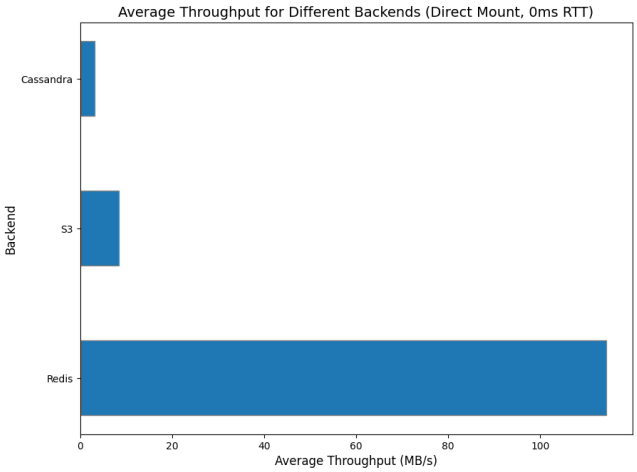

Figure 29: Average throughput for Redis, S3 and ScylllaDB backends for direct mounts (0ms RTT)

For the throughput of network-capable direct mounts, Redis has the highest average throughput at 114 MB/s compared to both ScylllaDB at 3 MB/s and S3 at 8 MB/s.

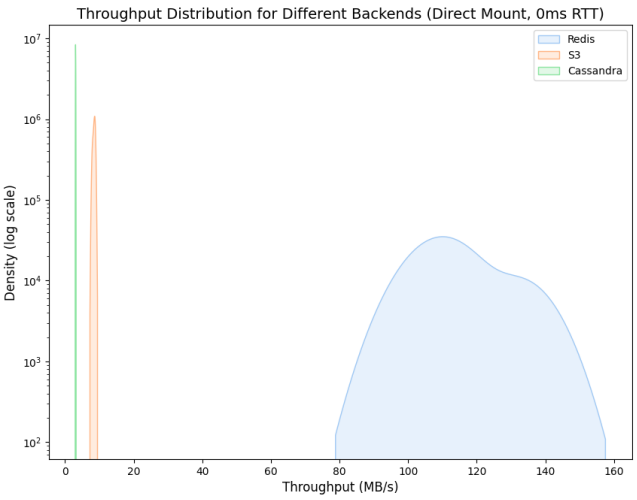

Figure 30: Kernel density estimation (with logarithmic Y axis) for the throughput distribution for Redis, S3 and ScylllaDB for direct mounts (0ms RTT)

The throughput distribution for the different backends with direct mounts shows a similarly drastic difference between Redis and the other options; a logarithmic Y axis is used to show the kernel density estimation, and while Redis does have a far larger spread compared to S3 and ScylllaDB, the throughput is also noticably higher.

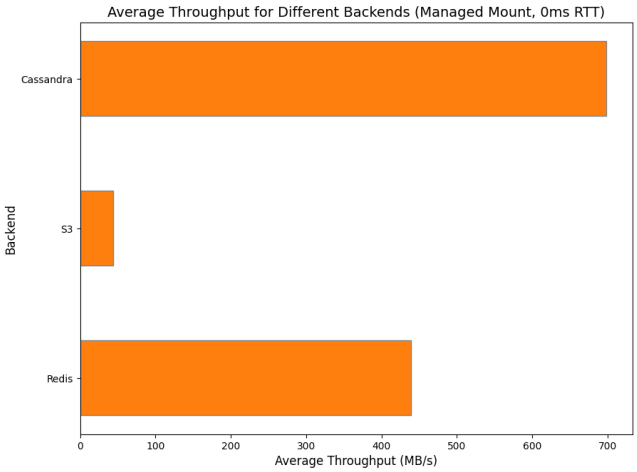

Figure 31: Average throughput for Redis, S3 and ScylllaDB backends for managed mounts (0ms RTT)

For managed mounts, ScylllaDB performs better than both S3 and Redis, managing 689 MB/s to 439 MB/s for Redis and 44 MB/s for S3.

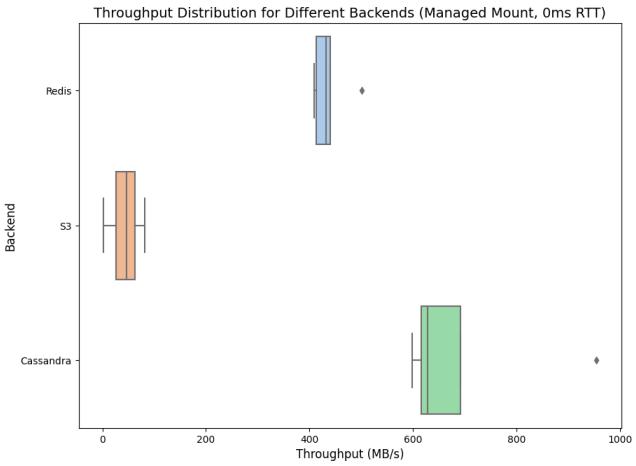

Figure 32: Box plot for the throughput distribution for Redis, S3 and ScylllaDB for managed mounts (0ms RTT)

As for the distribution, ScylllaDB has a high spread but also the highest throughput, while Redis has the lowest spread. S3 is also noticably here with a consistently lower throughput compared to both alternatives, with an average spread.

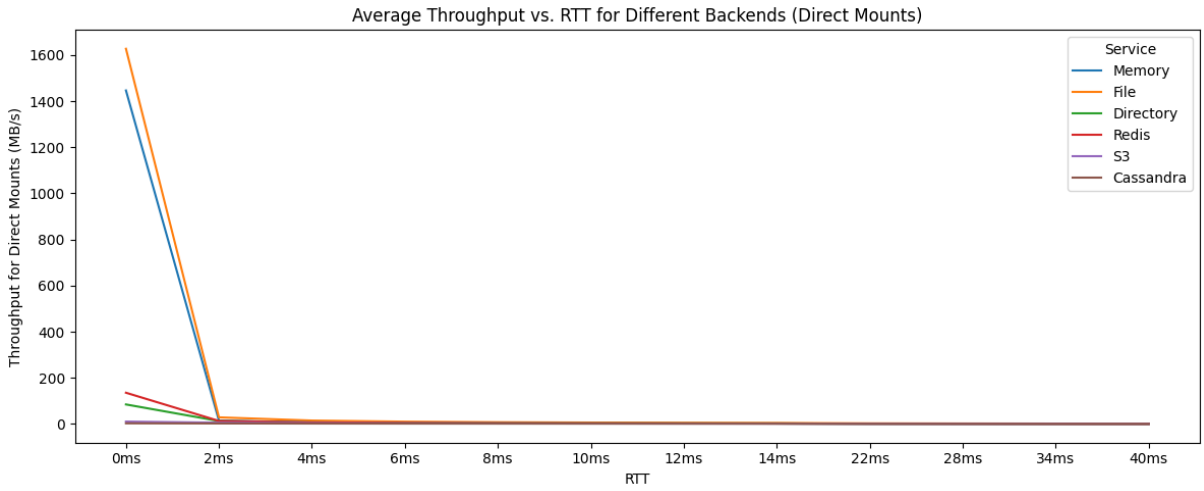

Figure 33: Average throughput for memory, file, directory, Redis, S3 and ScylllaDB backends for direct mounts by RTT

When looking at average throughput for direct mounts, all backends drop in throughput as RTT increases. Memory and file are very fast at above 1.4 GB/s at 0 ms RTT, while Redis achieves 140 MB/s as the closest network-capable alternative. The directory backend noticably has a lower throughput than Redis, despite not being network-capable. All other backends are at below 15 MB/s for direct mounts, even at 0 ms RTT, and all backends trend towards below 3 MB/s at a RTT of 40 ms for direct mounts.

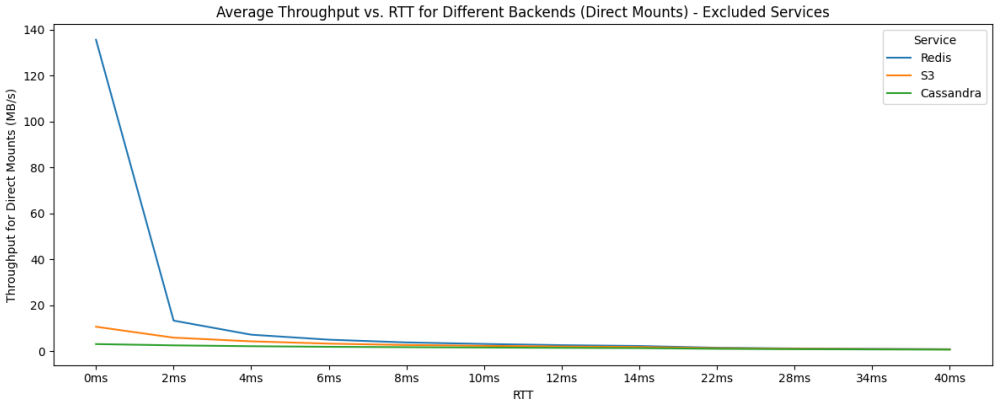

Figure 34: Average throughput for Redis, S3 and ScylllaDB backends for direct mounts by RTT

The network-capable backends in isolation again show the striking difference between Redis and the other backends' direct mount performance, but all generally trend towards low throughput performance as RTT increases.

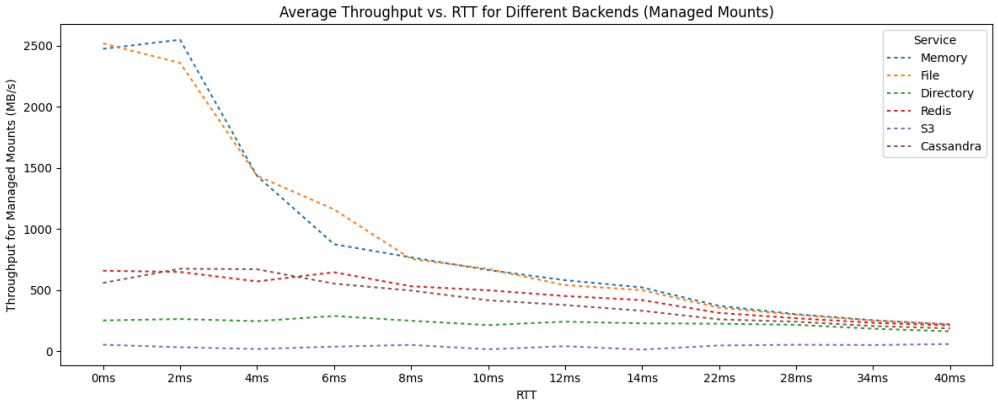

Figure 35: Average throughput for memory, file, directory, Redis, S3 and ScylllaDB backends for managed mounts by RTT

For managed mounts, the memory and file backends outperform all other options at over 2.5 GB/s, while the closest network-capable technology reaches 660 MB/s at 0 ms RTT. Similarly to the latency measurements, all technologies trend towards a similar throughput as RTT increases, with sharp drops for the memory and file backends after the RTT has reached 2 ms. Noticably, both the directory and S3 backends underperform even for managed mounts, with throughput reaching only 55 MB/s at 40 ms RTT.

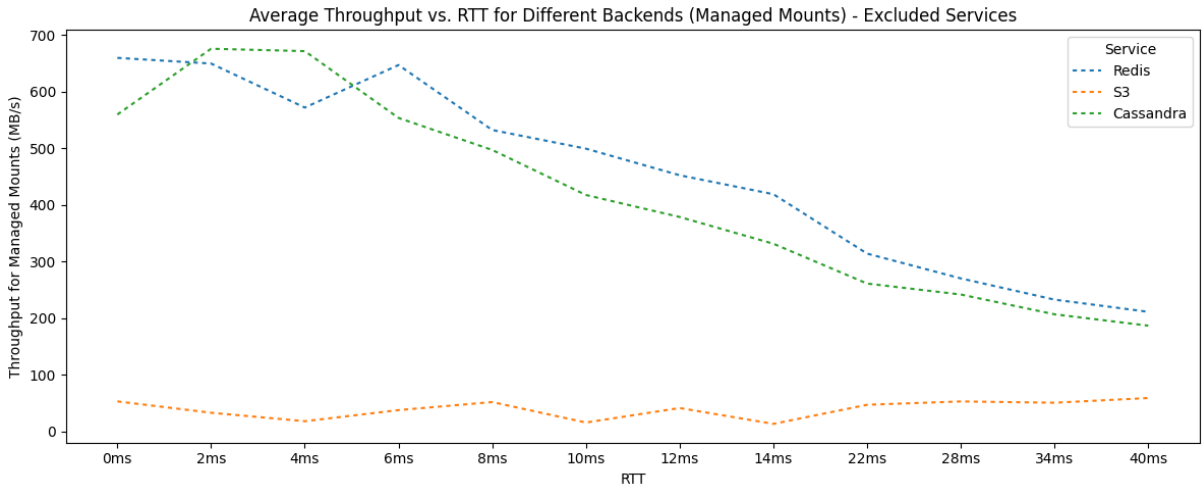

Figure 36: Average throughput for Redis, S3 and ScylllaDB backends for managed mounts by RTT

When looking at just the network-capable backends in isolation, S3's low throughput becomes apparent again. Both Redis and ScylllaDB start between 550-660 MB/s at 0 ms RTT, then begin to drop after 6 ms until they reach 170-180 MB/s at 40 ms, with Redis consistently having slightly higher throughputs compared to ScylllaDB.

6 Discussion

6.1 Userfaults

`userfaultfd` is a simple way to map almost any object into memory. It is a comparatively simple approach, but also has significant architecture-related problems that limit its use. One problem is that it is only able to catch page faults, which means that it can only ever handle a data request the first time a chunk of memory gets accessed, since all future requests to a memory region handled by `userfaultfd` will simply return directly from RAM. This prevents the usage of this approach for accessing remote resources that update over time, and also makes it hard to use it for applications with concurrent writers or shared resources, since there would be no way of updating a section with a conflict.

Due to these limitations, `userfaultfd` is essentially limited to read-only mounts of remote resources, not synchronization. While it could be a viable solution for post-copy migration, it also prevents pullling chunks from being pulled before they have been accessed without adding an additional layer of indirection. The `userfaultfd` API socket is also synchronous, meaning that each page fault for each chunk needs to be handled one after the other, making it very vulnerable to long RTT values. This also results in the initial read latency always being at least the RTT to the backend.

While it has a lower first chunk latency compared to direct mounts and managed mounts at 0 ms RTT (see figure 8), the latency grows linearly as the RTT increases (see figure 10) due to the aforementioned synchronous API socket and no way of pulling data in the background. While it has more predictable throughput and latency than the NBD-based solutions, for high RTT deployments, it becomes essentially unusable due to the low throughput. In summary, while this approach is interesting and idiomatic to Go and Linux, for most data, esp. larger datasets and high-RTT networks like a WAN, one of the alternative solutions is a better choice.

6.2 File-Based Synchronization

Similarly to `userfaultfd`, the approach based on `mmap`ing a memory region to a file and then synchronizing this file also has limitations. While `userfaultfd` is only able to catch the first reads to a file, this system is only able to catch wries, making it unsuitable for post-copy migration scenarios. It makes this system write-only, and very inefficient when it comes to adding hosts to the network at a later point, since all data needs to be continously synced to all other hosts that state could potentially be migrated too.

To work around this issue, a central forwarding hub can be used, which reduces the amount of data streams required from the host currently hosting the data, but also adds other drawbacks such as operational complexity and additional latency. Thanks to this support for the central forwarding hub, file-based synchronization might be a good choice for highly throughput-constrained networks, but the inability to do post-copy migration due to it being write-only makes it a suboptimal choice for migration scenarios.

6.3 FUSE

File systems in user space provide a solution that allows for both pre- and post-copy migration, but doesn't come without downsides. As it operates in user space, it depends on context switching, which does add additional overhead compared to a file system implementation in kernel space. While some advanced file system features like `inotify` and others aren't available for a FUSE, the biggest problem is the development overhead of implementing a FUSE, which requires the implementation of a completely custom file system. The optimal solution for memory synchronization is not to provide an entire file system to track reads and writes on, but instead to track a single file; for this use case, NBD serves as as an existing API providing this simpler approach, making FUSE not the optimal technology to implement memory synchronization.

6.4 Direct Mounts

Direct mounts have a high spread when it comes to first chunk latency at 0 ms RTT(see figure 9), but are more predictable when it comes to their throughput (see figure 14). Similarly to the drawbacks of `userfaultfd`, it's first chunk latency grows linearly as the RTT increases (see figure 10), due to the lack of pre-emptive pulls. Despite this, it has the highest throughput at 0 ms RTT, even higher than managed mounts (see figure 12) due to it having less expensive internal I/O operations because of the lack of this pull system. While compared to `userfaultfd`, its read throughput doesn't drop as rapidly as RTT increases (see figure 15). Its write speed is heavily influenced by RTT (see figure 17) since writes need to be written to the remote as they happen, as there is no background push system either. These characteristics make direct mounts a good access method to choose if the internal overhead of using managed mounts is higher than the overhead caused for direct mounts by the RTT, which can be the case in LAN or other very low-latency networks.

6.5 Managed Mounts

Managed mounts have an internal overhead to due to the duplicate I/O operations required for background pull, resulting in a worse throughput for low RTT scenarios compared to direct mounts (esp. for 0 ms RTT; see figure 12), as well as higher first chunk latencies (see figure 8). As soon as the RTT reaches levels more typical for a WAN environment however, this overhead becomes negligible compared to the benefits gained over the other access methods thanks to the background push and pull systems (see figure 10 and 15).

Tuning the background workers to the specific environment can substantially increase a managed mounts performance (see figure 11 and 16), since data can be fetched in parallel. The pull priority function can allow for even more optimized pulls, and pre-emptive pulls can significantly reduce initial chunk latency since data can be pulled asynchronously before the device is even available. Higher worker counts to increase the amount of data that is being pulled preemptively (see figure 19), as well as preferring polling over the `udev`-based method for detecting device readiness (see figure 18), both can reduce the overall device `Open()` time and thus reduce the initial overhead of using this access method for a remote resource. Write throughput for managed mounts is also significantly higher than for direct mounts (see figure 17), which is the case due to the background push system; writes are first written to the faster local backend, and are then asynchronously written to the remote, resulting in much faster write speeds. These characteristics make managed mounts the perferred access method for WAN environments, where the RTT usually is high enough to balance out the internal overhead caused by the background push and pull systems I/O operations.

6.6 Chunking

In general, server-side chunking should almost always be the preferred technology due to the much better throughput compared to client-side chunking (see figure 20). For direct mounts, due to their linear/synchronous access pattern, the throughput is low for both server- and client-side chunking as RTT increases, but even with linear access server-side chunking still outperforms the alternative (see figure 21). For managed mounts, client-side chunking can still halve the throughput of a mount compared to server-side chunking (see figure 22). If the data chunks are smaller than the NBD block size, it reduces the number of chunks that can be fetched if the number of workers remains the same. This isn't the case with server-side chunking because it doesn't need an extra worker on the client side for each additional chunk that needs to be fetched. This allows the background pull system to fetch more, thus increasing throughput.

6.7 RPC Frameworks

Out of the frameworks tested, Dudirekta consistently has lower throughput than the alternatives (see figure 23). It performs better for managed mounts than direct mounts thanks to its support for concurrent RPCs and is less sensitive to RTT compared to gRPC and fRPC for managed mounts, but even for the latter, its throughput is much lower compared to both alternatives (see figure 25) due to its lack of connecting pooling. Despite these drawbacks however, Dudirekta remains an interesting option for prototyping due to the decreased friction in developer overhead, bi-directional PC support and transport layer independence.

gRPC offers considerably faster throughput compared to Dudirekta for both managed and direct mounts (see figure 23). It has support for connection pooling, giving it a significant performance benefit over Dudirekta for managed mounts (see figure 25), do to it being able to more efficiently pull chunks in the background concurrently. It also has good throughput for 0 ms RTT scenarios, and is essentially an industry standard, resulting in good tooling as well as known scalability characterstics.

While gRPC offers a throughput improvement compared to Dudirekta, fRPC is able to improve on gRPC even further due to its internal optimizations. It is also faster than Dudirekta in both direct and managed mounts (see figure 23) due to its support for connection pooling, and compared to gRPC, is a more lean stack than gRPC and Protocol Buffers, making it more simple and maintainable. As a less commonly used solution however it also has less enterprise testing and tooling available, which means that is a more performant, but also less proven option.

6.8 Backends

Redis is the network-capable backend with the the lowest amount of initial chunk latency at a 0 ms RTT (see figure 26). When used for direct mounts, it has a lower throughput compared to managed mounts (see figure 28), showing good support for concurrent chunk access; it also has the highest thoughput for direct mounts by a significant margin (see figure 29) due to its optimized wire protocol and fast key lookups. It also has good throughput in managed mounts due to these optimizations (see figure 31), making it a good choice for ephemeral data like caches, where quick access times are necessary or the direct mount API provides benefits, i.e. in LAN deployments.

ScylllaDB has the highest throughput for 0 ms RTT deployments for managed mounts, showing a very good concurrent access performance (see figure 31). It does however fall short when it comes to usage in direct mounts, where the performance is worse than any other backend (see figure 29), showing the databases high read latency overhead for looking up individual chunks, which is also backed up by looking at it's initial chunk latency distribution (see figure 27). For managed mounts however, as the RTT increases, it shows only slightly lower performance compared to Redis (see figure 35); as a result, it is a good choice of backend if most data will be accessed concurrently by the managed mounts background pull system, but a bad choice if chunks will be accessed outside of this due to the low direct mount throughput. Another use case where ScylllaDB can potentially be beneficial due to its configurable consistency is storing persistent data in a way that is more dependable than Redis or S3.

S3 has the lowest throughput of all network-capable backends that were implemented for managed mounts (see figure 31). Its performance is consistently low even as RTT increases (see figure 35), presumably due to the high overhead of having to make multiple HTTP requests to retrieve individual chunks, despite performing better than ScylllaDB for a direct mount scenario (see figure 29). S3 remains a good choice of backends if its use is required due to architectural constraints, or if the chance of persistently stored chunks being read outside of the managed mounts background pull system is high, where Casssandra has considerably worse throughput.

6.9 Limitations

While the mount APIs are functional for most use cases, there are performance and usability issues due it being implemented in Go. Go is a garbage collected language, and if the garbage collector is active, it has to stop goroutines. If the `mmap` API is used to access a managed mount or a direct mount, it is possible that the garbage collector tries to manage an object with a reference to the exposed slice, or tries to release memory as data is being copied from the NBD device. If the garbage collector then tries to access the slice, it can stop the goroutine providing the slice in the form of the NBD server, causing the deadlock. One workaround for this is to lock the `mmap`ed region into memory, but this will also cause all chunks to be fetched from the remote into memory, which leads to a high `Open()` latency; as a result, the current workaround for this is to simply start the NBD server in a separate process, so as to prevent the garbage collector from stopping the NBD server and trying to access the slice at the same time. Another workaround for this issue could be to instead use a language without a garbage collector such as Rust, which doesn't allow for the deadlock to occur in the first place.

NBD, the underlying technology and protocol for the mount API, has proven to be fairly performant, but it still could be improved upon to get closer to the performance of other access methods, like raw memory access. One such option is `ublk`(**linux2023ublk?**), which has the potential to significantly improve concurrent access speeds over the socket-based connection between the client and server that NBD uses. It is similar in architecture to NBD, where a user space server provides the block device backend, and a kernel `ublk` driver creates `/dev/ublkb*` block devices not unlike the `/dev/nbd*` devices created by NBD. At the point of this thesis being written however, documentation on this emerging kernel technology is still lacking, and NBD continues to be the standard way of creating block devices in user space.

6.10 Using Mounts for Remote Swap with `ram-dl`

`ram-dl`(**pojtinger2023ramdl?**) is an experimental tech demo built to demonstrate how the mount API can be used. It uses the fRPC mount backend to expand local system memory, enabling a variety of use cases such as mounting a remote systems RAM locally or easily inspecting a remote systems memory contents.

It is based on the direct mount API and uses `mkswap`, `swapon` and `swapoff` to enable the Kernel to page out to the mount's block device:

```
// Create a swap partition on the block device
exec.Command("mkswap", devPath).CombinedOutput()

// Enable paging and swapping to the block device
exec.Command("swapon", devPath).CombinedOutput()

// When the mount is stopped, stop paging and swapping to the block
   device
defer exec.Command("swapoff", devPath).CombinedOutput()
```

`ram-dl` exposes two commands that achieve this. The first, `ram-ul` exposes RAM by exposing a memory, file or directory-based backend using a fRPC endpoint. `ram-dl` itself then connects to this endpoint, starts a direct mount and sets the block device up for swapping. While this system is intended mostly as a tech demo and, due to latency and throughput limitations, is not intended for critical deployments, it does show how simple using the r3map API can be, as the entire project consistents of under 300 source lines of code, most of which is argument handling and boilerplate around configuration.

6.11 Mapping Tape Into Memory With `tapisk`

6.11.1 Overview

`tapisk`(**pojtinger2023tapisk?**) is a tool that exposes a tape drive as a block device. While seemingly unrelated to memory synchronization, it does serve as an interesting use case due to the similarities to STFS (mentioned earlier in the FUSE section), which exposed a tape drive as a file system, and serves as an interesting example for how even seemingly incompatible backends can be used to store and synchronize memory.

Using a tape drive as such a backend is challenging, since they are designed for linear access and don't support random reads, while block devices need support for reading and writing to arbitrary locations. Tapes also have very high read/write latencies due to slow seek speeds, taking up to more than a minute depending on the offset of the tape that is being accessed. Due to the modularity of r3map's managed mount API however, it is possible to work around these issues, and make the tape appear as a random-access block device.

6.11.2 Implementation

To achieve this, the background writes and reads provided by the managed mount API can be used. Using these, a faster storage backend (i.e. the disk) can be used as a caching layer, although the concurrent push/pull system can't be used due to tapes only supporting synchronous read/write operations. By using the managed mount, writes are de-duplicated and both read and write operations can become asynchronous, since both happen on the fast local backend first, and the background sync system them handles either periodic writebacks to the tape for write operations or reading a chunk from the tape if it is missing from the cache.

Since chunking works differently for tapes than for block devices, and tapes are append-only devices where overwriting a section prior to the end would result in all following data being overwritten, too, an index must be used to simulate the offsets of the block device locations to their physical location on the tape, which the `bbolt` database is used for. In order to make non-aligned reads and writes to the tape possible, the existing `ArbitraryReadWriter` system can be used. When a chunk is then requested to be read, `tapisk` looks up the physical tape record for the requested offset, and uses the accelerated `MTSEEK` ioctl to seek to the matching record on the tape, after which the chunk is read from the tape into memory:

```
func (b *TapeBackend) ReadAt(p []byte, off int64) (n int, err error) {
    // Calculating the block for the offset
    block := uint64(off) / b.blocksize

    // Getting the physical record on the tape from the index
    location, err := b.index.GetLocation(block)
    // ...

    // Creating the seek operation
    mtop := &ioctl.Mtop{}
    mtop.SetOp(ioctl.MTSEEK)
    mtop.SetCount(location)

    // Seeking to the record
    syscall.Syscall(
        syscall.SYS_IOCTL,
        drive.Fd(),
        ioctl.MTIOCTOP,
        uintptr(unsafe.Pointer(mtop)),
    )
    // ...

    // Reading the chunk from the tape into memory
    return b.drive.Read(p)
}
```

Conversely, in order to write a chunk to the tape, `tapisk` seeks to the end of the tape (unless the last operation was a write already, in which case the tape must be at the end already). Once the seek has completed, the current physical record is requested from the tape drive, and stored as the record for the block that is to be written in the index, after which the chunk is written to the tape. This effectively makes it possible to overwrite existing chunks despite the tape being append-only, since subsequent writes to the same chunk result in the changes being written to the end of the file with the index referencing the new physical location, but does come at the cost of requiring defragmentation to clean up prior iterations of chunks.

6.11.3 Evaluation

`tapisk` is a unique application of r3map's technology, and shows how flexible it is. By using this index, the effectively becomes tape a standard `ReadWriterAt` stage (and `go-nbd` backend) with support for aligned-reads in the same way as the file or directory backends, and thanks to r3map's pipeline design, the regular chunking system could be reused, unlike in STFS were it had to be built from scratch. By re-using the universal RPC backend introduced earlier, which can give remote access to any `go-nbd` backend over a RPC library like Dudirekta, gRPC or fRPC, it is also possible to access a remote tape this way, i.e. to map a remote tape library robot's drive to a system over the network.

Being able to map a tape into memory without having to read the entire contents first can have a variety of use cases. Tapes can store a large amount of data, in the case of LTO-9, up to 18TB on a single tape(**lto2020gen9?**); being able to access such a large amount of data directly in memory, instead of having to work with tooling like `tar`, can significantly improve developer experience. In addition to making it much easier to access tape drives, `tapisk` can also serve as a replacement for LTFS. LTFS is a custom file system implemented as a kernel module, which allows for mounting a tape. If a `tapisk`-provided block device is formatted with a filesystem such as EXT4 or Btrfs, it can also be mounted locally, allowing the tape to be mounted as a file system as well, with the added benefit of also being able to support any filesystem that supports block devices as their backend. Compared to the LTFS approach, this results in a much more maintainable project; while LTFS is tens of thousands of kernel-level source lines of code, `tapisk` achieves effectively the same use case with just under 350.

6.12 Improving Cloud Storage Clients

6.12.1 Existing Solutions

r3map can also be used to create mountable remote filesystems with unique advantages over existing solutions. Currently, there are two main approaches to implementing cloud storage clients. Dropbox and Nextcloud are examples of a system that listens to file changes on a folder and synchronizes files as changes are detected, similarly to the file-based memory region synchronization approach discussed earlier. The big drawback of this approach is that everything that should be available needs to be stored locally; if a lot of data is stored in the cloud drive, it is common to only choose to synchronize a certain set of data to the local host, as there is no way to dynamically download files as they are being accessed. Read and write operations on such systems are however very efficient, since the system's file system is used and any changes are written to/from this file system asynchronously by the synchronization client. This approach also makes offline availability easy, as files that have been synchronized to the local system stay available even if network connectivity has been lost.

The other currently used option is to use a FUSE, i.e. `s3fs`(**gaul2023s3fs?**), which allows for files to be fetched on demand, but comes with a heavy performance penalty. This is the case because most implementations, if a write or read request is sent from the kernel to the FUSE, remote writes or reads happen directly over the network, which makes this approach very sensitive to networks with a high RTT. Offline usage is also usually not possible with a FUSE-based approach, and features such as `inotify`, symlinks etc. are hard to implement, leaving two imperfect solutions to implementing a cloud storage client.

6.12.2 Hybrid Approach

Using r3map makes it possible to get the benefits of both approaches by not having to download any files in advance and also being able to write back changes asynchronously, as well as being able to use almost any existing file system with its complete feature set. Files can also be downloaded preemptively to allow for offline access, just like with the approach that listens to file changes in a directory.

This is possible by once again using the managed mount API. The block device is formatted using a valid filesystem, i.e. EXT4, and then mounted on the host. By configuring the background pull systems workers and pull priority function, it is possible to also download files for offline access, and files have not yet been downloaded to the local system can be pulled from the remote backend as their chunks are being accessed. If a chunk is available locally, reads are also much faster than they would be with a FUSE implementation, and since writes are made to the local backend first, and then being synchronized back to the remote using the remote push system, the same applies to the writes too. Furthermore, by using the migration API, it is possible to migrate the file system between two hosts in a highly efficient way.

By combining the advantages of both approaches into a hybrid one, it is possible to bridge the gap between them, showing that memory synchronization technology like r3map can be used to not only synchronize memory regions, but other state too, including disks.

6.13 Universal Database, Media and Asset Streaming

6.13.1 Streaming Access to Remote Databases

Another use case that r3map can be used for is accessing a remote database locally. While using a database backend (such as the ScyllaDB backend introduced earlier) is one option of storing the chunk, this use case is particularly interesting for file-based databases like SQLite that don't define a wire protocol. Using r3map, instead of having to download an entire SQLite database before being able to use it, it can instead be mounted with the mount API, which then fetches the necessary offsets from a remote backend storing the database as they are being accessed. For most queries, not all data in a database is required, especially if indexes are used; this makes it possible to potentially reduce the amount of transfered data by streaming in only what is required.

Since reads are cached using the local backend with the managed mount API, only the first read should potentially have a performance impact (if it has not been pulled first by the background pull system); similarly so, since writes are written to the local backend first, and then asynchronously written back, the same applies to them as well. Moreover, if the location of i.e. indexes within the SQLite database is known, a pull heuristic can be specified to fetch these first to speed up initial queries. Thanks to the managed mount API providing a standard block device, no changes to SQLite are be required in order for it to support such streaming access; the SQLite file could simply be stored on a mounted file system provided by the mount's block device.

6.13.2 Making Arbitrary File Formats Streamable

In addition to making databases streamable, r3map can also be used to access files in formats that usually don't support being accessed before they are fully available locally. One such format is MP4; usually, if a user downloads a MP4 file, they can't start playback before the file is available locally completely. This is because MP4 typically stores metadata at the end of the file.

The reason for this being stored at the end is usually that the parameters required for this metadata requires encoding the video first. This results in a scenario where, assuming that the file is downloaded from the first to the last offset, the client needs to wait for the file to be completely accessible locally before playing it. While MP4 and other formats supports ways to encode such metadata in the beginning or once every few chunks in order to make them streamable, this is not the case for many already existing files and comes with other tradeoffs(**adobe2020mp4atom?**).

By using r3map however, the pull heuristic function can be used to immediately pre-fetch the metadata, independently of where it is placed; the rest of the chunks can then be fetched either by using the background pull system and/or ad-hoc as they are being accessed. Similarly to the approach used to stream in remote databases, this does not require any changes to the media player being used, since the block device providing the resource can simply be mounted as a file system and thus be used transparently.

6.13.3 Streaming App and Game Assets

Another streaming use case relates to the in-place streaming of assets. Usually, a game needs to be fully downloaded before it is playable; for many modern high-budget titles, this can be hundreds of gigabytes of data, resulting in very long download times even on fast internet connections. Usually however, not all assets need to be downloaded before the game can be played; only some of them are, i.e. the launcher, UI libraries or the first level's assets. While theoritically it would be possible to design a game engine in such a way that assets are only fetched from a remote as they are being required, this would require extensive changes to most engine's architecture, and also be hard to port back to existing titles; furthermore, current transparent solutions that can fetch in assets (i.e. mounting a remote NBD drive or FUSE) are unlikely to be viable solutions considering their high sensitivity to network latency and the high network throughput required for streaming in these assets.

By using the managed mount API to stream in the assets, the overhead of such a solution can be reduced, without requiring changes to the game or its engine. By using the background pull system, reads from chunks that have already been pulled are almost as fast as native disk reads, and by analyzing the access pattern of an existing game, a pull heuristic function can be generated which preemptively pulls the game assets that are loaded first, keeping latency spikes as low as possible. By using the callbacks for monitoring the pull progress provided by the managed mounts, the game can also be paused until a certain local chunk availability is reached in order to prevent latency spikes from missing assets that would need to be fetched directly from the remote, while still allowing for faster startup times.

This concept is not limited to games however, and could also be applied to launching any application. For many systems, completely scanning a binary or script into memory isn't required for it to start execution; similarly to the situation of game engines, adding streaming support would require changes to the interpreters or VMs, since they don't provide a streaming API out of the box aside from being able to read files from the filesystem. With the managed mount API, this existing interface can be reused to add streaming support to these systems by simply pointing them to a filesystem provided by the mount's block device, or, if the interpreter/VM supports it, `mmap`ing the block device directly and executing the resulting memory region.

6.14 Universal App State Mounts and Migrations

6.14.1 Modelling State

Synchronization of app state is a fairly complex problem, and even for simple scenarios, a custom protocol is built for simple apps. While it is possible to use real-time databases like Firebase to synchronize some application states, it and similar solutions to it are usually limited in which data structures they can store and require specific APIs to synchronize them. Usually, even for a simple migration of state between two hosts, synchronization requires state to be manually marshalled, sent over a network, received on a destination host, and unmarshalled. This requires a complex synchronization protocol, and decisions such as when to synchronize state and when to start pulling from the remote need to be made manually, resulting in a database on a third host being used even for simple migrations from one host to another. Almost all of these data structures can ultimately be represented by a byte array; by allocating them from a slice `mmap`ed by r3map, we can use the managed mount, direct mount or migration APIs to implement a universal way of both synchronization and migration of application state, without having to implement a custom protocol.

6.14.2 Mounting State

By allocating all structures on r3map's provided `mmap`ed byte slice, many interesting use cases become possible. For example, a TODO app could use it as its backend. Once loaded, the app mounts the TODO list as a byte slice from a remote server using the managed mount API; since authentication is pluggable and i.e. a database backend like ScylllaDB with a prefix for this user provides a way to do both authentication and authorization, such an approach can scale fairly well. Using the preemptive background pull system, when the user connects, they can start streaming in the byte slice from the remote server as the app is accessing it, but also pull the majority of the required data first by using the pull heuristic function. If the TODO list is modified by changing it in the `mmap`ed memory region, the changes are asynchronously written back to the underlying block device, and thus to the local backend, where the asynchronous writebacks can sync them back to the remote. If the local backend is persistent, i.e. file-based, such a system can even survive network outages.

6.14.3 Migrating State

In addition to using managed mounts to access remotely stored application state, migration of arbitrary app state also becomes a posibility. If a user has a TODO app running on a host like their smartphone, but wants to continue writing a task description on their desktop system, they can migrate the app's state directly and without a third party/remote database by using r3map. For this use case, the migration API can be used. In order to optimize the migration, the pre-copy phase can be started automatically, i.e. if the phone and desktop are physically close to each other or in the same network; in such a LAN migration case, the process is able to benefit from low latencies and high throughputs. It is also possible to integrate the migration API deeply with system events, i.e. by registering a service that migrates applications off a system before a shutdown procedure completes.

6.14.4 Migrating Virtual Machines

It is important to note that there are a few limitations with synchronizing and migrating an application's internal stateful data structures this way; locking is not handled by r3map and would need to be done using a higher-level protocol; moreover, this assumes that the in-memory representation of the data structure is consistent across all, something which is not necessarily the case with programming languages such as Go with multiple processor architectures being involved. While projects such as Apache Arrow(**apache2023arrow?**) allow for application state to represented in a language and CPU architecture independent way, this comes with some of the same restrictions on which state can be synchronized as with other solutions such as Firebase.

In order to keep the possibility of migrating arbitrary state, but also allow for cross-architecture compatibility, VMs can be used. Keeping with the TODO app example, if the resulting app is compiled to Wasm, instead of having to allocate all memory that is to be synchronized from the r3map-provided `mmap`ed byte slice, it is possible to instead simply synchronize the Wasm VM's linear memory as a whole, which also allows storing the entire app's state on a remote as well as migrating an entire app. Similarly so, the app's binary, mounted WASI filesystems etc. could all be synchronized this way, too. Thanks to the preemptive pull implementation outlined earlier, the VM startup and device initialization can also be parallized to allow for shorter latencies while resuming the VM.

This capability is not limited to Wasm VMs however; rather, it is possible to add these features to almost any hypervisor or virtual machine that supports mapping an application's/virtual machine's state to a block device or memory region, essentially adding the capability to suspend/resume and migrate any application in the same way that is possible today over WAN, without requiring any or only minimal changes to the applications themselves.

7 Summary

As is evident from the discussion, there are multiple ways and configurations for implementing a solution for universally accessing, synchronizing and migration memory regions efficiently, while each configuration has different strenghts and weaknesses as shown by the benchmarks, making them each suitable for different use cases.

When it comes to access methods, `userfaultfd` is an interesting API that is idiomatic to both Linux in as a first-party solution and Go due to its fairly low implementation overhead. This approach however falls short when it comes to throughput, especially when used in WAN, where other options can provide better performance. The delta synchronization method for `mmap`ed files provides a simple way of memory synchronization for specific scenarios, but does have a very significant I/O and compute overhead due to its polling and hashing requirements that make it unsuitable for most applications; similarly so, FUSE provides an extensive API for implementing a complete file system in user space, but has significant implementation overhead making it a suboptimal choice for memory synchronization. Direct mounts provide an access method for LAN deployment scenarios, where networks typically have low latency and the lack of I/O overhead compared to other methods makes it a compelling choice, while managed mounts are the preferred access method for WAN environments. This is due to their efficient use of background push and pull, making it possible for them to adapt to the high latencies typical for such deployments, despite having slightly higher internal I/O overhead compared to direct mounts. For most real-world applications, the mount and migration APIs provide a fast and reliable way of achieving a truly universal method of working with remote memory.

As for RPC framework and transport choice, most production environments are well-suited for both fRPC and gRPC as a high-performance offering, where fRPC can offer slightly better average throughput, compared to gRPC's better developer tooling as a result of it's longstanding legacy. For backend choice, the file backend provides a good option for memory migration and synchronizations, as it can provide a performant, reliable and persistent way of storing a memory region without using up too much host memory. For memory access use cases, Redis shows aconsistently strong throughput in both managed and direct mount scenarios due to its concurrency optimizations, especially if ephermeral data is accessed in LAN environments, while Cassandra (and ScyllaDB) provide a good option for applications using managed mounts that need strong concurrency guarantees. These different approaches show that is possible to adapt the necessary semantics for accessing, synchronizing and migrating resources backed by memory regions to a wide variety of backends, wire protocols and APIs.

8 Conclusion

The proposed solution consisting of the direct mount, managed mount and migration APIs as implemented in the form of the r3map library present a efficient method of accessing, synchronizing and migrating remote memory regions over a network, with example use cases and benchmarks showing that r3map is able to provide both throughput and latency characteristics that make it possible to use as part of applications today.

ram-dl demonstrates how minimal the implementation overhead is by implementing a system to share and mount a remote system's memory in under 300 source lines of code, while tapisk shows that the APIs can be used to efficiently map almost any resource, including a linear-access tape drive, to the concepts provided. Aside from these examples, the solution also makes many entirely new use cases that were previously thought of as extraodinarly hard to achieve possible, such as file synchronization that can combine the benefits of NBD with those of existing cloud storage clients, allowing to stream remote databases without requiring changes to their architecture, making arbitrary file formats streamable and optimizing app and game asset downloading processes.

While there are limitations with the proposed solution's underlying technologies, these do provide future research opportunities. For example, the use of Rust as a language that is garbage collection-free could be studied as an option to further increase throughput, fix encountered deadlock issues and reduce overall resource usage, and exploring emerging alternatives for creating block devices to NBD in user space such as ublk could help further improve the implementation presented.

Despite these limitations, the promise of providing a truly universal way of working with remote memory, without having to signficantly change existing applications or hypervisors, is provided in the form of the reference implementation. It is also able to provide multiple specialized configurations for LAN and WAN environments, making it possible to use remote memory technology in completely different and much more dymanic environments than before. As a result, entirely new ways of thinking about application architecture and lifecycles become possible, which can help enable the applications of tomorrow to become both simpler to maintain and more scalable than those built today.

9 Bibliography