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

TODO: Add subtitle

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Introduction

Introduction

TODO: Add introduction

Technology

The Linux Kernel

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

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

Linux Kernel Modules

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

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

Kernel modules interact with the kernel via APIs (Application Programming Interfaces). Despite their utility, since they run in kernel space, modules do carry a potential risk. If not written with careful attention to detail,

UNIX Signals and Handlers

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

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

To customize how a process should react upon receiving a specific signal, handlers can be utilized. Handlers dictate the course of action a process

Principle of Locality

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

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

Spatial locality, on the other hand, refers to the use of data elements that

Memory Hierarchy

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

TODO: Add graphic of the memory hierarchy

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

Following registers in the hierarchy is cache memory, typically divided into

Memory Management in Linux

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

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

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

User space is the memory segment where applications and certain drivers

Swap Space

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

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

The Linux kernel employs a Least Recently Used (LRU) algorithm to

Page Faults

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

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

To minimize the occurrence of page faults, memory management algorithms such as the afore-mentioned Least Recently Used (LRU) and the more straightforward clock algorithm are often employed. These algorithms effectively manage the order and priority of memory pages,

mmap

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

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

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

inotify

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

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

Linux Kernel Caching

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

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

Linux also caches file system metadata in specialized structures known as the dentry and inode caches. This metadata encompasses varied information such as file names, attributes, and locations. The key benefit

TCP, UDP and QUIC

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

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

UDP is a connectionless protocol that does not make the same guarantees about the reliability or ordered delivery of data packets [17]. This lends

Delta Synchronization

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

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

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

File Systems In Userspace (FUSE)

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

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

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

```
static int example_getattr(const char *path, struct stat *stb
```

Network Block Device (NBD)

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

TODO: Add sequence diagram of the NBD protocol

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

After receiving this, the server sends the size of the export and other

Virtual Machine Live Migration

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

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

Pre-Copy

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

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

Following this, the VM is suspended at the source, enabling the synchronization of the remaining chunks of data to the destination

Post-Copy

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

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

The main advantage of post-copy migration centers around the fact that it eliminates the necessity of re-transmitting chunks of "dirty" or changed data before hitting the maximum tolerable downtime. This process can

Workload Analysis

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

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

When migration is proposed, the system verifies whether it is in an optimal cycle for migration. If it is, the migration proceeds; if not, the migration is postponed until the next cycle. The proposed process employs a Bayesian

Streams and Pipelines

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

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

Pipelines comprise a series of data processing stages, wherein the output

gRPC

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

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

Redis

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

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

S3 and Minio

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

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

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

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

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

Despite these robust capabilities, Cassandra does come with certain

Planning

Pull-Based Synchronization With userfaultfd

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

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

With userfaultfd, we first register the memory region that we want to

Push-Based Synchronization With mmap and Hashing

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

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

In order to actually detect the changes to the underlying file, an obvious

Push-Pull Synchronization with FUSE

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

While implementing such a custom file system in the kernel is possible, it is a complex task that requires writing a custom kernel module, using a supported language by the kernel (mostly C or a limited subset of Rust), and in general having significant knowledge of kernel internals.

Mounts with NBD

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

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

From an initial overview, the biggest benefit of mmaping such a block device instead of a file on a custom file system is the reduced complexity. For the narrow usecase of memory synchronization, not all of the features

Overview

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

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

While most wire security issues with the protocol can be worked around by simply using TLS, the big issue of it's latency sensitivity remains. Usually, individual blocks would only be fetched as they are being accessed, resulting in a ready latency per block that is at least the RTT. In order to work around this issue, instead of directly connecting a NBD

Chunking

And additional issue that was mentioned before that this approach can approve upon is better chunking support. While it is possible to specify the NBD protocol's chunk size by configuring the NBD client and server, this is limited to only 4KB in the case of Linux's implementation. If the RTT between the backend and the NBD server however is large, it might be preferable to use a much larger chunk size; this used to not be possible by using NBD directly, but thanks to this layer of indirection it can be implemented.

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

Background Pull and Push

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

And example of this can be data that consists of one or multiple headers followed by raw data. If this structure is known, rather than fetching everything linearly in the background, the headers can be fetched first in order to allow for i.e. metadata to be displayed before the rest of the data has been fetched. Similarly so, if a file system is being synchronized, and

Overview

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

To improve on this the pull-based migration API, the migration process is split into two distinct phases. This is required due the constraint mentioned earlier; the mount API does not allow for safe concurrent access of a remote resource by two readers or writers at the same time. This poses a signficant problem for the migration scenario, as the app that is writing to the source device would need to be suspended before the

Migration Protocol and Critical Phases

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

Implementation

Registration and Handlers

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

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

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

uffdioAPI := constants.NewUffdioAPI(
```

userfaultfd Backends

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

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


    return n, nil
}
```

Caching Restrictions

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

Linux caches reads to the backing file similarly to how it does if read etc. are being used, meaning that only the first page fault would be responded to by reading from disk; this means that any future changes to the backing file would not be represented in the mmaped region, similarly to how userfaultfd handles it. The same applies to writes, meaning that in the same way that files need to be synced in order for them to be flushed to disk, mmaped regions need to be msynced in order to flush changes to

Detecting File Changes

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

To speed up the process of hashing, instead of hashing the entire file, we

Synchronization Protocol

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

TODO: Add sequence diagram for the protocol

Multiplexer Hub

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

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

File Advertisement

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

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

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

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

File Receiver

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

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

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

  go func() {
```

File Transmission

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

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

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

// Comparing the hashes
blocksToSend := []int64{}
```

Hash Calculation

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

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

lock := semaphore.NewWeighted(parallel)

// ...

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

File Reception

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

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

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

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

FUSE Implementation in Go

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

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

```
type Fs interface {
    Create(name string) (File, error)
    Mkdir(name string, perm os.FileMode) error
```

Overview

Due to a lack of existing, lean and maintained NBD libraries for Go, a custom pure Go NBD library was implemented. Most NBD libraries also only provide a server and no the client component, but both are needed for the NBD-based migration approach to work. By not having to rely on CGo or a pre-existing NBD library like nbdkit, this custom library can also skip a significant amount of the overhead that is typically associated with C interoperability, particularly in the context of concurrency in Go with CGo [44].

Server

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

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

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

Client

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

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

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

Client Lifecycle

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

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

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

// Subscribing to events for the device name
```

Optimizing Access to the Block Device

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

Combining the NBD Client and Server to a Mount

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

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

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

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

Stages

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

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

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

Chunking

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

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

buf := make([]byte, a.chunkSize)
// Repeat until all chunks that need to be fetched have been
for remaining > 0 {
    // Calculating the chunk and offset within the chunk
```

Background Pull

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

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

// ...

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

Background Push

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

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

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

Pipeline

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

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

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

Concurrent Device Initialization

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

Device Lifecycles

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

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

WAN Optimization

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

By using the pull-only, unidirectional API to emulate the pre-copy setup, the destination can simply keep track of which chunks it still needs to pull itself, meaning that if there is a network outage, it can just resume pulling or decide to restart the pre-copy process. Unlike the pre-copy system used for the file synchronization/hashing approach, this also means that destination hosts don't need to subscribe to a central multiplexing hub.

Overview

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

Seeder

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

```
type SeederRemote struct {
    ReadAt func(context context.Context, length int, off int6
    Size   func(context context.Context) (int64, error)
    Track  func(context context.Context) error
    Sync   func(context context.Context) ([]int64, error)
    Close  func(context context.Context) error
}
```

Unlike the remote backend, the seeder also exposes a mount through the familiar path, file or slice APIs, meaning that even as the migration is in progress, the underlying ressource can still be accessed by the application on the source host. This fixes the architectural constraint of the mount

Leecher

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

TODO: Add graphic of pipeline design

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

```
// For `ReadAt/WriteAt`: Waits for finalization, then calls a.lock.L.Lock()
```