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

A user-friendly approach to application-agnostic state synchronization

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Abstract

This study presents a comprehensive comparison and implementation of various methods for synchronizing memory regions in Linux systems over a network. Four approaches are evaluated: (1) handling page faults in userspace with `userfaultfd`, (2) utilizing `mmap` for change notifications, (3) hash-based change detection, and (4) custom filesystem implementation. Each option is thoroughly examined in terms of implementation, performance, and associated trade-offs. The study culminates in a summary that compares the options based on ease of implementation, CPU load, and network traffic, and offers recommendations for the optimal solution depending on the specific use case, such as data change frequency and kernel/OS compatibility.

Contents

1 Introduction

- Research question: Could memory be the universal way to access and migrate state?
- Why efficient memory synchronization is the missing key component for it to be that universal way
- High-level use cases for memory synchronization in the industry today (VM live migration, accessing remote ressources transparently)

2 Technology

2.1 User Space and Kernel Space

- Kernel
  - Core of the operating system
  - Direct interacts with hardware
  - Manages system ressources (CPU time, memory etc.)
  - Enforces security policies
  - Responsible for progress scheduling, memory management and drivers
  - Kernel space is the memory region in which the kernel is stored and executes in
- User space
  - Portion of system memory where user mode applications execute
  - Applications can't directly access hardware or kernel memory, will need to use an API to get access[1]
- Syscalls
  - Serve as the API between user space and kernel space
  - Makes it possible for applications in user space to access low-level OS services
  - Common examples are `open()`, `read()`, `write()` and `close()`
- `ioctl`
  - Is a fairly generic syscall
  - Allows for device-specific actions that can't be done with or expressed by regular system calls, providing a universal I/O model[2]
  - Can be an implementation hurdle for languages because they are defined by using numeric constants and arbitrary pointers, making type safety an issue [3]

2.2 The Linux Kernel

- Open-Source kernel created by Linus Torvals in 1991
- Written in C, with recently Rust being added as an additional allowed language[4]
- Powers millions of devices worldwide (servers, desktops, phones, embedded devices)
- Is a bridge between applications and hardware
  - Provides an abstraction layer
  - Compatible with many architectures (ARM, x86, RISC-V etc.)
- Is not an operating system in itself, but is usually made an operating system by a distribution[5]
- Distributions add userspace tools (e.g. GNU coreutils or BusyBox), desktop environments and more, turning into a full operating system
- Is a good fit for this thesis due to it's open-source nature (allows anyone to view, modify and contribute to the source code)

2.3 Linux Kernel Modules

- Linux kernel is monolithic, but extensible thanks to kernel modules[5]
- Small pieces of kernel-level code that can be loaded and unloaded as kernel modules
- Can extend the kernel functionality without reboots
- Are dynamically linked into the running kernel
- Helps keep kernel size manageable and maintainable
- Kernel modules are written in C
- Interact with kernel through APIs
- Poorly written modules can cause signficant kernel instability
- Modules can be loaded at boot time or dynamically (`modprobe`, `rmmod` etc.)[6]
- Module lifecycle can be implemented with initialization and cleanup functions

2.4 UNIX Signals and Handlers

- Signals
  - Are software interrups that notify a process of important events (exceptions etc.)
  - Can originate from the kernel, user input or different processes
  - Function as an asynchronous communication mechanism between processes or the kernel and a process
  - Have default actions, i.e. terminating the process or ignoring a signal[7]
- Handlers

  - Can be used to customize how a process should respond to a signal
  - Can be installed with `sigaction()`[8]
- Signals are not designed as an IPC mechanism, since they can only alert of an event, but not of any additional data for it

2.5 UNIX Sockets

- Overview
  - Allow processes on the same host system to communicate with each other[9]
  - Unlike UNIX signals, they can be used for IPC with addition data
  - Are also available in Linux
- Types
  - Stream sockets use TCP to provide reliable, two-way, connection-based byte streams, making them optimal for applications that need strong consistency
  - Datagram sockets use UDP, and thus allow for connection-less communication with less guarantees, prioritizing speed
- Named and unnamed
  - Named UNIX sockets are a special file type on the file system and can be identified as a path, and allow easy communication between unrelated processes
  - Unnamed UNIX sockets exist only in memory and disappear after the creating process terminates
- UNIX sockets allow file descriptor passing (see section on `userfaultfd`), which allows sharing a file descriptor between processes

2.6 Memory Hierarchy

- Memory in computers can be classified based on size, speed, cost and proximity to the CPU
- Principle of locality: The most frequently accessed data and instructions should be in the closest memory[10]
- Locality is important mostly due to the "speed of the cable" - throughput (due to dampening) and latency (due to being limited by the speed of light) decreases as distance increases
- Registers
  - Closest to the CPU
  - Very small amount of storage (32-64 bits of data)

  - Used by the CPU to perform operations
  - Very high speed, but limited in storage size

- Cache Memory
  - Divided into L1, L2 and L3
  - The higher the level, the larger and less expensive a layer
  - Buffer for frequently accessed data
  - Predictive algorithms optimize data usage
- Main Memory (RAM)
  - Offers larger capacity than cache but is slower
  - Stores programs and open files
- Secondary Storage (SSD/HDD)
  - Slower but RAM but can store larger amounts of memory
  - Typically stores the OS etc.
  - Is persistent (keeps data after power is cut)
- Tertiary Storage (optical disks, tape)
  - Slow, but very cheap
  - Tape: Can store very large amounts of data for relatively long amounts of time
  - Typically used for archives or physical data transport (e.g. import from personal infrastructure to AWS)
- Evolution of the hierarchy
  - Technological advancements continue to blur this clear hierarchy[11]
  - E.g. NVMe rivals RAM speeds but can store larger amounts of data
  - This thesis also blurs the hierarchy by exposing e.g. tertiary or secondary storage with the same interfae as main memory

2.7 Memory Management in Linux

- Memory management is a crucial part of every operation system - maybe even the whole point of an operating system
- Creates buffer between applications and physical memory
- Can provide security guarantees (e.g. only one process can access it’s memory)
- Kernel space
  - Runs the kernel, kernel extensions, device drivers

  - Managed by the kernel memory module
  - Uses slab allocation (groups objects of the same size into caches, speeds up memory allocation, reduces fragmentation of the memory)[12]

- User space
  - Applications (and some drivers) store their memory here[13]
  - Managed through a paging system
  - Each application has it's own private virtual address space
  - Virtual address space divided into pages of 4 KB
  - Pages can be mapped to any "real" location in physical memory

2.8 Swap Space

- A portion of the secondary storage is for virtual memory[13]
- Essential for systems running multiple applications
- Moves inactive parts of ram to secondary storage to free up space for other processes
- Implementation in Linux
  - Linux uses a demand paging system: Memory is only allocated when it is needed
  - Can be either a swap partition (separate area of the secondary storage) or file (regular file that can be expanded/trucnated)
  - Swap paritions and files are transparent to use
  - Kernel uses a LRU algoruithm for deciding which pages
- Role in hiberation
  - Before hibernating, the system saves the content of RAM into swap (where it is persistent)
  - When resuming, memory is read back from swap
- Role on performance
  - If swap is used too heavily, since the secondary storage is usually slower than primary memory, it can lead to signficant slowdowns
  - "Swapiness" can be set for the kernel, which controls how likely the system is to swap memory pages

2.9 Page Faults

- Page faults occur when the process tries to access a page not available in primary memory, which ause the OS to swap the required page from secondary storage into primary memory[6]

- Types
  - Minor page faults: Page is already in memory, but not linked to the process that needs it
  - Major page fault: Needs to be loaded from secondary storage
- The LRU (and simpler clock algorithm) can minimize page faults
- Techniques for handling page faults
  - Prefetching: Anticipating future page requests and loading them into memory in advance
  - Page compression: Compressing inactive pages and storing them in memory pre-emptively (so that less major faults happen)[14]
- Usually, handling page faults is something that the kernel does

2.10 `mmap`

- Overview
  - UNIX system call for mapping files or devices into memory
  - Multiple possible usecases: Shared memory, file I/O, fine-grained memory allocation
  - Commonly used in applications like databases
  - Is a “power tool”, needs to be used carefully and intentionally
- Functionality
  - Establishes direct link (memory mapping) between a file and a memory region[15]
  - When the system reads from the mapped memory region, it reads from the file directly and vice versa
  - Reduces overhead since no or less context switches are needed
- Benefits:
  - Enables zero-copy operations: Data can be accessed directly as though it were in memory, without having to copy it from disk first
  - Can be used to share memory between processes without having to go through the kernel with syscalls[7]
- Drawback: Bypasses the file system cache, which can lead to stale data if multiple processes read/write at the same time

2.11 `inotify`

- Event-driven notification system of the Linux kernel[16]

- Monitors file system for events (i.e. modifications, access etc.)
- Uses a watch feature for monitoring specific events, e.g. only watching writes
- Reduces overhead and ressource use compared to polling
- Widely used in many applications, e.g. Dropbox for file synchronization
- Has limitations like the limit on how many watches can be established

2.12 Linux Kernel Disk and File Caching

- Disk caching
  - Temporarely stores frequently accessed data in RAM
  - Uses principle of locality (see Memory Hierarchy)
  - Implemented using the page cache subsystem in Linux
  - Uses the LRU algorithm to manage cache contents
- File caching
  - Linux caches file system metadata in the `dentry` and `inode` caches
  - Metadata includes i.e. file names, attributes and locations
  - This caching accelerates the resoluton of path names and file attributes (i.e. the last change data for polling)
  - File reads/writes pass through the disk cache
- Complexities
  - Data consistency: Between the disk and cache via writebacks. Aggressive writebacks lead to reduced performance, delays risk data loss
  - Release of cached data under memory pressure: Cache eviction requires intelligent algorithms, i.e. LRU[6]

2.13 LAN and WAN

- RTT
  - Round trip time (RTT) represents the time data takes to travel from a source to a destination and back
  - Provides an insight into application latency
  - Varies according to many factors such as network type, system load and physical distance
- LAN
  - Local area networks (LAN) is network infrastructure in geographically small areas[17]

  - Often the network of a single data center or office
  - Typically have a low RTT and thus latency due to a short number of relays and the small distances that need to be travelled
  - Perimeter security is often applied, where a LAN is seen as a trusted network that doesn't require authentication between internal systems

- WAN
  - Wide area networks (WAN) span large geographical areas
  - Most famous example is the open internet
  - Due to physical distance and number of hops, often higher RTT and latency
  - Typically open to wiretapping and packet inspection, requiring encryption and authentication for services to operate

2.14 TCP, UDP, TLS and QUIC

- TCP
  - Connection-oriented
  - Has been the reliable backbone of internet commmunication
  - Guaranteed delivery and maintained data order
  - Includes error checking, lost packet retransmission, and congestion control mechanisms
  - Powers the majority of the web[18]
- UDP
  - Connectionless
  - No reliability or ordered packet delivery guarantees
  - Faster than TCP due to less guarantees
  - Suitable for applications that require speed over reliability (i.e. online gaming, video calls etc.)[19]
- TLS
  - Encryption protocol that intents to secure communication over a public network like the internet, has been updated many times to evolve with time
  - Uses both symmetric and asymmetric encryption[20]
  - Is used for most internet communication, esp. in combination with HTTP in the form of HTTPS
  - Consists of a handshake phase, in which the parameters necessary to establish a secure connection are exchanged, as well as session keys and certificates, before continuing on to the data transfer phase using the encrypted connection

  - Can also be used for authentication through the use of mutual TLS (mTLS), where the client is authenticated by the server

- QUIC
  - Modern transport layer protocol developed by Google and standardized by the IETF in 2020
  - Intents to combine the best aspects of TCP and UDP
  - Provides reliability and ordered delivery guarantees
  - Reduces connection establishment times/initial latency by combining connection and security handshakes
  - Avoids head-of-line blocking by allowing independent delivery of separate streams[21]

2.15 Delta Synchronization

- Traditionally, when files are synchronized between hosts, the entire file is transfered
- Delta synchronization is a technique that intents to instead transfer only the part of the file that has changed
- Can lead to reduced network and I/O overhead
- The probably most popular tool for file synchronization like this is rsync
- When a delta-transfer algorithm is used, it computes the difference between the local and the remote file, and then synchronizes the changes
- The delta sync algorithm first does file block division
- The file on the destination is divided into fixed-size blocks
- For each block in the destination, a weak and fast checksum is calculated
- The checksums are sent over to the source
- On the source, the same checksum calculation process is run, and compared against the checksums that were sent over (matching block identification)
- Once the changed blocks are known, the source sends over the offset of each block and the changed block's data to the destination
- When a block is received, the destination writes the chunk to the specified offset, reconstructing the file
- Once one polling interval is done, the process begins again[22]

2.16 File Systems In Userspace (FUSE)

- Software interface that allows writing custom file systems in userspace
- Developers can create file systems without having to engage in low-level kernel development

- Available on multiple platforms, mostly Linux but also macOS and FreeBSD
- In order to implement file systems in user space, we can use the FUSE API
- Here, a user space program registers itself with the FUSE kernel module
- This program provides callbacks for the file system operations, e.g. for `open`, `read`, `write` etc.
- When the user performs a file system operation on a mounted FUSE file system, the kernel module will send a request for the operation to the user space program, which can then reply with a response, which the FUSE kernel module will then return to the user
- This makes it much easier to create a file system compared to writing it in the kernel, as it can run in user space
- It is also much safer as no custom kernel module is required and an error in the FUSE or the backend can't crash the entire kernel
- Unlike a file system implemented as a kernel module, this layer of indirection makes the file system portable, since it only needs to communicate with the FUSE module
- Does have significant performance overhead due to the context switching between the kernel and the file system in userspace[23]
- Is used for many high-level interfaces to extenal services, i.e. to mount S3 buckets or a remote system's disk via SSH

2.17 Network Block Device (NBD)

- NBD uses a protocol to communicate between a server (provided by user space) and a client (provided by the NBD kernel module)
- The protocol can run over WAN, but is really mostly designed for LAN or localhost usage
- It has two phases: Handshake and transmission[24]
- There are multiple actors in the protocol: One or multiple clients, the server and the virtual concept of an export
- When the client connects to the server, the server sends a greeting message with the server's flags
- The client responds with its own flags and an export name (a single NBD server can expose multiple devices) to use
- The server sends the export's size and other metadata, after which the client acknowledges the received data and the handshake is complete
- After the handshake, the client and server start exchanging commands and replies
- A command can be any of the basic operations needed to access a block device, e.g. read, write or flush
- Depending on the command, it can also contain data (such as the chunk to be written), offsets, lengths and more
- Replies can contain an error, success value or data depending on the reply's type

- NBD is however limited in some respects; the maximum message size is 32 MB, but the maximum block/chunk size supported by the kernel is just 4096 KB, making it a suboptimal protocol to run over WAN, esp. in high latency scenarios
- The protocol also allows for listing exports, making it possible to e.g. list multiple memory regions on a single server
- NBD is an older protocol with multiple different handshake versions and legacy features
- Since the purpose of NBD in this use case is minimal and both the server and the client are typically controlled, it makes sense to only implement the latest recommended versions and the baseline feature set
- The baseline feature set requires no TLS, the latest "fixed newstyle" handshake, the ability to list and choose an export, as well as the read, write and disc(onnect) commands and replies
- As such, the protocol is very simple to implement
- With this simplicity however also come some drawbacks: NBD is less suited for use cases where the backing device behaves very differently from a random-access store device, like for example a tape drive, since it is not possible to work with high-level abstractions such as files or directories
- This is, for the narrow memory synchronization use case, however more of a feature than a bug
- Since it works on the block level, it can't offer shared acesses to the same file for multiple clients

2.18 Virtual Machine Live Migration

2.18.1 Pre-Copy

- While these systems already allow for some optimizations over simply using the NBD protocol over WAN, they still mean that chunks will only be fetched as they are being needed, which means that there still is a guaranteed minimum downtime
- In order to improve on this, a more advanced API (the managed mount API) was created
- A field that tries to optimize for this use case is live migration of VMs
- Live migration refers to moving a virtual machine, its state and connected devices from one host to another with as little downtime as possible
- There are two types of such migration algorithms; pre-copy and post-copy migration
- Pre-copy migration works by copying data from the source to the destination as the VM continues to run (or in the case of a generic migration, app/other state continues being written to)
- First, the initial state of the VM's memory is copied to the destination
- If, during the push, chunks are being modified, they are being marked as dirty
- These dirty chunks are being copied over to the destination until the number of remaining chunks is small enough to satisfy a maximum downtime criteria

- Once this is the case, the VM is suspended on the source, and the remaining chunks are synced over to destination
- Once the transfer is complete, the VM is resumed on the destination
- This process is helpful since the VM is always available in full on either the source or the destination, and it is resilient to a network outage occurring during the synchronization
- If the VM (or app etc.) is however changing too many chunks on the source during the migration, the maximum acceptable downtime criteria might never get reached, and the maximum acceptable downtime is also somewhat limited by the available RTT[25]

2.18.2 Post-Copy

- An alternative to pre-copy migration is post-copy migration
- In this approach, the VM is immediately suspended on the source, moved to the destination with only a minimal set of chunks
- After the VM has been moved to the destination, it is resumed
- If the VM tries to access a chunk on the destination, a page fault is raised, and the missing page is fetched from the source, and the VM continues to execute
- The benefit of post-copy migration is that it does not require re-transmitting dirty chunks to the destination before the maximum tolerable downtime is reached
- The big drawback of post-copy migration is that it can result in longer migration times, because the chunks need to be fetched from the network on-demand, which is very latency/RTT-sensitive[25]

2.18.3 Workload Analysis

- “Reducing Virtual Machine Live Migration Overhead via Workload Analysis” provides an interesting analysis of options on how this decision of when to migrate can be made[26]
- While being designed mostly for use with virtual machines, it could serve as a basis for other applications or migration scenarios, too
- The proposed method identifies workload cycles of VMs and uses this information to postpone the migration if doing so is beneficial
- This works by analyzing cyclic patters that can unnecessarily delay a VM’s migration, and identifies optimal cycles to migrate VMs in from this information
- For the VM use case, such cycles could for example be the GC of a large application triggering a lot of changes to the VMs memory etc.
- If a migration is proposed, the system checks for whether it is currently in a beneficial cycle to migrate in which case it lets the migration proceed; otherwise, it postpones it until the next cycle

- The algorithm uses a Bayesian classifier to identify a favorable or unfavorable cycle
- Compared to the alternative, which is usually waiting for a significant percentage of the chunks that were not changed before tracking started to be synced first, this can potentially yield a lot of improvements
- The paper has found an improvement of up to 74% in terms of live migration time/downtime and 43% in terms of the amount of data transferred over the network
- While such a system was not implemented for r3map, using r3map with such a system would certainly be possible

2.19 Streams and Pipelines

- Fundamental concepts in computer science
- Sequentually process elements
- Allow for the efficient processing of large amounts of data, without having to load everything into memory
- Form the backbone of efficient, modular data processing
- Streams
  - Represent a continous sequence of data
  - Can be a source or destination of data (i.e. files, network connections, stdin/stdout etc.)
  - Allow processing of data as it becomes available
  - Minimized memory consumption
  - Especially well suited for long-running processes (where data gets streamed in for a extended time)[27]
- Pipelines
  - Series of data processing stages: Output of one stage serves as input to the next[28]
  - Stages can often be run in parallel, improving performance due to a higher degree of concurrency
  - Example: Instruction pipeline in CPU, were the stages of instruction execution can be performend in parallel
  - Example: UNIX pipes, where the output of a command (e.g. `curl`) can be piped into another command (e.g. `jq`) to achieve a larger goal

2.20 Go

- Overview
  - Is a statically typed, compiled, open-source language by Google launched in 2009[29]

  - Known for it's simplicity
  - Developed to address the unsuitability of traditional languages for modern, distributed systems development
  - Built with input from many original UNIX developers, like Rob Pike and Ken Thompson
  - Popular particularly in cloud services and network programming

- Goroutines
  - Headline feature of Go
  - Enables concurrent function execution
  - Similar to threads conceptually, but much more scalable to millions of Goroutines per program
  - Synchronization and data transfer between Goroutines is done by channels, typed, concurrency-safe conduits for data

2.21 gRPC

- Open-Source and high-performance RPC framework
- Developed by Google in 2015
- Features
  - Uses HTTP/2 as the transport protocol to benefit from header compression and request multiplexing
  - Uses protobuf as the IDL and wire format, a high-performance, polyglot mechanism for data serialization (instead of the slower and more verbose JSON of REST APIs)
  - Supports unary RPCs, server-streaming RPCs, client-streaming RPCs and bidirectional RPCs
  - Has pluggable support for load balancing, tracing, health checking and authentication[30]j
- Supports many languages (Go, Rust, JS etc.)
- Developed by the CNCF

2.22 fRPC and Polyglot

- fRPC
  - Is an Open-Source RPC framework released by Loophole Labs in 2022
  - proto3-compatible, meaning that it can be used as a drop-in replacement for gRPC that promises better performance characteristics

  - Has the ability to stop the RPC system and retrieve an underlying connection, making it possible to re-use connections for different purposes [31]
  - Uses frisbee, a messaging framework to implement the request-response semantics [32]
- Polyglot
  - Is the high-performance serialization framework used by fRPC
  - Works similarly to protocol buffers, but is much simpler and carries less legacy code
  - Is also language independent, with current implementations for Go, Rust and TypeScript[33]

2.23 Redis

- In-memory data structure store
- Used as a database, cache and/or message broker
- Created by S. Sanfilippo in 2009
- Different from other NoSQL databases by supporting various data structures like lists, sets, hashes or bitmaps
- Uses in-memory data storage for maximum speed and efficiency[34]
- Allows for low-latency reads/writes
- While not intended for persistance, it is possible to store data on disk
- Has a non-blocking I/O model and offers near real-time data processing capabilities
- Includes a pub-sub system to be able to function as a message broker[35]

2.24 S3 and Minio

- S3
  - Object storage service for data-intensive workloads
  - Offered by AWS
  - Can be globally distributed to allow for fast access times from anywhere on the globe
  - Range of storage classes with different requirements
  - Includes authentication and authorization
  - Exposes HTTP API for accessing the stored folders and files[36]
- Minio
  - Open-source storage server compatible wiht S3
  - Lightweight and simple, written in Go
  - Can be hosted on-prem and is open source
  - Allows horizontal scalability to storage large amounts of data across nodes[37]

2.25 Cassandra and ScylllaDB

- Popular wide-column NoSQL databases
- Combines Amazon's Dynamo model and Google's Bigtable model to create a highly available database
- Apache Cassandra
  - Highly scalable, eventually consistent
  - Can handle large amounts of data across many servers with no single point of failure
  - Consistency can be tuned according to needs (eventual to strong)
  - Doesn't use master nodes due to it's use of a P2P protocol and distributed hash ring design
  - Does have high latency under heavy load and requires fairly complex configuration[38]
- ScyllaDB
  - Launched in 2015
  - Written in C++ and has a shared-nothing architecture, unlike Cassandra which is written in Java
  - Compatible with Cassandra's API and data model[39]
  - Designed to overcome Cassandra's limitations esp. around latency, esp. P99 latency
  - Performance improvements were confirmed with various benchmarking studies[40]

3 Planning

3.1 Pull-Based Synchronization With `userfaultfd`

- An implementation of post-copy migration
- Memory region is created on the destination host
- Reads to the region by the migrated ressource trigger a page fault
- When we encounter such a page fault, we want to fetch the relevant offset from the remote
- Typically, page faults are resolved by the Kernel
- Here, we want to handle handle the page faults in user space
- Traditionally, it was possible to use `SIGSEGV` signal handlers to use handle this from the program
- This however is complicated and slow
- Instead we use a new kernel API ("Userfaults")
- We register the region to be handled by userfault
- Then we start an handler that fetches the offsets from the remote on a page fault
- This handler is connected to the registered region using a file descriptor

- We can transfer the handler's file descriptor between processes over a socket

3.2 Push-Based Synchronization With mmap and Hashing

- As mentioned before `mmap` allows mapping a memory region to a file
- We can put the migratable ressource into this memory region
- If we get writes to the region, we eventually get writes to the region
- If we're able to detect these writes and copy them to the destination, we can implement a pre-copy migration system
- `mmap`ed regions still using caching to speed up reads
- Changes from the region can be flushed to the disk with `msync`
- Usually we could use inotify to detect changes to the file, but inotify doesn't work with `mmap`ed regions
- As a result, we went for a polling-based solution instead
- Polling has drawbacks, which we tried to work around upon in our implementation

3.3 Push-Pull Synchronization with FUSE

- Can serve as the basis for a pre- or post-copy migration
- Similarly to the file-based synchronization we `mmap` a file into memory
- The file is stored on a custom file system
- We then catch reads (for a post-copy migration) or writes (for a pre-copy migration)
- Implementing a file system in the kernel is possible but cumbersome as described before
- With FUSE we can implement the file system in user space, making this much simpler, as we'll show in the implementation

3.4 Mounts with NBD

- Another `mmap`-based solution for pre- and/or post-copy migration
- Instead of `mmap`ing a file, a device is `mmap`ed
- This device is a block device provided by a NBD client
- We can then connect the NBD client to a remote NBD server, which contains the migratable resource
- Any reads to/from the region go to/from the block device and are resolved by the remote NBD server
- Isn't per se a synchronization methods on it's own, but rather a remote mount
- Unlike a FUSE this means we only need to implement a block device, not a full file system

- Implementation and performance overhead of a NBD server is fairly low as we'll show in the implementation however

3.5 Push-Pull Synchronization with Mounts

- Also tracks changes to the memory region of the migratable ressource using NBD
- Limitations of the NBD protocol in WAN
  - Usually, the NBD server and client don't run on the same system
  - NBD was originally designed to used as a LAN protocol to access a remote hard disk
  - As mentioned before, NBD can run over WAN, but is not designed for this
  - The biggest problem with running NBD over a public network, even if TLS is enabled is latency
  - Individual chunks would only be fetched to the local system as they are being accessed, adding a guaranteed minimum latency of at least the RTT
  - Instead of directly connecting a client to a remote server, we add a layer of indirection, called a `Mount` that consists of both a client and a server, both running on the local system
- Combining the NBD server and client to a reusable unit
  - We then connect the server to the backend with an API that is better suited for WAN usage
  - This also makes it possible to implement smart pull/push strategies instead of simply directly writing to/from the network ("managed mounts")
- The mount WAN protocol
  - The simplest form of the mount API is the direct mount API
  - This API simply swaps out NBD for a transport-independent RPC framework, but does not do additional optimizations
  - It has two simple actors: A client and a server, with only the server providing methods to be called (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/backend.go#L14-L19)
  - The protocol as such is stateless, as there is only a simple remote read/write interface (add state machine and sequence diagram here)
- Chunking
  - One additional layer that needs to be implemented however is proper chunking support
  - While we can specify a chunk size for the NBD client in the form of a block size, we can only go up to 4 KB chunks
  - For scenarios where the RTT between the backend and server is large, it might make sense to use a much larger chunk size for the actual networked transfers

  - Many backends also have constraints that prevent them from functioning without a specific chunk size or aligned offsets, such as using e.g. tape drives, which require setting a block size and work best when these chunks are multiple MBs instead of KBs
  - Even if there are no constraints on chunking on the backend side (e.g. when a file is used as the backend), it might make sense to limit the maximum supported message size between the client and server to prevent DoS attacks by forcing the backend to allocate large chunks of memory to satisfy requests, which requires a chunking system to work

- Server-side vs. client-side chunking

  - It is possible to do the chunking in two places; on the mount API's side, and on the (potentially remote) backend's side
  - Doing the chunking on the backend's side is usually much faster than on the mount API's side, as writes with lengths smaller than the chunk size will mean that the remote chunk needs to be fetched first, significantly increasing the latency esp. in scenarios with high RTT

- Combining pre- and post-copy migration with managed mounts

  - For the managed mount API, both paradigms were implemented
  - The managed mount API is primarily intended as an API for reading from a remote resource and then syncing changes back to it however, not migrating a resource between two hosts
  - The migration API (will be discussed later) is a more well-optimized version for this use case

- Background pull

  - The pre-copy API is implemented in the form of preemptive pulls based on another `ReaderAt`
  - It allows passing in a pull heuristic function, which it uses to determine which chunks should be pulled in which order
  - Many applications commonly access certain bits of data first
  - If a resource should be available locally as quickly as possible, then using the correct pull heuristic can help a lot
  - For example, if the data pulled consists of a header, then using a pull heuristic that pulls these header chunks first can be of help
  - If a file system is being synchronized, and the superblocks of the file system are being stored in a known pattern, the pull heuristic can be used to fetch these superblocks first
  - If a format like MP4, which has an index, is used then said index can be fetched first, be accessed first and during the time needed to parse the index, the remaining data can be pulled in the background

- Background push
  - In order to also be able to write back however, it needs to have a push system as well
  - This push system is being started in parallel with the pull system
  - It also takes a local and a remote `ReadWriterAt`
  - This integrates with the callbacks supplied by the syncer, which ensures that we don't sync back changes that have been pulled but not modified, only the ones that have been changed locally

3.6 Pull-Based Synchronization with Migrations

- Also tracks changes to the memory region of the migratable ressource using NBD
- Optimization mounts for migration scenarios
  - We have now implemented a managed mounts API
  - This API allows for efficient access to a remote resource through memory
  - It is however not well suited for a migration scenario
  - For migrations, more optimization is needed to minimize the maximum acceptable downtime
  - For the migration, the process is split into two distinct phases
  - The same preemptive background pulls and parallelized device/syncer startup can be used, but the push process is dropped
  - The two phases allow pulling the majority of the data first, and only finalize the move later with the remaining data
  - This is inspired by the pre-copy approach to VM live migration, but also allows for some of the benefits of the post-copy approach as we'll see later
  - Why is this useful? A constraint for the mount-based API that we haven't mentioned before is that it doesn't allow safe concurrent access of a resource by two readers or writers at the same time
  - This poses a problem for migration, where the downtime is what should be optimized for, as the VM or app that is writing to the source device would need to be suspended before the transfer could begin
  - This adds very significant latency, which is a problem
  - The mount API was also designed in such a way as to make it hard to share a resource this way
  - The remote backend for example API doesn't itself provide a mount to access the underlying data, which further complicates migration by not implementing a migration lifecycle
- The migration protocol

  - To fix this, the migration API defines two new actors: The seeder and the leecher
  - The seeder represents a resource that can be migrated from/a host that exposes a migratable resource
  - The leecher represents a client that wants to migrate a resource to itself
  - Initially, the protocol starts by running an application with the application's state on the seeder's mount
  - When a leecher connects to the seeder, the seeder starts tracking any writes to it's mount
  - The leecher starts pulling chunks from the seeder to it's local backend
  - Once the leecher has received a satisfactory level of locally available chunks, it as the seeder to finalize, which then causes the seeder to stop the remote app, `msync`/flushes the drive, and returns the chunks that were changed between it started tracking and finalizing
  - The leecher then marks these chunks as remote, immediately resumes the VM, and queues them to be pulled immediately
  - By splitting the migration into these two distinct phases, the overhead of having to start the device on the leecher can be skipped and additional app initialization that doesn't depend on the app's state (e.g. memory allocation, connecting to databases, loading models etc.) can be performed before the application needs to be suspended
  - This solution combines both the pre-copy algorithm (by pulling the chunks from the seeder ahead of time) and the post-copy algorithm (by resolving dirty chunks from the seeder after the VM has been migrated) into one coherent protocol (add state machine diagram here)
  - This way, the maximum tolerable downtime can be drastically reduced, and dirty chunks don't need to be re-transmitted multiple times
  - Effectively, it drops the maximum guaranteed 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 starting to track and finalize

- The finalization phase

  - A interesting question to ask with the two-step migration API is when to start the finalization step
  - As is visible from the migration API protocol state machine showed beforehand, the finalization stage is critical and hard or impossible to recover from depending on the implementation
  - While for the memory sync on its own, one could just call `Finalize` multiple times to restart it
  - But since `Finalize` needs to return a list of dirty chunks, it requires the VM or app on the source device to be suspended before `Finalize` can return

  - While not necessarily the case, such a suspend operation is not idempotent (since it might not just be a suspension that is required, but also a shutdown of dependencies etc.)

4 Implementation

4.1 Userfaults in Go with `userfaultfd`

- General functionality
  - By listening to page faults, we know when a process wants to access a specific piece of memory
  - We can use this to then pull the chunk of memory from a remote, map it to the address on which the page fault occured, thus only fetching data when it is required
  - Usually, handling page faults is something that the kernel does
  - In our case, we want to handle page faults in userspace, and implement post-copy with them
  - In the past, this used to be possible from userspace by handling the `SIGSEGV` signal in the process
  - In our case however, we can use a recent system called `userfaultfd` to do this in a more elegant way (available since kernel 4.11)
  - `userfaultfd` allows handling these page faults in userspace
  - The region that should be handled can be allocated with e.g. `mmap`
  - Once we have the file descriptor for the `userfaultfd` API, we need to transfer this file descriptor to a process that should respond with the chunks of memory to be put into the faulting address
  - Once we have received the socket we need to register the handler for the API to use
  - If the handler receives an address that has faulted, it responds with the `UFFDIO_COPY` `ioctl` and a pointer to the chunk of memory that should be used on the file descriptor (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/handler.go
- API design for `userfault-go`
  - Implementing this in Go was quite tricky, and it involves using `unsafe`
  - We can use the `syscall` and `unix` packages to interact with `ioctl` etc.
  - We can use the `ioctl` syscall to get a file descriptor to the `userfaultfd` API, and then register the API to handle any faults on the region (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/register.go#L15)
  - Passing file descriptors between processes is possible by using a UNIX socket (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/transfer/unix.go)

- Implementing `userfaultfd` backends
  - A big benefit of using `userfaultfd` and the pull method is that we are able to simplify the backend of the entire system down to a `io.ReaderAt` (code snippet from https://pkg.go.dev/io#ReaderAt)
  - That means we can use almost any `io.ReaderAt` as a backend for a `userfaultfd-go` registered object
  - We know that access will always be aligned to 4 KB chunks/the system page size, so we can assume a chunk size on the server based on that
  - For the first example, we can return a random pattern in the backend (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-abc/main.go) - this shows a great way of exposing truly arbitrary information into a byte slice without having to pre-compute everything or changing the application
  - Since a file is a valid `io.ReaderAt`, we can also use a file as the backend directly, creating a system that essentially allows for mounting a (remote) file into memory (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-file/main.go)
  - Similarly so, we can use it map a remote object from S3 into memory, and access only the chunks of it that we actually require (which in the case of S3 is achieved with HTTP range requests) (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-s3/main.go)

4.2 File-Based Synchronization

- File-based synchronization
  - We can do this by using `mmap`, which allows us to map a file into memory
  - By default, `mmap` doesn't write changes from a file back into memory, no matter if the file descriptor passed to it would allow it to or not
  - We can however add the `MAP_SHARED` flag; this tells the kernel to write back changes to the memory region to the corresponding regions of the backing file
  - Linux caches reads to such a backing file, so only the first page fault would be answered by fetching from disk, just like with `userfaultfd`
  - The same applies to writes; similar to how files need to be `sync`ed in order for them to be written to disks, `mmap`ed regions need to be `msync`ed in order to flush changes to the backing file
  - In order to synchronize changes to the region between hosts by syncing the underlying file, we need to have the changes actually be represented in the file, which is why `msync` is critical

  - For files, you can use `O_DIRECT` to skip this kernel caching if your process already does caching on its own, but this flag is ignored by the `mmap`

- `inotify` vs. polling

  - Usually, one would use `inotify` to watch changes to a file
  - `inotify` allows applications to register handlers on a file's events, e.g. `WRITE` or `SYNC`. This allows for efficient file synchronization, and is used by many file synchronization tools
  - It is also possible to filter only the events that we need to sync the writes, making it the perfect choice for this use case
  - For technical reasons however (mostly because the file is represented by a memory region), Linux doesn't fire these events for `mmap`ed files though, so we can't use it
  - The next best option are two: Either polling for file attribute changes (e.g. last write), or by continously hashing the file to check if it has changed
  - Polling on its own has a lot of downsides, like it adding a guaranteed minimum latency by virtue of having to wait for the next polling cycle
  - This negatively impacts a maximum allowed downtime scenario, where the overhead of polling can make or break a system
  - Hashing the entire file is also a naturally IO- and CPU-intensive process because the entire file needs to be read at some point
  - Still, polling & hashing is probably the only reliable way of detecting changes to a file
  - Instead of hashing the entire file, then syncing the entire file, we can want to really sync only the parts of the file that have changed between two polling iterations

- Detecting file changes

  - We can do this by opening up the file multiple times, then hashing individual offsets, and aggregating the chunks that have changed
  - When picking algorithms for this hashing process, the most important metric to consider is the throughput with which it can compute hashes, as well as the change of collisions
  - If the underlying hashing algorithm is CPU-bound, this also allows for better concurrent processing
  - Increases the initial latency/overhead by having to open up multiple file descriptors
  - But this can not only increase the speed of each individual polling tick, it can also drastically decrease the amount of data that needs to be transferred since only the delta needs to be synchronized
  - Hashing and/or syncing individual chunks that have changed is a common practice

- Delta synchronization protocol

  - We have implemented a simple protocol for this delta synchronization, just like rsync's

    delta synchronization algorithm (code snippet from https://github.com/loopholelabs/darkmagyk/blob/m orchestrator/main.go#L1337-L1411 etc.)
  - For this protocol specifically, we send the changed file's name as the first message when starting the synchronization, but a simple multiplexing system could easily be implemented by sending a file ID with each message
  - Its intended to be simpler than `rsync`, and to support a central forwarding up instead of requiring P2P connectivity between each host
  - Defines three actors: Multiplexer, file advertiser, and file receiver
  - TODO: Add sequence diagram for the protocol

- Multiplexer hub
  - Accepts TLS connections
  - Reads client certificate
  - Reads the common name from the certificate, contains the `syncerID`
  - Spawns a new goroutine for the syncerID to handle the communication with the specific syncer
  - In goroutine it reads the type of the peer
  - For the `src-control` peer type (code snippet from https://github.com/loopholelabs/darkmagyk/blob/1 cloudpoint/main.go#L824-L844)
    - Reads a file name from the syncer
    - Registers the connection as the one providing the file with this name
    - Broadcasts the file as one now being available
  - For the `dst-control` peer type (code snippet from https://github.com/loopholelabs/darkmagyk/blob/1 cloudpoint/main.go#L845-L880)
    - It listens to the broadcasted files from `src-control` peers
    - Relays any received file names to the `dst-control` peers so that it can subscribe
    - Also sends all currently known file names to the peer
  - For the `dst` peer type (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4af/cmc cloudpoint/main.go#L882-L942)
    - Decodes a file name from the connection
    - Looks for a corresponding `src-control` peer
    - If it has found a peer, it creates and sends a new ID for this connection to the `src-control` peer
    - Waits until the `src-control` peer has connected to the hub with this ID with a new `src-data` peer by listening for `src-data` peer ID broadcasts
    - Spawns two new goroutines that copy to/from this newly created synchronization connection and the connection of the `dst` peer, relaying any information between the two

  - For the `src-data` peer type (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4 cloudpoint/main.go#L943-L954)
    - Decodes the ID for the peer
    - Broadcasts the ID, which allows the `dst` peer to continue

- File advertisement (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4afe0828452c61 L269)
  - Connects to the multiplexer hub
  - Registers itself as a `src-control` type of peer
  - Sends the file name to the multipler hub
  - Starts a loop to handle `dst` peer types
  - As mentioned earlier, `dst` peer types send a ID
  - Waits until ID is received, then spawns new goroutine
  - Connects to the multiplexer hub again, this time registering itself as a `src-data` peer
  - Opens the file that it advertises
  - Sends the ID that it has received back to the multiplexer hub, which connects it to the matching `dst` peer
  - Starts a new loop that starts the file transmission to the `dst` peer through the multiplexer
  - Checks if context is cancelled to stop the transmission by breaking the loop if requested
  - After a file transmission is done, waits for the specified polling interval, then starts a file transmission cycle again

- File receiver loop (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4afe0828452c61c L243)
  - Connects to the multiplexer hub
  - Registers itself as the `dst-control` type of peer
  - Spawns a new goroutine after a file name has been received from the multiplexer hub
  - Connects to the multiplexer hub again, this time registering itself as a `dst` peer
  - Opens up the destination file
  - Registers itself as the `dst` peer
  - Sends the file name to the multiplexer, causing the multiplexer to look for a peer that advertises the requested file
  - Starts the file receiption process in a loop, and exits if a callback determines that the file has been synced completely

- File transmission (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4afe0828452c61c L110)
  - Does the actual file transmission part of delta synchronization

  - Receives the remote hashes from the mulitplexer hub
  - Calculates the local hashes
  - Compares local and remote hashes
  - Sends hashes that don't match to the remote via the multiplexer hub
  - If the remote sent less hashes than there were locally, asks the remote to truncate it's file to the size of the local file that is being synchronized
  - Loops over each of the chunks that need to be sent, and sends the updated data for the file in order

- Hash calculation (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4afe0828452c61c L73)

  - `GetHashesForBlocks` handles the concurrent calculation of the hashes for a file on both the file transmitter and receiver
  - It uses a semaphore to limit the amount of concurrent access to the file that is being hashed
  - Acquires a lock on the semaphore for each hash calculation worker
  - Each worker goroutine then opens up the file and calculates a CRC32 hash
  - To allow for easy transmission of the hashes over the network, it encodes them as hash values
  - After all hashes have been calculated, it adds them to the array and returns them

- File receiption (code snippet from https://github.com/loopholelabs/darkmagyk/blob/159d4afe0828452c61c63 L125)

  - Is the receiving part of the delta synchronization algorithm
  - Starts by calculating hashes for the local copy of the file
  - Sends the local hashes to the emote, then waits until it has received the remote's hashes
  - If the remote detected that the file needs to be truncated (by sending a negative cutoff value), it truncates the file
  - If the remote's file has grown beyond what the local file is since the last synchronization cycle, it extends the file
  - It then reads from the remote in order, and copies each chunk to the matching offset in the local file

4.3 FUSE Implementation in Go

- Implementing a FUSE in Go means tight integration with libraries
- It makes sense to divide the process into two aspects
- Creating a backend for a file system abstraction API like `afero.Fs`

  - By using a file system abstraction API like `afero.Fs`, we can separate the FUSE implementation from the actual file system structure, making it unit testable and making it possible to add caching in user space (code snippet from https://github.com/pojntfx/stfs/blob/main/pkg/fs/fil
  - It is possible to use even very complex and at first view non-compatible backends as a FUSE file system's backend
  - For example, STFS used a tape drive as the backend, which is not random access, but instead append-only and linear (https://github.com/pojntfx/stfs/blob/main/pkg/operations/update.go)
  - By using an on-disk index and access optimizations, the resulting file system was still performant enough to be used, and supported almost all features required for the average user
  - Creating an adapter between the FS abstraction API and the FUSE library's backend interface
  - It is possible to map any `afero.Fs` to a FUSE backend, so it would be possible to switch between different file system backends without having to write FUSE-specific (code snippet from https://github.com/JakWai01/sile-fystem/blob/main/pkg/filesystem/fs.go)

- While this approach is interesting, the required overhead of implementing it (which is known from prior projects like STFS[41] and sile-fystem[42]) and other factors that will be touched upon later led to it not being pursued further

4.4 NBD with go-nbd

- Overview
  - Due to the lack of pre-existing libraries, a new pure Go NBD library was implemented
  - This library does not rely on CGo/a pre-existing C library, meaning that a lot of context switching can be skipped
- Server
  - The server is completely in user space, there are no kernel components involved here
  - The backend interface for `go-nbd` is very simple and only requires four methods: `ReadAt`, `WriteAt`, `Size` and `Sync`
  - The key difference here to the way backends were designed in `userfaultfd-go` is that they can also handle writes
  - A good example backend that maps well to a block device is the file backend (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/backend/file.go)
  - `go-nbd` exposes a `Handle` function to support multiple users without depending on a specific transport layer (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go)

- This means that systems that are peer-to-peer (e.g. WebRTC), and thus don't provide a TCP-style `accept` syscall can still be used easily
- It also allows for easily hosting NBD and other services on the same TCP socket
- Role of exports (https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L23-L28)
- Role of options (https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L30-L35)
- To make it easier to parse, the headers and other structured messages are modeled as Go structs where possible (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/protocol/negotiation.go)
- Only the "fixed newstyle" handshake is implemented to keep it simple[24]
- Server initiates the handshake by sending the handshake header and ignoring client flags (https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L56-L68)
- The handshake is implemented using a simple for loop, which either returns on error or breaks
- The server encodes/decodes messages with the `binary` package (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L73-L76)
- Client sends options with ID
- `NEGOTIATION_ID_OPTION_INFO` and `NEGOTIATION_ID_OPTION_GO` exchange the information about the chosen export (i.e. block size, export size, export name and description etc.), and if `GO` is specified it continues directly to the transmission phase
- If an export isn't found, the server aborts the connection (https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L102-L120)
- `NEGOTIATION_ID_OPTION_LIST` encodes and sends the list of exports to the client
- `NEGOTIATION_ID_OPTION_ABORT` aborts the handshake, causing the server to close the connection
- The actual transmission phase is done similarly, by reading headers in a loop, switching on the message type and reading/sending the relevant data/reply
- `TRANSMISSION_TYPE_REQUEST_READ` executes a read request on the backend and sends the relevant chunk to the client (https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L331-L351)
- `TRANSMISSION_TYPE_REQUEST_WRITE` reads an offset and chunk from the client, then writes it to the client; if the read-only option is specified for the server, a permission error is returned to the client instead (https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L353-L368)
- `TRANSMISSION_TYPE_REQUEST_DISC` gracefully disconnects from the server and causes the backend to sync, i.e. to flush it's changes to the disk

- Client
  - The NBD client is implemented by using the kernel NBD client and a userspace component
  - In order to use it, one needs to find a free NBD device allocated by the kernel NBD module first
  - NBD devices are pre-created by the NBD kernel module and more can be specified with the `nbds_max` parameter
  - In order to find a free one, we can either specify it directly, or check whether we can find a NBD device with zero size in `sysfs` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg
  - Client is started by calling `Connect` and a connection and options are provided
  - Role of options (https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L38-L45)
  - The connection is set on the kernel's NBD device (https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L182-L189)
  - Relevant `ioctl` numbers depend on the kernel and are extracted using `CGo` (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/ioctl/negotiation_cgo.go)
  - The handshake for the NBD client is negotiated in user space by the Go program
  - Client also only supports the newstyle negotiation and aborts otherwise
  - Handshake is implented as a simple for loop, basically the same as for the server but inverted, once again switches on the types
  - On a `NEGOTIATION_TYPE_REPLY_INFO`, the client handles the export size with `NEGOTIATION_TYPE_INFO_EXPORT`, the name, description and block size with `NEGOTIATION_TYPE_INFO_BLOCKSIZE`
  - Client validates that the blocksize is valid (within specified bounds, and a power of two https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L261-L277)
  - After the metadata for the export has been fetched in the handshake, the kernel NBD client is configured with these values using `ioctls` (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L290-L328)
  - `go-nbd` can also list exports with `List`
  - For this, the newstyle handshake is initiated, but this time `NEGOTIATION_ID_OPTION_LIST` is provided
  - The client reads the export names from the server and disconnects (https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L393-L410)
- Client Lifecycle
  - The `DO_IT` syscall never returns until it is disconnected, meaning that an external system must be used to detect whether the device is actually ready
  - Two ways of detecting whether the device is ready: By polling `sysfs` for the size parameter, or by using `udev`

  - udev manages devices in Linux
  - When a device becomes available, the kernel sends a udev event, which we can subscribe to and use as a reliable and idiomatic way of waiting for the ready state (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L104C10-L138)
  - In reality however, polling sysfs directly can be faster than subscribing to the udev event, so we give the user the option to switch between both options (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L140-L178)
  - Disconnecting the NBD client is also an asynchronous operation
  - Works by calling three ioctls: TRANSMISSION_IOCTL_CLEAR_QUE, TRANSMISSION_IOCTL_DIS and TRANSMISSION_IOCTL_CLEAR_SOCK (https://github.com/pojntfx/go-nbd/blob/main/pkg/client L359)
  - Calling these ioctls causes DO_IT to exit, thus terminating the call to Connect

- Optimizations for the NBD client implementation

  - When opening the block device that the client has connected to, usually the kernel does provide a caching mechanism and thus requires sync to flush changes
  - As mentioned earlier, by using O_DIRECT however, it is possible to skip the kernel caching layer and write all changes directly to the NBD client/server
  - This is particularly useful if both the client and server are on the local system, and if the amount of time spent on syncing should be as small as possible
  - It does however require reads and writes on the device node to be aligned to the system's page size, which is possible to implement with a client-side chunking system but does require application-specific code

- Combining the NBD client and server

  - The server and client are connected by creating a connected UNIX socket pair (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_direct.go#L59-L62)
  - This exposes a path mount, which essentially returns the path to the NBD device for further abstraction layers
  - By building on this basic direct mount, we can add a file (code snippet from https://github.com/pojntfx/r3m mount API
  - The file mount API allows accessing the remote ressource with the common read/write syscalls, like a local file would be accessed
  - The lifecycle is integrated with Open and Close for the device with both mounts, preventing potential synchronization pitfalls over just manually implementing it in each application
  - The slice API is another mount API (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/r

  that `mmap`s the NBD device directly
  - Using the `mmap`/slice approach has a few benefits
  - First, it makes it possible to use the byte slice directly as though it were a byte slice allocated by `make`, except its transparently mapped to the (remote) backend
  - `mmap`/the byte slices also swaps out the syscall-based file interface with a truly random access one, which allows for faster concurrent reads from the underlying backend
  - Also has the benefit of supporting applications that aren't build for streaming support/-files, but depend on storing/fetching data in a byte slice
  - Alternatively, it would also be possible to format the server's backend with a file system or the block device using standard file system tools
  - When the device then becomes ready, it can be mounted to a directory on the system
  - This way it is possible to `mmap` one or multiple files on the mounted file system instead of `mmap`ing the block device directly
  - This allows for handling multiple remote regions using a single server, and thus saving on initialization time and overhead
  - Using a proper file system however does introduce both storage overhead and complexity, which is why e.g. the FUSE approach was not chosen

4.5 Mounts

- The `ReadWriterAt` pipeline

  - In order to implement the chunking system, we can use a abstraction layer that allows us to create a pipeline of readers/writers - the `ReadWriterAt`, combining an `io.ReaderAt` and a `io.WriterAt`
  - This way, we can forward the `Size` and `Sync` syscalls directly to the underlying backend, but wrap a backend's `ReadAt` and `WriteAt` methods in a pipeline of other `ReadWriterAt`s
  - One such `ReadWriterAt` is the `ArbitraryReadWriterAt` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/arbitrary_rwat.go)
  - It allows breaking down a larger data stream into smaller chunks
  - In `ReadAt`, it calculates the index of the chunk that the offset falls into and the position within the offsets
  - It then reads the entire chunk from the backend into a buffer, copies the necessary portion of the buffer into the input slice, and repeats the process until all requested data is read
  - Similarly for the writer, it calculates the chunk's index and offset
  - If an entire chunk is being written to, it bypasses the chunking system, and writes it directly to not have to unnecessarily copy the data twice

  - If 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 has changed, and writes the entire chunk back, until all data has been written
  - This simple implementation can be used to allow for writing data of arbitrary length at arbitrary offsets, even if the backend only supports a few chunks
  - In addition to this chunking system, there is also a `ChunkedReadWriterAt`, which ensures that the limits concerning the maximum size supported by the backend and the actual chunks are being respected
  - This is particularly useful when the client is expected to do the chunking, and the server simply checks that the chunking system's chunk size is respected
  - In order to check if a read or write is valid, it checks whether a read is done to an offset multiple of the chunk size, and whether the length of the slice of data to read/write is the chunk size (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/chunked_rwat.go)

- Background pull

  - The `Puller` component asynchronously pulls chunks in the background (code snipped from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/puller.go)
  - After sorting the chunks, the puller starts a fixed number of worker threads in the background, each of which ask for a chunk to pull
  - Note that the puller itself does not copy to/from a destination; this use case is handled by a separate component
  - It simply reads from the provided `ReaderAt`, which is then expected to handle the actual copying on its own
  - The actual copy logic is provided by the `SyncedReadWriterAt` instead
  - This component takes both a remote reader and a local `ReadWriterAt`
  - If a chunk is read, e.g. by the puller component calling `ReadAt`, it is tracked and market as remote by adding it to a local map
  - The chunk is then read from the remote reader and written to the local `ReadWriterAt` , and is then marked as locally available, so that on the second read it is fetched locally directly
  - A callback is then called which can be used to monitor the pull process
  - Note that if it is used in a pipeline with the `Puller`, this also means that if a chunk which hasn't been fetched asynchronously yet will be scheduled to be pulled immediately
  - WriteAt also starts by tracking a chunk, but then immediately marks the chunk as available locally no matter whether it has been pulled before
  - The combination of the `SyncedReadWriterAt` and the `Puller` component implements the pull post-copy system in a modular and testable way
  - Unlike the usual way of only fetching chunks when they are available however, this system

also allows fetching them pre-emptively, gaining some benefits of pre-copy migration, too
- Using this `Puller` interface, it is possible to implement a read-only managed mount
- This is very similar the `rr`+ prefetching mechanism from "Remote Regions" (reference atc18-aguilera)

- Background push
  - Once opened, the pusher starts a new goroutine in the background which calls `Sync` in a set recurring interval (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/pushe
  - Once sync is called by this background worker system or manually, it launches workers in the background
  - These workers all wait for a chunk to handle
  - Once a worker receives a chunk, it reads it from the local `ReadWriter`, and copies it to the remote
  - In order to actually mark chunks that are pushable (chunks that have been written to or have already been synced from the remote), the progress callback is used to call the pusher's `MarkOffsetPushable` method (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L171C24-L185)
  - Only these marked chunks are being considered to be pushed when a write occurs, so as to prevent syncing chunks that have not yet been made locally available
  - The pusher also serves as a step in the `ReadWriterAt` pipeline
  - In order to do this, it exposes `ReadAt` and `WriteAt`
  - `ReadAt` is a simple proxy, while `WriteAt` also marks a chunk as pushable (since it mutates data) before writing to the local `ReadWriterAt`
  - For a read-only scenario, the `Pusher` step is simply skipped (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L142-L169)

- Benefits of managed mounts over direct mounts
  - For the direct mount, the NBD server was directly connected to the remote, while in this setup a pipeline of pullers, pushers, a syncer and an `ArbitraryReadWriter` is used (graphic of the four systems and how they are connected to each other vs. how the direct mounts work)
  - Using this simple interface also makes the entire system very testable
  - In the tests, a memory reader or file can be used as the local or remote `ReaderWriterAt` s and a simple table-driven test can be created (code snippet from https://github.com/pojntfx/r3map/blob/
  - Thanks to making these individual components (background pull, push, chunking, chunk validation) unit-testable on their own, edge cases (like different pull heuristics) can be

tested easily, too

- Parallelizing NBD device initialization
  - The background push/pull system allows pulling from the remote `ReadWriterAt` before the NBD device is open
  - This is possible because the device doesn't need to start accessing the data before it can start pulling
  - As a result we can start pulling in the background as the NBD client and server are still starting (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L25 L272)
  - These two components typically start fairly quickly, but can still take multiple ms
  - Often, it takes as long as one RTT, so parallelizing this startup process can significantly reduce the initial latency and pre-emptively pull quite a bit of data
  - If no background pulls are enabled, the creation of the `Puller` is simply skipped (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L187-L222)
- Implementing device lifecycles
  - Similarly to how the direct mounts API used the basic path mount to build the file and `mmap` interfaces, the managed mount builds on this interface in order to provide the same interfaces
  - It is however a bit more complicated for the lifecycle to work
  - For example, in order to allow for a `Sync()` API, e.g. the `msync` on the `mmap`ed file must happen before `Sync()` is called on the syncer
  - This is done through a hooks system (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg L37)
  - The same hooks system is also used to implement the correct lifecycle when `Close`ing the mount
  - With all of these components in place, the managed mounts API serves as a fast and efficient option to access almost any remote resource in memory
- Optimization for WAN
  - Another optimization that has been made to support this WAN deployment scenario is the pull-only architecture
  - Usually, a pre-copy system pushes changes to the destination in the migration API
  - This however makes such a system hard to use in a scenario where NATs exist, or a scenario in which the network might have an outage during the migration
  - With a pull-only system emulating the pre-copy setup, the client can simply keep track of which chunks it still needs to pull itself, so if there is a network outage, it can just resume

pulling like before, which would be much harder to implement with a push system as the server would have to track this state for multiple clients and handle the lifecycle there
- The pull-only system also means that unlike the push system that was implemented for the hash-based synchronization, a central forwarding hub is not necessary
- In the push-based system for the hash-based solution, the topology had to be static/all destinations would have to have received the changes made to the remote app's memory since it was started
- In order to cut down on unnecessary duplicate data transmissions, a central forwarding hub was implemented
- This central forwarding hub does however add additional latency, which can be removed completely with the migration protocol's P2P, pull-only algorithm

4.6 Live Migration

- Overview
  - As mentioned in Pull-Based Synchronization with Migrations before, the mount API is not optimal for a migration scenario
  - Splitting the migration into two separate phases can help a lot to fix the biggest problem, the maximum guaranteed downtime
  - The flexible architecture of the `ReadWriterAt` components allow the reuse of lots of code for both the mount API and the migration API
- Implementing the seeder
  - The seeder defines a simple read-only RPC API with the familiar `ReadAt` methods, but also new APIs such as returning dirty chunks from `Sync` and adding a `Track` method (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go#L15-L21)
  - Unlike the remote backend, a seeder also exposes a mount through a path, file or byte slice, so that as the migration is happening, the underlying data can still be accessed by the application
  - This fixes the issue that the mount API had for migrations, where only one end of the API exposed it's mount, while the other part simply served data
  - The tracking aspect is implemented in the same modular and composable way as the syncer etc. - by using a `TrackingReadWriterAt` that is connected to the seeder's `ReadWriterAt` pipeline
  - Once activated by `Track`, the tracker intercepts all `WriteAt` calls and adds them to a local de-duplicated store (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/trac L40)

  - When `Sync` is called, the changed chunks are returned and the de-duplicated store is cleared
  - Insert graphic of the pipeline here
  - A benefit of the protocol being defined in such a way that only the client ever calls an RPC, not the other way around, is that the transport layer/RPC system is completely interchangeable
  - This works by returning a simple abstract `service` utility struct struct from `Open` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go)
  - This abstract struct can then be used as the backend for any RPC framework, e.g. for gRPC (code snippet https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder_grpc.go)

- Implementing the leecher

  - The leecher then takes this abstract API as an argument
  - As soon as the leecher is opened, it calls track on the seeder, and in parallel starts the device (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/migration/path_leecher.go#L1 L203)
  - The leecher introduces a new component, the `LockableReadWriterAt`, into its internal pipeline (add graphic of the internal pipeline)
  - This component simply blocks all read and write operations to/from the NBD device until `Finalize` has been called (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks L37)
  - This is required because otherwise stale data (since `Finalize` did not yet mark the changed chunks) could have poisoned the cache on the e.g. `mmap`ed device
  - Once the leecher has started the device, it sets up a syncer (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/migration/path_leecher.go#L214-L252)
  - A callback can be used to monitor the pull process (code snippet from https://github.com/pojntfx/r3map/b migration-benchmark/main.go#L544-L548)
  - As described before, after a satisfactory local availability level has been reached, `Finalize` can be called
  - `Finalize` then calls `Sync()` on the remote, marks the changed chunks as remote, and schedules them also be pulled in the background (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/migration/path_leecher.go#L257-L280)
  - As an additional measure aside from the lockable `ReadWriterAt` to make accessing the path/file/slice too early harder, only `Finalize` returns the managed object, so that the happy path can less easily lead to deadlocks
  - After a leecher has successfully reached 100% local availability, it calls `Close` on the

  seeder and disconnects the leecher from the seeder, causing both to shut down (code snippet from https://github.com/pojntfx/r3map/blob/main/cmd/r3map-migration-benchmark-server/main.go#L137)
  - Once the leecher has exited, a seeder can be started, to allow for migrating from the destination to another destination again

4.7 Pluggable Encryption and Authentication

- Compared to existing remote mount and migration solutions, r3map is a bit special
- As mentioned before, most systems are designed for scenarios where such resources are accessible in a high-bandwidth, low-latency LAN
- This means that some assumptions concerning security, authentication, authorization and scalability were made that can not be made here
- For example encryption; while for a LAN deployment scenario it is probably assumed that there are no bad actors in the subnet, the same can not be said for WAN
- While depending on e.g. TLS etc. for the migration could have been an option, r3map should still be useful for LAN migration use cases, too, which is why it was made to be completely transport-agnostic
- This makes adding encryption very simple
- E.g. for LAN, the same assumptions that are being made in existing systems can be made, and fast latency-sensitive protocols like the SCSI RDMA protocol (SRP) or a bespoke protocol can be used
- For WAN, a standard internet protocol like TLS over TCP or QUIC can be used instead, which will allow for migration over the public internet, too
- For RPC frameworks with exchangeable transport layers such as dudirekta (will be explained later), this also allows for unique migration or mount scenarios in NATed environments over WebRTC data channels, which would be very hard to implement with more traditional setups
- Similarly so, authentication and authorization can be implemented in many ways
- While for migration in LAN, the typical approach of simply trusting the local subnet can be used, for public deployments mTLS certificates or even higher-level protocols such as OIDC can be used depending on the transport layer chosen
- For WAN specifically, new protocols such as QUIC allow tight integration with TLS for authentication and encryption
- While less relevant for the migration use case (since connections can be established ahead of time), for the mount use case the initial remote `ReadAt` requests' latency is an important metric since it strongly correlates with the total latency
- QUIC has a way to establish 0-RTT TLS, which can save one or multiple RTTs and thus signficantly reduce this overhead, and handle authentication in the same step

4.8 Optimizing Backends For High RTT

- In WAN, where latency is high, the ability to fetch chunks concurrently is very important
- Without concurrent background pulls, latency adds up very quickly as every memory request would have at least the RTT as latency
- The first prerequisite for supporting this is that the remote backend has to be able to read from multiple regions without locking the backend globally
- For the file backend for example, this is not the case, as the lock needs to be acquired for the entire file before an offset can be accessed (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/backend/file.go#L17-L25)
- For high-latency scenarios, this can quickly become a bottleneck
- While there are many ways to solve this, one is to use the directory backend
- Instead of using just one backing file, the directory backend is a chunked backend that uses a directory with one file for each chunk instead of a global file
- This means that the directory backend can lock each file individually, speeding up concurrent access
- This also applies to writes, where even concurrent writes to different chunks can be done at the same time as they are all backed by a separate file
- The directory backend keeps track of these chunks by using an internal map of locks (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/directory.go#L22-L24)
- When a chunk is first accessed, a new file is created for the chunk (code snipped from https://github.com/pojntfx/r3map/blob/main/pkg/backend/directory.go#L77-L94)
- If the chunk is being read, the file is also truncated to one chunk length
- Since this could easily exhaust the number of maximum allowed file descriptors for a process, a check is added (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/directory.go#L5 L75)
- If the maximum allowed number of open files is exceeded, the first file is closed and removed from the map, causing it to be reopened on a subsequent read

4.9 Using Remote Stores as Backends

- Using key-value stores as ephemeral mounts
  - RPC backends provide a way to access a remote backend
  - This is useful, esp. if the remote resource should be protected in some way or if it requires some kind of authorization
  - Depending on the use case however, esp. for the mount API, having access to a remote backend without this level of indirection can be useful

  - Fundamentally, a mount maps fairly well to a remote random-access storage device
  - Many existing protocols and systems provide a way to access essentially this concept over a network
  - One of these is Redis, an in-memory key-value store with network access
  - Chunk offsets can be mapped to keys, and bytes are a valid key type, so the chunk itself can be stored directly in the KV store (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/ba L63)
  - Using Redis is particularly useful because it is able to handle the locking server-side in a very efficient way, and is well-tested for high-throughput etc. scenarious
  - Redis also has very fast read/write speeds due to its bespoke protocol and fast serialization
  - Authentication can also be handled using the Redis protocol, so can multi-tenancy by using multiple databases or a prefix

- Mapping large resources into memory with S3

  - While the Redis backend is very useful for read-/write usage, when deployment to the public internet is required, it might not be the best one
  - The S3 backend is an good choice for mapping public information, e.g. media assets, binaries, large read-only filesystems etc. into memory
  - S3 used to be an object storage service from AWS, but has since become a more or less standard way for accessing blobs thanks to open-source S3 implementations such as Minio
  - Similarly to how files were used as individual files, one S3 object per chunk is used to store them
  - S3 is based on HTTP, and like the Redis backend requires chunking due to it not supporting updates of part of a file
  - In order to prevent having to store empty data, the backend interprets “not found” errors as empty chunks

- Document databases as persistent mount remotes

  - Another backend option is a NoSQL server such as Cassandra
  - Specifically, as noted earlier, ScyllaDB - which improves upon Cassandra’s latency, which is important for the mount API
  - This is more of a proof of concept than a real usecase, but shows the versitility and flexibility of how a database can be mapped to a memory region, which can be interesting for accessing e.g. a remote database’s content without having to use a specific client
  - `ReadAt` and `WriteAt` are implemented using Cassandra’s query language (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/cassandra.go#L36-L63)
  - Similarly to Redis, locking individual keys can be handled by the DB server

  - But in the case of Cassandra, the DB server stores chunks on disk, so it can be used for persistent data
  - In order to use Cassandra, migrations have to be applied for creating the table etc. (code snippet from https://github.com/pojntfx/r3map/blob/main/cmd/r3map-direct-mount-benchmark/main.go#L369-L396)

4.10 Bi-Directional and Concurrent RPCs with Dudirekta

- Overview of the framework and why a custom one was implemented
  - Designed specifically with the hybrid pre- and post-copy scenario in mind
  - Support for concurrent RPCs allows for efficient background pulls since multiple chunks can be pulled at the same time
  - Bi-directional API makes it possible to initiate pre-copy migrations and transfer chunks from the source host without having to make the destination host `dial`able
    - Dudirekta supports defining functions on both the client and the server
    - This is very useful for implementing e.g. a pre-copy protocol where the source pushes chunks to the destination by simply calling a RPC on the destination, instead of the destination calling a RPC on the source
    - Usually, RPCs don't support exposing or calling RPCs on the client, too, only on the server
    - This would mean that in order to implement a pre-copy protocol with pushes, the destination would have to be `dial`able from the source
    - In a LAN scenario, this is easy to implement, but in WAN it is complicated and requires authentication of both the client and the server
  - Transport independence allows for i.e. QUIC to be used for the 0-RTT handshake in WAN
  - Support for passing callbacks and closures to RPCs makes it possible to model remote generators and yields to easily report i.e. remote migration progress
  - Because it reflection-based, both RPC definition and calling an RPC are both transparent, making it optimal for prototyping r3map
- Usage (as described in Dudirekta README)
- Protocol definition (JSONL, structure etc.) and how it is multiplexed
- RPC provider implementation
  - If an RPC, such as `ReadAt`, is called, it is looked up via reflection and validated (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L323-L357)

  - The arguments, which have been supplied as JSON, are then unmarshalled into their native types, and the local wrapper struct's method is called in a new goroutine (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L417-L521)

- RPC call implementation
  - To use a RPC backend on the destination side, the wrapper struct's remote representation is used (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/backend.go#L14-L19)
  - For the destination site, the remote representation's fields are iterated over, and replaced by functions which marshal and unmarshal the function calls into the dudirekta JSON protocol (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L228-L269)
  - To do this, the arguments are marshalled into JSON, and a unique call ID is generated (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L109-L130)
  - Once the remote has responded with a message containing the unique call ID, it unmarshalls the arguments, and returns (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pk
L217)
- Closure implementation
- Using Dudirekta for r3map
  - Since dudirekta has a few limitations (such as the fact that slices are passed as copies, not references, and that context needs to be provided), the resulting remote struct can't be used directly
  - To work around this, the standard go−nbd backend interface is implemented for the remote representation, creating a universally reusable, generic RPC backend wrapper (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/rpc.go)
  - The same backend implementation is also done for the seeder protocol (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go)

4.11 Connection Pooling with gRPC

- Drawbacks of Dudirekta
  - While the dudirekta RPC serves as a good reference implementation of the basic RPC protocol, it does not scale particularly well
  - This mostly stems from two aspects of how it is designed
  - JSON(L) is used for the wire format, which while simple and easy to analyze, is slow to marshal and unmarshal

  - Dudirekta's bi-directional RPCs do however come at the cost of not being able to do connection pooling, since each client `dial`ing the server would mean that the server could not reference the multiple client connections as one composite client without changes to the protocol
  - While implementing such a pooling mechanism in the future could be interesting, it turned out to not be necessary thanks to the pull-based pre-copy solution described earlier
  - Instead, only calling RPCs exposed on the server from the client is the only requirement for an RPC framework, and other, more optimized RPC frameworks can already offer this

- Switching to code generation

  - Dudirekta uses reflection to make the RPCs essentially almost transparent to use
  - By switching to a well-defined protocol with a DSL instead, we can gain further benefits from not having to use reflection and generating code instead
  - One popular such framework is gRPC
  - gRPC is a high-performance RPC framework based on Protocol Buffers
  - Because it is based on Protobuf, we can define the protocol itself in the `proto3` DSL
  - The DSL also allows to specify the order of each field in the resulting wire protocol, making it possible to evolve the protocol over time without having to break backwards compatibility
  - This is very important given that r3map could be ported to a language with less overhead in the future, e.g. Rust, and being able to re-use the existing wire protocol would make this much easier

- Benefits of gRPC over Dudirekta

  - While dudirekta is a simple protocol that is easy to adapt for other languages, currently it only supports Go and JS, while gRPC supports many more
  - A fairly unique feature of gRPC are streaming RPCs, where a stream of requests can be sent to/from the server/client, which, while not used for the r3map protocol, could be very useful to implementing a pre-copy migration API with pushes similarly to how dudirekta does it by exposing RPCs from the client
  - As mentioned before, for WAN migration or mount scenarios, things like authentication, authorization and encryption are important, which gRPC is well-suited for
  - Protobuf being a proper byte-, not plaintext-based wire format is also very helpful, since it means that e.g. sending bytes back from `ReadAt` RPCs doesn't require any encoding (wereas JSON, used by dudirekta, `base64` encodes these chunks)
  - gRPC is also based on HTTP/2, which means that it can benefit from existing load balancing tooling etc. that is popular in WAN for web uses even today

- Implementation of the gRPC backend

  - This backend is implemented by first defining the protocol in the DSL (code snippet from https://github.com/pojntfx/r3map/blob/main/api/proto/migration/v1/seeder.proto)
  - After generating the bindings, the generated backend interface is implemented by using the dudirekta wrapper struct as the abstraction layer (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder_grpc.go)
  - Unlike dudirekta, gRPC also implements concurrent RPCs and connection pooling
  - Similarly to how having a backend that allows concurrent reads/writes can be useful to speed up the concurrent push/pull steps, having a protocol that allows for concurrent RPCs can do the same
  - Connection pooling is another aspect that can help with this
  - Instead of either using one single connection with a multiplexer (which is possible because it uses HTTP/2) to allow for multiple requests, or creating a new connection for every request, gRPC is able to intelligently re-use existing connections for RPCs or create new ones, speeding up parallel requests

4.12 Optimizing RPC Throughput and Latency with fRPC

- Despite these benefits, gRPC is not perfect however
- Protobuf specifically, while being faster than JSON, is not the fastest serialization framework that could be used
- This is especially true for large chunks of data, and becomes a real bottleneck if the connection between source and destination would allow for a high throughput
- This is where fRPC, a RPC library that is easy to replace gRPC with, becomes useful
- Because throughput and latency determine the maximum acceptable downtime of a migration/the initial latency for mounts, choosing the right RPC protocol is an important decision
- fRPC also uses the same proto3 DSL, which makes it an easy drop-in replacement, and it also supports multiplexing and connection polling
- Because of these similarities, the usage of fRPC in r3map is extremely similar to gRPC (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder_frpc.go)

5 Results

5.1 Testing Environment

- Hardware specifications
- Benchmark scripts are reproducible (add link here)
- Multiple runs were done for each measurement to ensure consistenly

5.2 Access Methods

- Latency for technologies
  - Compared to disk and memory, all three other access methods (userfaultfd, direct mounts, managed mounts) are slow
  - Latency of accessing a chunk on userfaultfd is 15x slower than on disk
  - Latency of accessing a chunk on direct mounts is 28x slower than on disk, almost double the latency of `userfaultfd`
  - Managed mounts is 40x slower than disk
  - It is however important to consider the total scale: All of these are in the scale of µs
  - Interesting to see that managed mounts have a significantly higher latency
  - Looking at the distribution, we can see a similar pattern
  - Spread for managed mount is the smallest, but there are significant outliers up until 1750 µs
  - Direct mounts has the highest spread of the access methods
  - Up until now we took a look at 0ms RTT, meaning that we connected the backend to the mount/userfault handler directly
  - If we take a look at how it behaves as the RTT changes, we get a different story
  - Both userfaultfd and direct mounts first chunk latency grows linearly
  - For managed mount, the latency gets slightly higher up to a RTT of 25ms, but even then max. 1/2 the RTT of the latency that is expected compared to the direct mount of userfaultfd
  - After 25ms, the first chunk latency is the same or lower as it was for 0 RTT
  - Similar pattern if workers are tuned for managed mounts
  - For zero workers, latency grows almost linearly with the RTT
  - If more than 0 workers are used, the latency keeps growing almost linearly, until it peaks at 10ms and then drops down to the pre-0ms value
- Throughput for technologies
  - For throughput of the memory, disk, userfaultfd, direct mounts and managed mounts access methods, we can see similar trends but some distinctions
  - Memory has the highest throughput at over 20 GB/s
  - This however is not followed by the disk as it was in latency, but rather by the direct mounts (at 2.8GB/s) and managed mounts (2.4GBs)
  - The disk is at 2.3GB/s, while userfaultfd falls behind at 450 MB/s
  - If we only look at userfaultfd, managed mounts and direct mounts this different between the access methods becomes more clear

  - When it comes to the throughput distribution, userfaultfd has the lowest spread, while managed mounts has the highest, closely followed by direct mounts
  - Interestingly, the median of direct mounts is especially high compared to the other access methods
  - As with latency, the results are very different if the RTT increases
  - Direct mounts and userfaultfd drop to below 10MB/s and 1MB/s respectively after the RTT reaches 6ms and continue to drop as the RTT increases
  - Managed mounts also drop, but much less drastically, even at a RTT of 25ms the throughput is still at over 500MB/s
  - If the RTT is lower than 10ms, almost 1GB/s can still be achieved with managed mounts
  - Similar results to what was measured with latency can be seen with different pull worker counts for the managed mount as RTT increases
  - Low worker counts have good performance at 0ms RTT
  - As RTT increases, the higher the pull worker count, the higher the throughput continues to be
  - For 16384 workers, throughput can be kept at consistently over 1GB/s even at 30ms latency

- Write throughput for technologies

  - So far we've looked at read throughput
  - For a migration/remote access usecase that mostly makes sense
  - But for completeness, data for write throughput is also available
  - Note that the file is opened with O_DIRECT for this benchmark, causing additional overhead when writing, but also - for benchmark purposes - don't require `msync`
  - As RTT increases, managed mounts have much better write performance
  - Write speed for direct mounts drop down to below 1 MB/s after 4ms, while they are consistenly above 230 MB/s for managed mounts, independent of RTT

5.3 Initialization

- Mount initialization time

  - There are two diffent ways to check for whether a device is ready: Polling or subscribing to the udev events
  - Initialization time for udev is higher on average than the polling method
  - Spread is similar for both methods

- Pre-emptive pulls

  - When looking at pre-emptive pulls as RTT increases, the role of workers becomes apparent

  - The higher the worker count is, the more data can be pulled
  - While for 4096 workers, almost 40 MB can be pulled pre-emptively at 7ms, this drops to 20 MB for 2048 workers, 5 MB for 512 workers and continues to drop
  - Even for a RTT of 0ms, more workers mean more pre-emptively pulled data in general

5.4 Chunking

- Server vs. client-side chunking
  - Chunking can be done on either client- or server-side for both direct and managed mount
  - It is clear that unless the RTT is 0, managed mounts yield significantly higher throughput than direct mounts of both server- and client side chunking
  - When looking at the direct mounts, server-side chunking is very fast for 0ms RTT at almost 500 MB/s throughput, but drops very quickly to 75MB/s at 1ms, 40MB/s at 2ms, and continues to drop down to just over 5MB/s at 20ms RTT
  - For client-side chunking, the result is lower at just 30MB/s even at a 0ms RTT, and then continues to drop steadily it reaches 4.5MB/s at 20ms RTT
  - Looking at managed mounts the scenario is very different
  - Throughput also declines as RTT increases, but less drastically
  - Server-side chunking has much higher throughputs, at 450MB/s at 0ms RTT vs. 230MB/s at 0ms for the client-side chunking
  - As RTT increases, both direct and managed backends drop following a similar pattern, dropping to 300 MB/s for managed mounts and 240MB/s for direct mounts at 20ms RTT

5.5 RPC frameworks

- Throughput for RPC frameworks
  - Looking at the performance for dudirekta, gRPC and fRPC for managed and direct mounts, quite drastic throughput differences can be seen as the RTT increases
  - While for 0 RTT, the direct mounts provide the best throughput in line with the measurements for the different access technologies, the throughput drops drastically as RTT increases compared to managed mounts
  - In general, dudirekta has much lower throughput than both gRPC and fRPC
  - When looking at direct mounts specifically, the sharp difference between dudirekta and gRPC/RPC is visible
  - While at 0 RTT, fRPC is at 390MB/s and gRPC is at 500MB/s, dudirekta reaches just 50MB/s
  - At 2ms, throughput for all RPC frameworks drop drastically as the RTT increases, with fRPC and gRPC both reaching 40MB/s, and Dudirekta dropping to just 20MB/s

  - All three RPC frameworks reach a throughput of just 7 MB/s at 14ms, and then continue to decrease until they reach 3 MB/s at 40ms
  - For managed mounts, there is a similarly stark difference between dudirekta and the other RPC frameworks
  - Dudirekta's throughput stays consistently low, at an average of 45MB/s, interestingly performing better in the >20ms RTT section
  - Like in direct mounts, gRPC manages to outperform fRPC at 0ms RTT at 395MB/s vs. 250MB/s, but fRPC gets consistenly higher values starting at 2ms RTT
  - fRPC has a better throughput than gRPC on average, managing to keep above 300MB/s throughput until 25ms RTT, while gRPC drops to below this after 14ms RTT
  - Average throughput of fRPC is higher, but as RTT reaches 40 it becomes less drastic at a max. of 50MB/s difference after 28ms RTT is reached

5.6 Backends

- Latency for backends
  - Average first chunk latency for memory, file, directory, Redis and Cassandra backends at 0ms RTT are all similar and within µs range
  - The latency for accessing Redis is the lowest at 2.5µs
  - Looking at the latency distribution, Redis once again is visible as having both the smallest amount of spread and the lowest amount of latency
  - Memory and S3 stick out for having a low amount of outlyers when it comes to first chunk latency, while the directory backend is notable for it's significant spread
- Throughput for backends
  - When looking at throughput, the backends have significantly more different characteristics compared to latency
  - File and memory consistently have high throughput
  - For direct mounts, file throughput is higher than memory througput at 2081 MB/s vs 1630MB/s on average respectively
  - For managed mounts, this increases to 2372MB/s vs. 2178MB/s respectively
  - When comparing the direct vs. managed mount values, Cassandra and the remote backends in general show vast differences in throughput
  - Cassandra reaches almost 700MB/s for a managed mount, while it falls to 3MB/s when using a direct one
  - Similarly so, for Redis and the directory backend, direct mounts are roughly 3.5 times slower than the managed mounts, while for S3 it is 5.5 times slower

- When looking at the throughput for the direct mounts with the option of having remote access, Redis has a much higher average throughput at 114MB/s compared to both Cassandra at 3MB/s and S3 at 8MB/s
- For managed mounts, Cassandra performs better than both S3 and Redis, managing 689MB/s compared to 439MB/s for Redis and 44 MB/s for S3
- The throughput distribution for the different backends with direct mounts is similarly interesting
- A logarithmic Y axis is used to show the kernel density estimation for Redis, which has a much wider spread compared to Redis and S3, but is also consistently higher than both even with this spread
- For managed mounts, Cassandra has a high spread but also the highest throughput, while Redis has the lowest
- S3 is also noticable here again for having a consistently lower througput compared to both alternatives, with an average spread
- When looking at average throughput for direct mounts, all backends see their throughput drop drastically as RTT increases
- Memory and file have vey high throughputs above 1.4GB/s at 0MS RTT, while Redis achieves just 140MB/s as the closest network-capable alternative
- Directory noticably has a lower throughput than Redis, despite not being network-capable
- All other technologies are at below 15MB/s for direct mounts, even at 0ms RTT, and all technologies towards below 3MB/s at 40ms for direct mounts
- When looking at the network-capable backends in isolation, the difference between Redis and the others in direct mount performance is striking, but all trend towards low throughput as the RTT increases
- For managed mounts, the memory and file backends again outperform all other options at over 2.5GB/s, while the closest network-capable technology reaches just 660MB/s
- Similarly to the latency, all technologies trend towards a similar throughput as RTT increases, with sharp drops for the memory and file backends after 2ms
- Noticably, both the directory and S3 backends underperform, even for managed mounts, dropping down to just 55MB/s at 40ms RTT
- When looking at just the network-capable backends, it is clear again that S3 underperforms drastically
- Both Redis and Cassandra start between 550-660MB/s at 0 RTT, and then begin to drop after 6ms RTT until the drop to 170-180MB/s at 40ms RTT, with Redis consistently being slightly above the throughput for Cassandra

6 Discussion

6.1 Userfaults

- By using `userfaultfd` we are able to map almost any object into memory
- This approach is very clean and has comparatively little overhead, but also has significant architecture-related problems that limit its uses
- The first big problem is only being able to catch page faults - that means we can only ever respond the first time a chunk of memory gets accessed, all future requests will return the memory directly from RAM on the destination host
- This prevents us from using this approach for remote resources that update over time
- Also prevents us from using it for things that might have concurrent writers/shared resources, since there would be no way of updating the conflicting section
- Essentially makes this system only usable for a read-only "mount" of a remote resource, not really synchronization
- Only a viable solution for post-copy migration
- Also prevents pulling chunks before they are being accessed without additional layers of indirection
- The `userfaultfd` API socket is also synchronous, so each chunk needs to be sent one after the other, meaning that it is very vulnerable to long RTT values
- Also means that the initial latency will be at minimum the RTT to the remote source, and (without caching) so will be each future request
- Has faster acccess latency compared to direct mounts and managed mounts at 0 RTT
- Latency grows with RTT linearly as with direct mounts because there are no background pulls and no pre-emptive pulls like for managed mounts
- Significantly lower througput than direct mounts and managed mounts, esp. compared to managed mounts due to linear access
- Has a lower spread than direct mounts and managed mounts, but even with the more predictable throughput, its still consistently slower than both
- For high RTT, it becomes essentially unusable
- In summary, while this approach is interesting and very idiomatic to Go, for most data, esp. larger datasets and in high-latency scenarios/in WAN, we need a better solution

6.2 File-Based Synchronization

- Similarly to `userfaultfd`, this system also has limitations
- While `userfaultfd` was only able to catch reads, this system is only able to catch writes to the file

- Only a viable solution for pre-copy migration
- Essentially this system is write-only, and it is very inefficient to add hosts to the network later on
- As a result, if there are many possible destinations to migrate state too, a star-based architecture with a central forwarding hub can be used
- The static topology of this approach can be used to only ever require hashing on one of the destinations and the source instead of all of them
- This way, we only need to push the changes to one component (the hub), instead of having to push them to each destination on their own
- The hub simply forwards the messages to all the other destinations
- Thanks to this support for a star-based archicture, file-based synchronization might be a good choice for highly throughput-constrained networks, where the potentially higher maximum downtime is acceptable

6.3 FUSE

- FUSE can provide both a solution for pre- and post-copy migration
- FUSE also has downsides
- It operates in user space, which means that it needs to do context switching, esp. compared to a file system in kernel space
- Some advanced file system features aren't available for a FUSE
- The overhead of FUSE (and implementing a completely custom file system) for synchronizing memory is significant
- The optimal solution would be to not expose a full file system to track changes, but rather a single file
- As a result of this, the significant implementation overhead of such a file system led to it not being chosen, since NBD is available as an alternative

6.4 Direct Mounts

- Have a high spread/are unpredictable when from a first chunk latency perspective at 0ms RTT, but is more predictable when it comes to throughput
- First chunk latency grows linearly like userfaultfd as RTT increases, because there are no preemptive pulls
- Has the highest throughput at 0ms RTT, even higher than managed mounts, because the duplicate I/O operations for the background pull are not necessary

- Throughput doesn't drop as rapidly as userfaultfd, but still significantly because there is no background pulls, all data needs to be accessed one by one
- Write speed is subpar because writes need to be delivered directly remotely, as there are no background pushes compared to managed mounts
- Is a good solution if the internal overhead of managed mounts is higher than the overhead of the RTT, which can be the case in LAN or other low-latency networks

6.5 Managed Mounts

- Have an internal overhead due to duplicate I/O operations for background pull, resulting in worse throughput for low (esp. 0ms) RTT scenarios and higher inital latencies
- As soon as the RTT reaches levels more typical for a WAN deployment, this becomes
- Tuning the background workers can substantially increase the throughput values, since data can be accessed in parallel
- Pull priority function can allow for even more optimized pulls
- Pre-emptive pulls can significantly reduce initial chunk latency, since multiple MB can be pulled before the device is open
- Higher worker counts can significanlty increase the amount of chunks that can be pre-emptively pulled
- Generally speaking, polling is almost always the better choice for an initialization algorithm in order to reduce the overall `Open()` time and thus reduce the overhead of mounting a remote ressource
- Write throughput for managed mounts is significantly better than that of the direct mounts, since writes only go to the local backend, and are then asynchronously synced to the remote using the periodic background push system
- Is a good solution for WAN deployments, as soon as the RTT increases to a point that is high enough for it to balance out the internal overhead of the duplicate I/O operations

6.6 Chunking

- In general, server-side chunking should always be preferred due to the much better throughput
- For direct mounts, due to the linear access, the throughput is low for both server- and client-side chunking due to linear access as RTT increases, but server-side chunking is much more performant in this circumstance for low RTTs
- For managed mounts, client-side chunking can half the throughput of a mount compared to server-side chunking, mostly because if the chunks are smaller than the NBD block size, it decreases the amount of chunks that can be pulled relative if the worker count stays the same,

while server-side chunking doesn't require and extra worker on the client for each extra chunk that needs to be pulled, making it possible for the background pull system to pull more, increasing the throughput

6.7 RPC Frameworks

- Dudirekta
  - Consistently has lower throughput than the alternaties
  - Performs better for managed mounts than direct mounts thanks to support for concurrent RPCs
  - Less sensistive to RTT compared to gRPC and fRPC for managed mounts
  - Even with managed mounts, much worse throughput compared to both due to no connection pooling support
  - Remains a good option for prototyping due to lower development overhead and transport layer independence
- gRPC
  - Considerably faster than Dudirekta for managed and direct mounts
  - Has support for connection pooling, giving it a significant performance benefit over Dudirekta for managed mounts
  - Very good performance for direct mounts with 0ms RTT
  - Is an industry standard, and has good tooling and known scalability characteristics
- fRPC
  - Has consistently higher throughput than gRPC in managed mounts due to internal optimizations
  - Is faster than Dudirekta in both direct mounts and managed mounts due to support for connection pooling
  - Is much more minimal than gRPC/Protocol Buffers, making it more maintainable
  - Has less enterprise testing and tooling available, making it a more performant but also less proven solution

6.8 Backends

- Redis
  - Has the smallest amount of spread and lowest amount of initial chunk latency at 0ms RTT

  - Direct mounts have a lower throughput compared to managed mounts, showing good support for concurrent chunk access
  - Has the highest throughput for direct mounts by a significant margin due to its optimized protocol and fast key lookups
  - Is also the fastest network-capable backend for managed mounts, once again due to the good support for concurrent access
  - Good choice for ephermeral data like caches, where speed or the need for the direct mount API is important

- Cassandra

  - Has highest throughput in 0ms RTT deployments for direct mounts, showing very good concurrent access performance
  - Falls short when it comes to usage in direct mounts, where the performance is worse than any other backend, showing a high read latency overhead for looking up individual chunks
  - As RTT increases, has only slightly less throughput compared to Redis
  - Is a good choice if most data will be accessed by the managed mount's background pull system, but a bad choice if chunks will be accessed outside of this due to the low direct mount throughput
  - Is a good choice if the data should be persistent, since Cassandra can provide varying consistency guarantees, unlike Redis

- S3

  - Has the lowest performance of the network-capable backends that were implemented for managed mounts
  - Performance for managed mounts is consistently low even as RTT increases, presumably due to the high overhead of having to make HTTP requests to retrieve individual chunks
  - Does perform better than Cassandra for direct mounts
  - Is a good choice if S3 is the only choice due to architectural constraints or if the chance of chunks being read outside of the managed mount's background pull system is high, where Cassandra has worse throughput

6.9 NBD

- Limitations of NBD and `ublk` as an alternative

  - NBD is the underlying technology for both direct and managed mounts
  - NBD is a battle-tested solution for this with fairly good performance
  - In the future a more lean implemenation called `ublk` could also be used

  - `ublk` uses `io_uring`, which means that it could potentially allow for much faster concurrent access
  - It is similar to NBD; it also uses a user space server to provide the block device backend, and a kernel `ublk` driver that creates `/dev/ublkb*` devices
  - Unlike as it is the case for the NBD kernel module, which uses a rather slow UNIX or TCP socket to communicate, `ublk` is able to use `io_uring` pass-through commands
  - The `io_uring` architecture promises lower latency and better throughput
  - Because it is however still experimental and docs are lacking, NBD was chosen

6.10 Language Limitations

- Issues with the r3map implementation
  - While the managed mounts API mostly works, there are some issues with it being implemented in Go
  - This is mostly due to deadlocking issues; if the GC tries to release memory, it has to stop the world
  - If the `mmap` API is used, it is possible that the GC tries to manage the underlying slice, or tries to release memory as data is being copied from the mount
  - Because the NBD server that provides the byte slice is also running in the same process that is consuming the byte slice, this causes a deadlock as the server that provides the backend for the mount is also frozen (https://github.com/pojntfx/r3map/blob/main/pkg/mount/slice_managed.go L93)
  - A workaround for this is to lock the `mmap`ed region into memory, but this will also cause all chunks to be fetched, which leads to a high `Open()` latency
  - This is fixable by simply starting the server in separate process or other context where the GC does not cause it to stop, and then `mmap`ing
  - Issues like this however are hard to fix, and point to Go potentially not being the correct language to use for this part of the system
  - In the future, using a language without a GC (such as Rust) could provide a good alternative
  - While the current API is Go-specific, it could also be exposed through a different interface to make it usable in Go

6.11 Use Cases

6.11.1 Remote Swap With `ram-dl`

- ram-dl is a fun experiment

- Is a tech demo for r3map
- Uses the fRPC backend to expand local system memory
- Allows mounting a remote system's RAM locally
- Can be used to inspect a remote system's memory contents
- Is based on the direct mount API
- Uses mkswap, swapon and swapoff (code snippet from https://github.com/pojntfx/ram-dl/blob/main/cmd/ram-dl/main.go#L170-L190)
- Enables paging out to the block device provided by the direct mount API
- `ram-ul` "uploads" RAM by exposing a memory, file or directory-backed file over fRPC
- `ram-dl` then does all of the above
- Not really intended for real-world usecases, but does show that this can be used for interesting usecases such as real, remote RAM/Swap
- Shows how easy it is to use the resulting library, as the entire project is just ~300 SLOC including backend configuration, flags and other boilerplate

6.11.2 Mapping Tape Into Memory With `tapisk`

- Overview
  - tapisk is an interesting usecase because of how close it is to STFS, which provided the inspiration for the FUSE-based approach
  - Very high read/write backend latency (multiple seconds, up to 90s, due to seeking)
  - Linear access, no random reads
  - Can benefit a lot from asynchronous writes provided by managed mounts
  - Fast storage acts as a caching layer
  - Backend is linear, so only one read/write possible at a time
  - With local backend, writes are de-duplicated automatically and both can be asynchronous/concurrent
  - Writes go to fast ("local") backend first, syncer then handles in both directions
  - Only one concurrent reader/writer makes sense
  - Syncing intervals to/from can maybe be minutes or more to make it more efficient (since long, connected write intervals prevent having to seek to a different position on disk)
- Implementation
  - Chunking works on tape drive records and blocks
  - `bbolt` DB is used as an index
  - Index maps simulated offsets to real tape records (code snippet from https://github.com/pojntfx/tapisk/bl
    L68)

  - When reading, first the tape looks up the real tape record for the requested offset (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/backend/tape.go#L71-L78)
  - Seeking can then use the accelerated `MTSEEK` ioctl to fast-forward to a record on the tape (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/mtio/tape.go#L25-L40)
  - After seeking to the block, the chunk is read from the tape into memory
  - When writing the drive seeks to the end of the tape (unless the last operation was a write already, in which case we're already at the end) (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/mtio/tape.go#L57-L72)
  - After that, the current real tape block position is requested, stored in the index, and the offset is written to the tape (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/backend/ta L119)

- Evaluation

  - `tapisk` is a unique usecase that shows the versitility of the approach chosen and how flexible it is
  - E.g. the chunking system didn't have to be reimplemented (like with STFS) - we could just use the managed mount API directly without changes
  - This essentially makes the tape a chunked, reusable `ReadWriterAt`, in the same way as the directory backend
  - By using a RPC backend, a remote tape can be accessed in the same way, making it possible to map e.g. a remote library's tape robot to your local memory
  - Tape backends can also be used to access large amounts of linear data (terrabytes) from a tape as though it were in memory
  - Makes it possible to access even large datasets or backups in a way that is comfortable and known to developers, compared to other tooling like `tar`
  - Can actually be a real usecase to replace LTFS
  - If a file system (e.g. EXT4) is written to the tape, the tape can be mounted as a filesystem, too
  - LTFS is a kernel module filesystem for tape drives that makes them a file system the same way as STFS
  - But is its own filesystem, while tapisk allows using any existing and tested filesystem on top of the generic block device
  - Doesn't support the caching, making it hard to use for memory mapping, too
  - Tapisk again shows how minimal the overhead of r3map is: While LTFS is 10s of thousands of SLOC, tapisk achieves a similar feature set in just under 350 SLOC

6.11.3 Improving File System Synchronization Solutions

- Existing Solutions
    - Another potential usecase is using r3map to create a mountable remote filesystem with unique properties
    - Currently there are two choices on how these can be implemented
    - Google Drive, Nextcloud etc. listen to file changes on a folder and synchronize files when they change, similarly to the file synchronization approach to memory synchronization
    - The big drawback is that everything needs to be stored locally
    - If a lot of data is stored (e.g. terabytes), the locally available files would need to be manually selected
    - There is no way to dynamically download files this way as they are required
    - It is however very efficient, since the filesystem is completely transparent to the user (writes are being synced back asynchronously)
    - It also supports an offline usecase easily
    - The other option is to use a FUSE, e.g. `s3-fuse`
    - This means that files can be fetched on demand
    - But comes with a heavy performance penalty
    - Writes are usually done (esp. for `s3-fuse`) when they are communicated to the FUSE by the kernel, which makes writes very slow
    - Offline usage is also hard/usually impossible
    - Also usually not a fully featured system (e.g. not `inotify` support, missing permissions etc.)
    - This leaves the choice between two imperfect systems, all with their own downsides
- Hybrid Approach
    - Using r3map can combine both approaches by both not having to download all files in advance and being able to write back changes asynchronously
    - It is also a fully-featured filesystem
    - Files can also be downloaded preemptively for offline access, just like with e.g. the Google Drive approach
    - This can be achieved by using the managed mount API
    - The block device is formatted using any valid filesystem (e.g. EXT4) and then mounted
    - If files should be downloaded in the background, the amount of pull workers can be set to anything >`0`
    - Reads to blocks/files that haven't been read yet can be resolved from the remote backend, giving a FUSE-like experience

  - Since read block are then written to the local one, making subsequent reads almost as fast as native disk reads (unlike in FUSE/like in the Nextcloud approach)
  - Writes are done to the local backend and can then be synced up in periodic intervals like with the Nextcloud approach, making them much faster than a FUSE, too
  - Similarly to how the mount API is used, the migration API could then be used to move the file system between two hosts in a highly efficient way and without a long downtime
  - This essentially creates a system that bridges the gap between FUSE and file synchronization, once again showing how the memory synchronization tooling can be used to solve essentially the synchronization of any state, including disks

6.11.4 Universal Database, Media and Asset Streaming

- Streaming access to remote databases
  - Another usecase is accessing a remote database locally
  - While using a database backend is one option of storing chunks, an actual database can also be stored in a mount as well
  - Particularly interesting for in-memory or on-disk databases like SQLite
  - Instead of having to download the entire SQLite database before using it, it can simply be mounted, and accessed as it is being used
  - This allows very efficient network access, as only very rarely the entire database is needed
  - Since reads are cached with the managed mount API, only the first read should potentially have a performance impact
  - Similarly so writes to the database will be more or less the same throughput as to the local disk, since changes are written back asynchronously
  - If the full database should eventually be accessible locally, the background pullers can be used
  - If the location of e.g. indexes is known, then the pull heuristic can be specified to fetch these first to speed up initial queries
  - SQLite itself doesn't have to support a special interface or changes for this to work, since it can simply use the mount's block device as it's backend, which shows how universally the concept is usable
- Making arbitrary file formats streamable
  - This concept doesn't just apply to SQLite however
  - Other uses could for example be to make formats that are usually not streamable due to technical limitations
  - One such format is famously MP4

  - Usually, if someone downloads a MP4, they can't start playing it before they have finished downloading
  - This is because MP4s store metadata at the end of the file (graphic with metadata)
  - The reason for storing this at the end is that generating the metadata requires encoding the video first
  - Usually, if the file is downloaded from start to finish, then this metadata wouldn't be accessible, and simply inverting the download order wouldn't work since the stream would still be stored in different offsets of the file
  - While there are ways of changing this (for example by storing the metadata in the beginning after the entire MP4 has been transcoded), this is not possible for already existing files
  - With r3map however, the pull heuristic function can be used to immediately pre-fetch the metadata, and the individual chunks of the MP4 file are then being fetched in the background or as they are being accessed
  - This allows making a format such as MP4 streamable before it has been fully downloaded yet, or making almost any format streamable no matter the codec without any changes required to the video/audio player or the format

- Streaming app and game assets
  - Another use case of this in-place streaming approach could be in gaming
  - Usually, all assets of a game are currently being downloaded before the game is playable
  - For many new AAA games, this can be hundreds of gigabytes
  - Many of these assets aren't required to start the game however
  - Only some of them (e.g. the launcher, UI libraries, first level etc.) are
  - In theory, one could design a game engine in such a way that these things are only fetched from a remote as they are required
  - This would however require changing the way new games work, and would not work for existing games
  - Playing games of e.g. a network drive or FUSE is also unrealistic due to performance limitations
  - Using the managed mount API, the performance limitations of such an approach can be reduced, without changes to the game
  - By using the background pull system, reads from chunks that have already been pulled are almost as fast as native disk reads
  - By analyzing the data access pattern of an existing game, a pull heuristic function can be created which will then preemptively pull the game assets that are required first, keeping latency spikes as low as possible
  - Using the chunk availability hooks, the game could also be paused for a loading screen un-

    til a certain local availability percentage is reached to prevent latency spikes from missing assets, while also still allowing for faster startup times
  - The same concept could also be applied to launching any application
  - For many languages, completely loading a binary or script into memory isn't required for it to start executing
  - Without significant changes to most interpreters and language VMs however, loading a file from a file system is the only available interface for loading programs
  - With the managed mount API, this interface can be re-used to add streaming execution support to any intepreter or VM, simply by using the managed mount either as a filesystem to stream multiple binaries/scripts/libraries from, or by using the block device as a file or `mmap`ing it directly to fetch it

6.11.5 Universal App State Synchronization and Migration

- Modelling state
  - Synchronization of app state is hard
  - Even for hand-off scenarios a custom protocol is built most of the times
  - It is possible to use a database sometimes (e.g. Firebase) to synchronize things
  - But that can't sync all data structures and requires using specific APIs to sync things
  - What if you could mount and/or migrate any ressource?
  - Usually these structures are marshalled, sent over a network, received on a second system, unmarshalled, and then are done being synced
  - Requires a complex sync protocol, and when to sync, when to pull etc. is inefficient and usually happens over a third party (e.g. a database)
  - Data structures can almost always be represented by am `[]byte`
  - If the data structures are allocated from on a `[]byte` from the block device, we can use the managed mount/direct mount/migration APIs to send/receive them or mount them
- Persisting state
  - This makes a few interesting usecases possible
  - For example, one could imagine a TODO app
  - By default, the app mounts the TODO list as a byte slice from a remote server
  - Since authentication is pluggable and e.g. a database backend with a prefix for the user can be used, this could scaly fairly well
  - Using the pre-emptive background pulls, when the user connects, they could stream the byte slice in from the remote server as the app is accessing it, and also pre-emptively pull the rest

  - If the TODO list is modified by changing it in memory, these changes are automatically being written to the local backend
  - The asynchronous push algorithm can then take these changes and sync them up to the remote backend when required
  - If the local backend is persistent, such a setup can even survive network outages

- Migrating app state between hosts

  - More interesting possibly than this smart mount, where the remote struct is completely transparent to the user, could be the migration usecase
  - If a user has the TODO app running on a phone, but wants to continue writing on their desktop, they can migrate the app's state with r3map
  - In this usecase, the migration API could be used
  - The pre-copy phase could be automatically detected if the phone comes physically close to the computer
  - Since this would be a LAN scenario, the migration would be very fast and allow for interesting hand-off usecases
  - The migration API also provides hooks and the protocol that would ensure that the app would be closed on the phone before it would be resumed on the desktop, making state corruption during the migration much harder than with a third-party that changes are synced (where the remote state may not be up-to-date yet as it is resumed)
  - Migrating an app's TODO list or any internal data structure is one option

- Migrating virtual machines

  - Locking is not handled by this, only one user is supported
  - The in-memory representation would also need to be universal
  - This is not the case with Go and for different CPU architectures
  - Using a toolkit like Apache Arrow can provide a universal format for this memory, but many other things are possible if a layer of indirection is added
  - But if e.g. Go is compiled to Wasm, and the Wasm VM's linear memory is pointed to the NBD device, it is
  - This would also allow storing in the entire app state in the Wasm remote, with no changes to the app itself
  - As mentioned before, a Wasm VM's memory could also be migrated this way, allowing for any Wasm app to be transfered between hosts
  - Similarly so, the Wasm app's binary, a mounted WASI filesystem etc. could all be synchronized this way, too
  - Thanks to the preemptive pull mechanism, restoring e.g. a Wasm VM outside a migration would also be a very fast operation

  - r3map can also be used for actual VM migration, with any hypervisor that supports mapping a VM's memory to a block device
  - This could also be other VMs (like e.g. the JVM), a browser's JS VM, or - if the processor architectures etc. match - an entire process's memory space
  - Using the mount API, it would also be possible to bring the concept of VM snapshots to almost any app
  - Essentially, its VM memory migration and snapshots, but for any kind of state
  - Its particularly interesting because it can do so without any specific requirements for the data structures of the state other than them being ultimately a `[]byte`, which by definition every state is (def. a process level or VM level etc.)

7 Summary

- Due to the flexible nature of r3map, it is possible to bring the semantics of VM live migration and snapshots to anything that has state
- Looking back at all synchronization options and comparing ease of implementation, CPU load and network traffic between them
- Summary of the different approaches, and how the new solutions presented might make it finally possible to use memory as the universal access format

8 Conclusion

- Answer to the research question (Could memory be the universal way to access and migrate state? - Yes!)
- What would be possible if memory became the universal way to access state (short recap of the ideas for use cases)
- Further research recommendations (e.g. using `ublk` instead of NBD)

9 References

9.1 Structure

- Abstract
- Tables
  - Content

  - Graphics
  - Abbrevations

- Introduction: Research question/goal
- Technology: Current state of research and introductions to related topics (like a reference book)
- Planning: Ideas on how to solve the research question/goal
- Implementation: Description of how the ideas were implemented
- Results: Which results each of the methods yielded (i.e. benchmarks)
- Discussion: How the individual results can be interpreted and what use cases there are for the methods
- Summary: How the discussion can be interpreted as a whole
- Conclusion: Answer to the research question and future outlook, future research questions

Citations

[1] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: Design and implementation*, 3rd ed. Upper Saddle River, NJ 07458: Pearson Education, Inc. Pearson Prentice Hall, 2006.

[2] M. Kerrisk, *The linux programming interface: A linux and UNIX system programming handbook*. No Starch Press, 2010.

[3] D. DeVault, "A hare code generator for finding ioctl numbers." https://drewdevault.com/2022/05/14/generating-ioctls.html, 2022.

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

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

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

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

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

[9] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX® network programming volume 1, third edition: The sockets networking API*. Addison Wesley, 2003, p. 1024.

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

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

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

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

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

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

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

[17] A. S. Tanenbaum, *Computer networks*. Pearson Education, Inc. Publishing as Prentice Hall PTR, 2003.

[18] “Transmission Control Protocol.” RFC 793; J. Postel, Sep. 1981. doi: 10.17487/RFC0793.

[19] “User Datagram Protocol.” RFC 768; J. Postel, Aug. 1980. doi: 10.17487/RFC0768.

[20] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3.” RFC 8446, Aug. 2018. doi: 10.17487/RFC8446.

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

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

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

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

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

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

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

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

[29] A. A. A. Donovan and B. W. Kernighan, *The go programming language*. Addison-Wesley Professional, 2015.

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

[31] L. Labs, “fRPC: A modern RPC framework designed for high performance and stability.” https://frpc.io/getting-started/overview, 2022.

[32] S. Vij, A. Sørlie, F. Pojtinger, and J. Sun, “Frisbee-go: Bring-your-own protocol messaging framework designed for performance and stability.” 2022.Available: https://github.com/loopholelabs/frisbee-go

[33] S. Shivansh Vij, “Polyglot: A high-performance serialization framework used for encoding and decoding arbitrary data structures across languages.” https://github.com/loopholelabs/polyglot, 2022.

[34] Redis Ltd, “Introduction to redis.” https://redis.io/docs/about/, 2023.

[35] Redis Ltd, “Redis pub/sub.” https://redis.io/docs/interact/pubsub/, 2023.

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

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

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

[39] Inc. ScyllaDB, “ScyllaDB ring architecture - overview.” https://opensource.docs.scylladb.com/stable/architecture/ringarchitecture/index.html, 2023.

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

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

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