Efficient Synchronization of Linux Memory Regions over a Network

A Comparative Study and Implementation

Felicitas Pojtinger

2023-08-04

Hochschule der Medien Stuttgart

Introduction

Introduction

In today's technological landscape, numerous methods exist for accessing remote resources, such as via databases or custom APIs. The same applies to resource synchronization, which is typically addressed on a case-by-case basis via methods such as third-party databases, file synchronization services, or bespoke synchronization protocols. The migration of resources is also frequently a challenge, relying on APIs better suited for persistence, like storing the resource in a remote database. These solutions are generally custom-built for specific applications, despite the typical internal representation of the resource as a memory region or file.

What if, instead of applying application-specific protocols and abstractions for accessing, synchronizing, and migrating resources, theses processes could be universally managed by directly operating on the memory region? While systems for interacting with remote memory exist, they primarily serve niche purposes, such as virtual machine live migration,

Technology

User Space and Kernel Space

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

User space on the other hand is the portion of system memory where user applications execute. Applications can't directly access hardware or kernel memory; instead they use APIs to access these them[1]. This API is provided in the form of syscalls, which serve as a bridge between user and kernel space. Well-known syscalls are open(), read(), write(), close() and ioctl(). While most syscalls have a specific purpose, ioctl serves as a more generic, universal one based on file descriptors and a data struct. Using it, it is possible to implement device-specific actions that can't be

The Linux Kernel

The Linux kernel was released by Linus Torvalds in 1991. Developed primarily in the C language, it has recently seen the addition of Rust as an approved option for further expansion and development, esp. for drivers[3]. The kernel powers millions of devices across the globe, including servers, desktop computers, mobile phones, and embedded devices. As a kernel, it serves as an intermediary between hardware and applications. It is engineered for compatibility with a wide array of architectures, such as ARM, x86, RISC-V, and others.

The kernel does not function as a standalone operating system in itself; rather, this role is fulfilled by distributions, which build upon the Linux kernel to create fully-fledged operating systems. Distributions supplement the kernel with additional 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 extensible, but not a microkernel. Despite this monolithic nature, it allows for the integration of kernel modules[4]. These modules are small pieces of kernel-level code that can be dynamically incorporated into the kernel, presenting the advantage of extending kernel functionality without necessitating system reboots. This functionality also helps to keep the kernel size both manageable and maintainable, thereby promoting efficiency. Kernel modules are developed using the C or Rust programming languages, like the kernel itself, ensuring compatibility and consistent performance.

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

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[6]. 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

UNIX Sockets

Sockets allow processes within the same host system to communicate with each other. Unlike UNIX signals, much like TCP sockets, they can be used for IPC by allowing not only to submit additional data for an event, and are particularly popular on Linux.

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

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[9].

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[10]. 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 kernel, serving as a critical buffer between applications and physical memory; it can be considered one of the fundamental purposes of a kernel itself. This system helps maintain system stability and provides security guarantees, such as ensuring that only a specific process can access its allocated memory.

Within the context of Linux, memory management is divided into two afore-mentioned major segments of kernel space and user space. The kernel memory module is responsible for managing kernel space. Slab allocation is a technique employed in managing this segment; the technique groups objects of the same size into caches, enhancing memory allocation speed and reducing fragmentation of memory[13]. User space is the memory segment where applications and certain drivers store their memory in Linux[14]. User space memory management involves a paging system, offering each application its unique private virtual address space.

Swap Space

Swap space refers to a designated portion of the secondary storage utilized as virtual memory in a computer system[14]. This feature plays a crucial role in systems that run multiple applications simultaneously; when memory resources are strained, swap space comes into play, relocating inactive parts of the RAM to secondary storage. This action frees up space in primary memory for other processes, enabling smoother operation and preventing a potential system crash.

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

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

Page Faults

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

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

To minimize the occurrence of page faults, memory management algorithms such as the 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 used in applications like databases.

One standout feature of mmap is its ability to create what is essentially a direct memory mapping between a file and a region of memory[17]. This connection means that read operations performed on the mapped memory region directly correspond to reading the file and vice versa, enhancing efficiency as the amount of expensive context switches (compared to i.e. the read or write system calls) can be reduced.

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

inotify

inotify is an event-driven notification system of the Linux kernel, designed to monitor the file system for different events, such as modifications and accesses, among others[18]. 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 of this is that it expedites the resolution of path names and file attributes, such as

RTT, LAN and WAN

Round-trip time (RTT) represents the time data takes to travel from a source to a destination and back. It provides a valuable insight into application latency, and can varying according to many factors such as network type, system load and physical distance. Local area networks (LAN) are geographically small networks that are typically characterised by having a low RTT, resulting in a low latency due to the small distance (typically no more than across an office or data center) that data needs to travel[19]. As a result of their small geographical size and isolation, perimeter security is often applied to such networks, meaning that the LAN is viewed as a trusted network that doesn't necessarily require authentication or encryption between internal systems, resulting in a potentially smaller overhead.

Wide area networks (WAN) on the other hand typically span a large geographical area, with the internet being an example that operates on a planetary scale. Due to the physical distance between source and

TCP, UDP, TLS and QUIC

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

TCP has long been the reliable backbone for internet communication due to its connection-oriented nature [20]. 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 [21]. 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[25].

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[30].

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 moving a virtual machine, its state, and its connected devices from one host to another, with the objective to minimize disrupted service by minimizing downtime during the processes. Algorithms that implement this 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[33].

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[34] 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[35].

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

Go

Go is a statically typed, compiled open-source programming language released by Google in 2009. It is typically known for its simplicity, and was developed to address the unsuitability of many traditional languages for modern distrbuted systems development. Thanks to input from many people affiliated with UNIX, such as Rob Pike and Ken Thomposon, as well as good support for concurrency, Go is particulary popular for the development of cloud services and other types of network programming. The headline feature of Go is “Goroutines”, a lightweight feature that allows for concurrent function execution that is similar to threads, but more scalable to support millions of Goroutines per program. Synchronization between different Goroutines is provided by using channels, which are type- and concurrency-safe conduits for data[37].

gRPC and Protocol Buffers

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

One of the notable features of the gRPC is its usage of HTTP/2 as the transport protocol. This allows it to exploit features of HTTP/2 such as header compression, which minimizes bandwidth usage, and request multiplexing, enabling multiple requests to be sent concurrently over a single connection. In addition to HTTP/2, gRPC utilizes Protocol Buffers (Protobuf), more specifically proto3, as the Interface Definition Language (IDL) and wire format. Protobuf is a compact, high-performance, and language-neutral mechanism for data serialization. This makes it preferable over the more dynamic, but more verbose and slower JSON format often used in REST APIs.

fRPC and Polyglot

fRPC is an open-source RPC framework released by Loophole labs in 2022. It is proto3-compatible, meaning that it can be used as a drop-in replacement for gRPC, promising better performance characteristics. A unique feature is it's ability to stop the RPC system to rerieve an underlying connection, which makes it possible to re-use connections for different purposes[39]. Internally, it uses Frisbee as it's messaging framework to implement the request-response semantics[40], and Polyglot, a high-performance serialization framework, as it's Protobuf equivalent. Polyglot achieves a similar goal as Protobuf, which is to encode data structures in a platform-independent way, but does so with less legacy code and a simpler wire format. It is also language-independent, with implementations for Go, Rust and TypeScript[41].

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[42].

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.

It offers a variety of storage classes for to different needs, i.e. for whether the requirement is for frequent data access, infrequent data retrieval, or long-term archival. This ensures that it can meet a wide array of demands through the same API. S3 also comes equipped with comprehensive security features, including authentication and authorization mechanisms. Communication with S3 is done through a HTTP API. Users and applications can interact with the stored data - including files and folders - via this API.[44].

Cassandra and ScylllaDB

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

Cassandra's consistency model is tunable according to needs, ranging from eventual to strong consistency. It distinguishes itself by not employing master nodes due to its usage of a peer-to-peer protocol and a distributed hash ring design. These design choices eradicate the bottleneck and failure risks associated with master nodes[46].

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 resource into memory, here we want to handle the page faults using a user space program instead. Traditionally, this was possible by registering a signal handler for the SIGSEGV handler, and then responding to fault from the program. This however is a fairly complicated and inefficient process. Instead, we can now use the userfaultfd system to register a page fault handler directly without having to go through a signal first.

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

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 resource's memory region, similar to the prior approaches, but differs from mounts with NBD in a few significant ways.

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

While most wire security issues with the protocol can be worked around by simply using TLS, the big issue of 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 resources 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[48] 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 4 KB. By having this simple backend interface, and thus only requiring read-only access, it is possible to implement the migration backend in many different ways. A simple backend can for example return a pattern to the memory region:

```
func (a abcReader) ReadAt(p []byte, off int64) (n int, err 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 simplified. It supports synchronizing multiple files at the same time by using the file names as IDs, and also supports a central forwarding hub instead of requiring peer-to-peer connectivity between all hosts, which also reduces network traffic since this central hub could also be used to forward one stream to all other peers instead of having to send it multiple times. The protocol defines three actors: The multiplexer, file advertiser and file receiver.

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[49]. 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[50]:

```
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 [53].

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[55]. 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 resource 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()
```

Pluggable Encryption, Authentication and Transport

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

This makes adding guarantees such as encryption as simple as choosing the best solution depending on the network conditions. For low-latency, trusted networks, a protocol like the SCSI RDMA protocol (SRP) can be

Concurrent Backends

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

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

```
func (b *FileBackend) ReadAt(p []byte, off int64) (n int, er
    b.lock.RLock()
    defer b.lock.RUnlock()

    n, err = b.file.ReadAt(p, off)
    //
```

Overview

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

Key-Value Stores with Redis

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

```
func (b *RedisBackend) ReadAt(p []byte, off int64) (n int, e
    // Retrieve a key corresponding to the chunk from Redis
    val, err := b.client.Get(b.ctx, strconv.FormatInt(off, 1
    // If  a key does not exist, treat it as an empty chunk
    if err == redis.Nil {
        return len(p), nil
    }
    // ...
}
```

Object Stores with S3

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

```
func (b *S3Backend) ReadAt(p []byte, off int64) (n int, err
    // Receiving a chunk using Minio's S3 client
```

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

```
func (b *CassandraBackend) ReadAt(p []byte, off int64) (n int
    // Executing a select query for a specific chunk, then s
    var val []byte
    if err := b.session.Query(`select␣data␣from␣`+b.table+`␣
```

Another aspect that plays an important role in performance for real-life deployments is the choice of RPC framework and transport protocol. As mentioned before, both the mount and migration APIs are transport-independent, and as a result almost any RPC framework can be used. A RPC framework developed as part of r3map is Dudireka[56]. As such, it was designed specifically with the hybrid pre-and post-copy scenario in mind. To optimize for this, it has support for concurrent RPCs, which allows for efficient background pulls as multiple chunks can be pulled at the same time.

The framework also allows for defining functions on both the client and the server, which makes it possible to initiate pre-copy migratons and transfer chnks from the source host to the destination without having the latter be dialable; while making the destination host available by dialing it is possible in trusted LAN deployments, NATs and security concerns make it harder in WAN deployment. As part of this bi-directional support

Usage

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

```
// Server
type local struct { counter int64 }

func (s *local) Increment(ctx context.Context, delta int64)
    return atomic.AddInt64(&s.counter, delta), nil
}
```

Protocol

The protocol used for dudirekta is simple and based on JSONL, a for exchanging newline-delimited JSON data[58]; a function call, i.e. to Println looks like this:

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

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

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

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

RPC Providers

If an RPC (such as ReadAt in the case of the mount API) is called, a method with the provided RPC's name is looked up on the provided implementation struct and if it is found, the provided argument's types are validated against those of the implementation by unmarshalling them into their native natives:

```
// Unmarshalling the function name and args from the protoco
var functionName string
json.Unmarshal(res[2], &functionName)

var functionArgs []json.RawMessage
json.Unmarshal(res[3], &functionArgs)

// Looking up the field on the local struct
function := reflect.
    ValueOf(r.local.wrappee).
```

RPC Calls

As mentioned earlier, on the caller's side, a placeholder struct representing the callee's available RPCs is provided to the registry. Once the registry is linked to a connection, the placeholder struct's methods are iterated over and the signatures are validated for compatibility with Dudirekta's limitations. They are then implemented using and set using reflection:

```
// Iterating over the fields of the struct
for i := 0; i < remote.NumField(); i++ {
    functionField := remote.Type().Field(i)
    functionType := functionField.Type
    // ...

    // Validating that the requirements for the return value
    if functionType.NumOut() <= 0 || functionType.NumOut() >
        return ErrInvalidReturn
```

Adapter for Mounts and Migrations

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

```
func NewRPCBackend(
    ctx context.Context, // Global context to be used for Du
    remote *services.BackendRemote, // Dudirekta placeholder
) *RPCBackend {
    return &RPCBackend{ctx, remote}
}

func (b *RPCBackend) ReadAt(p []byte, off int64) (n int, err
    // Calling the RPC
```

Connection Pooling with gRPC

While the dudirekta RPC implementation serves as a good reference implementation of how RPC backends work, it has issues with scalability (see figure 16). This is mostly the case because of it's JSONL-based wire format, which, while simply and easy to analyize, is quite slow to marshal and unmarshal. The bi-directional RPCs do also come at a cost, since they prevent an effective use of connection pooling; since a client dialing the server multiple times would mean that server could not reference multiple client connections as one composite client, it would not be able to differentiate two client connections from two separate clients. While implementing a future pooling mechanism based on a client ID is possible in the future, bi-directional RPCs can also be completely avoided entirely by implementing the pull- instead of push-based pre-copy solution described earlier where the destination host keeps track of the pull progress, effectively making unary RPC support the only requirement for a RPC framework.

Optimizing Throughput with fRPC

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

TODO: Add graphic from https://frpc.io/performance/grpc-benchmarks

fRPC[39], a drop-in replacement for gRPC, can improve upon this by switching out the serialization layer with the faster Polyglot[41] library and a custom transport layer. It also uses the proto3 DSL and the same code generation framework as gRPC, which makes it easy to switch to by simply re-generating the code from the DSL. The implementation of the fRPC adapter functions in a very similar way as the gRPC adapter:

```
type SeederFrpc struct {
    svc *Seeder // Dudirekta RPC implementation
```

Results

Testing Environment

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

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

To make the results reproducible, the benchmark scripts and notebooks to plot the related visualizations can be found in the accompanying repository[59], and multiple runs have been conducted for each benchmark to ensure consistency.

Latency

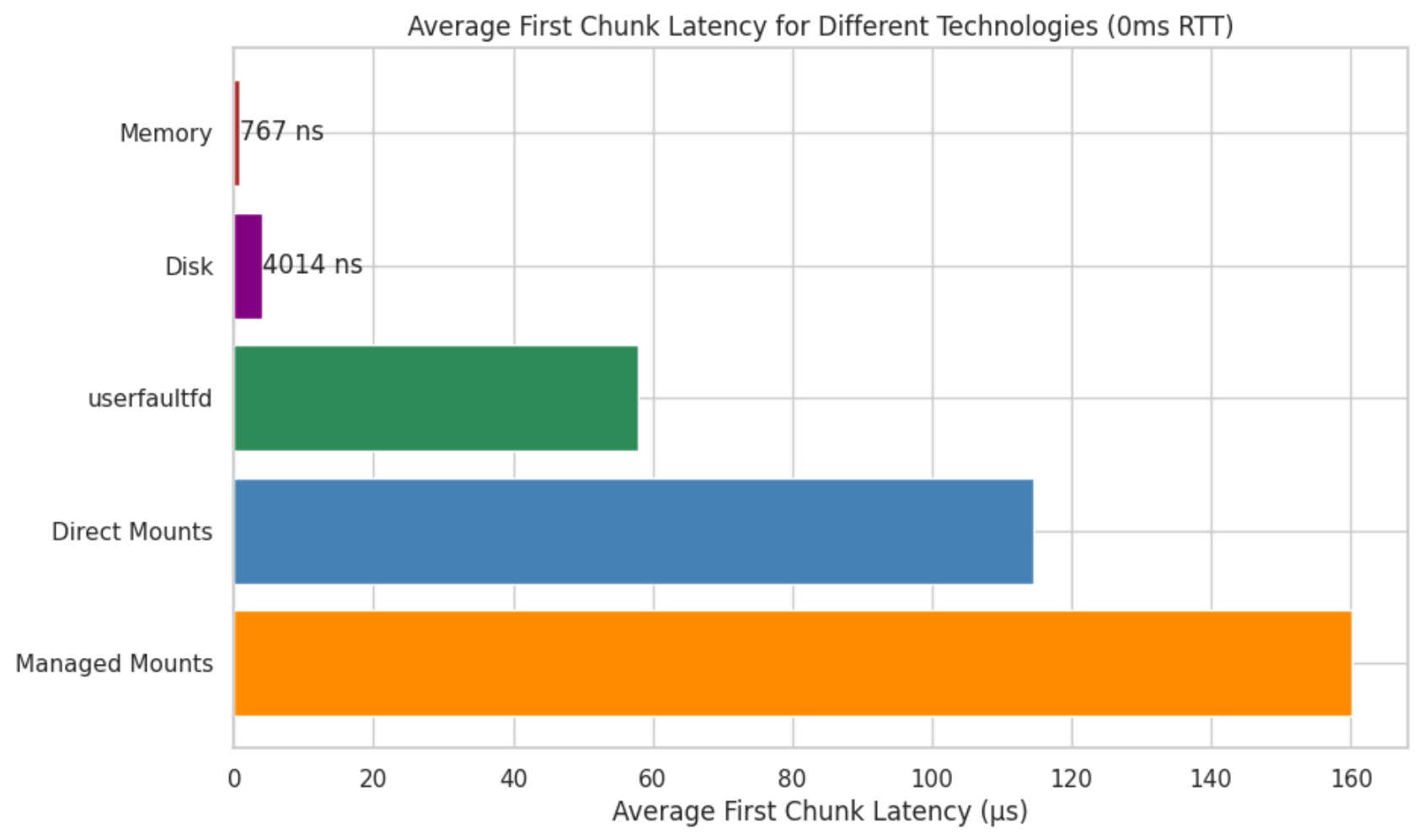


Figure 1: Average first chunk latency for different direct memory access, disk, userfaultfd, direct mounts and managed mounts (0ms RTT)

Read Throughput

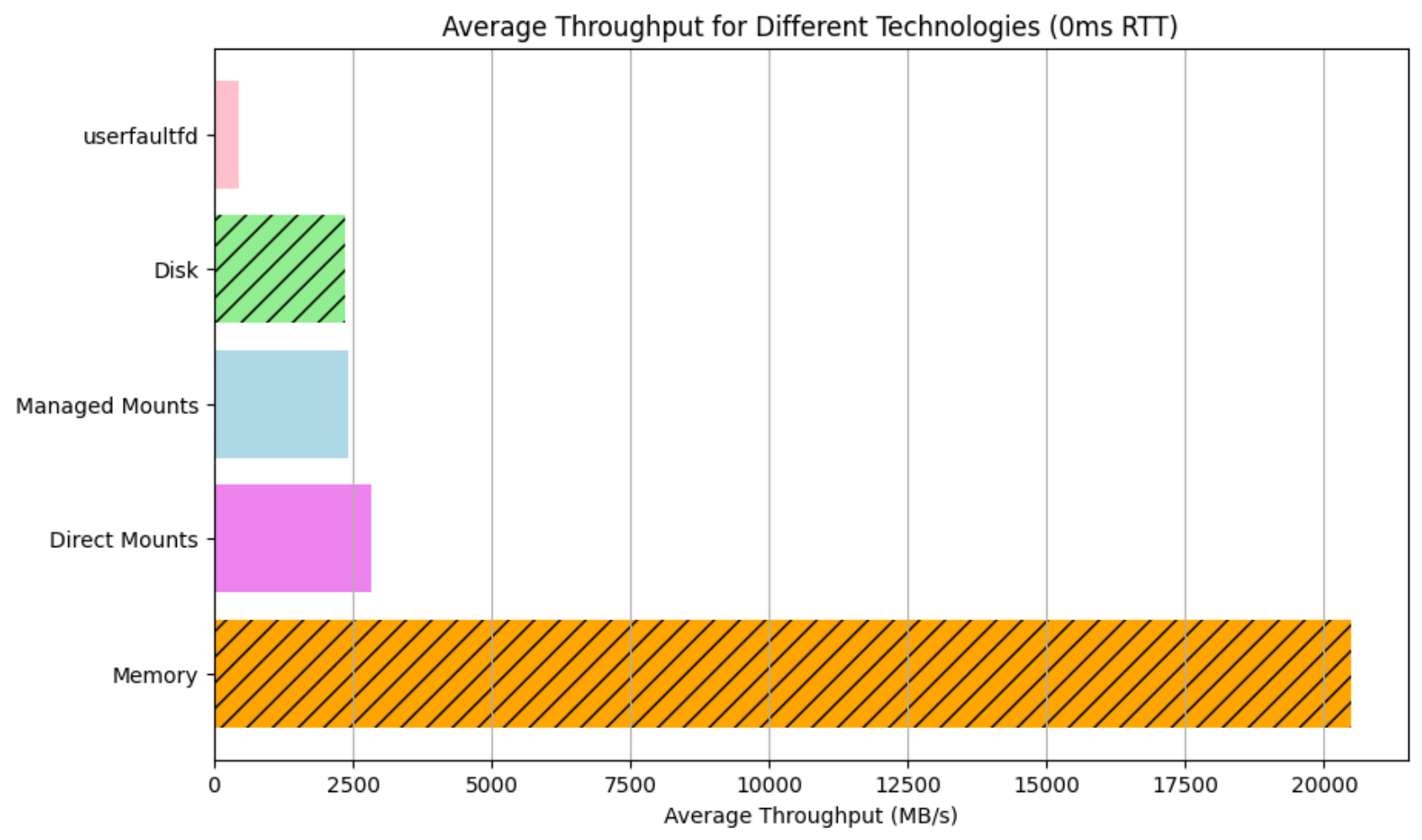


Figure 5: Average throughput for memory, disk, userfaultfd, direct mounts and managed mounts (0ms RTT)

Write Throughput

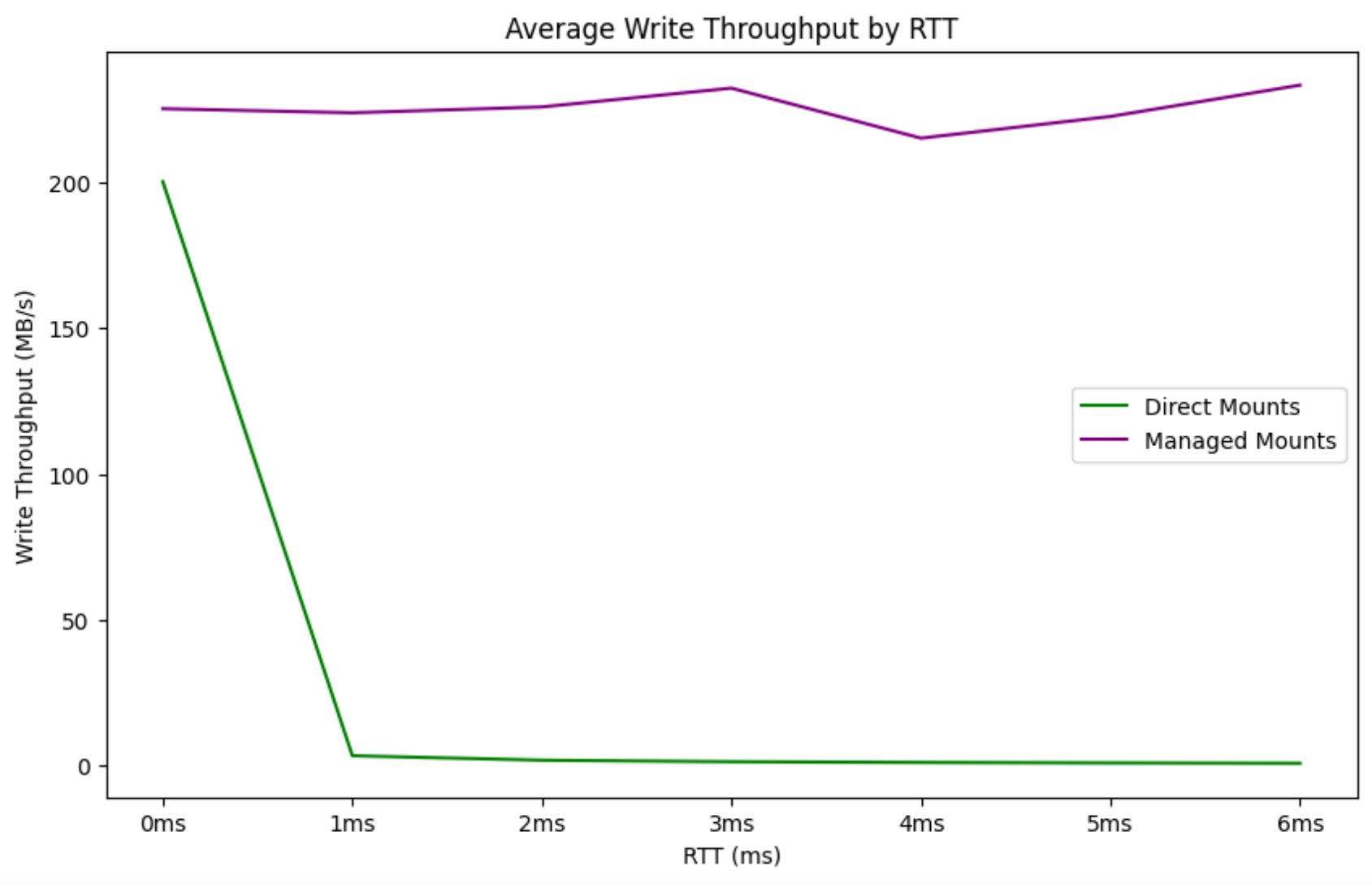


Figure 10: Average write throughput for direct and managed mounts by RTT

While for a migration or mount scenario read throughput is a critical

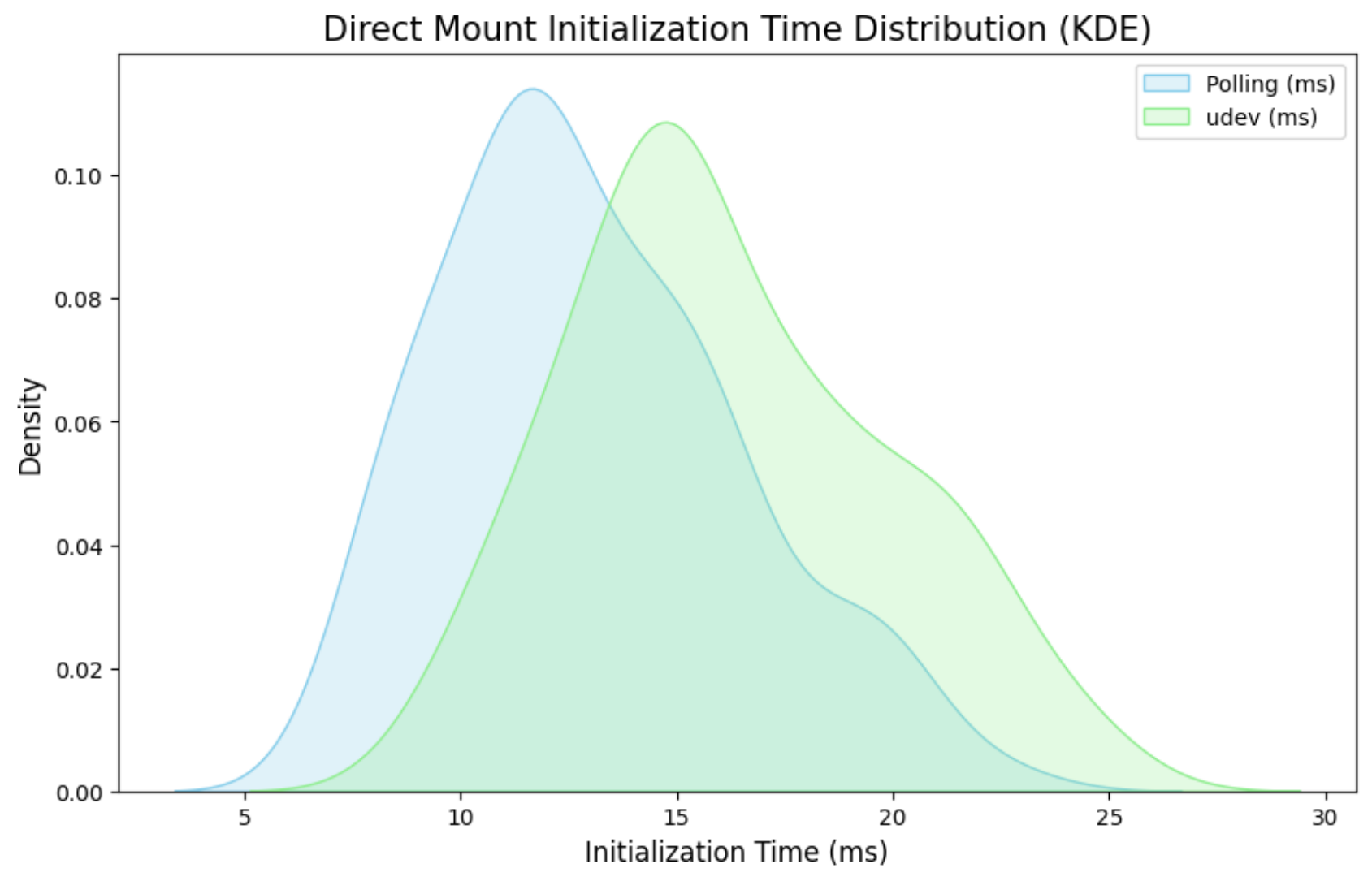


Figure 11: Kernel density estimation for the distribution of direct mount initialization time with polling vs. udev.

Chunking

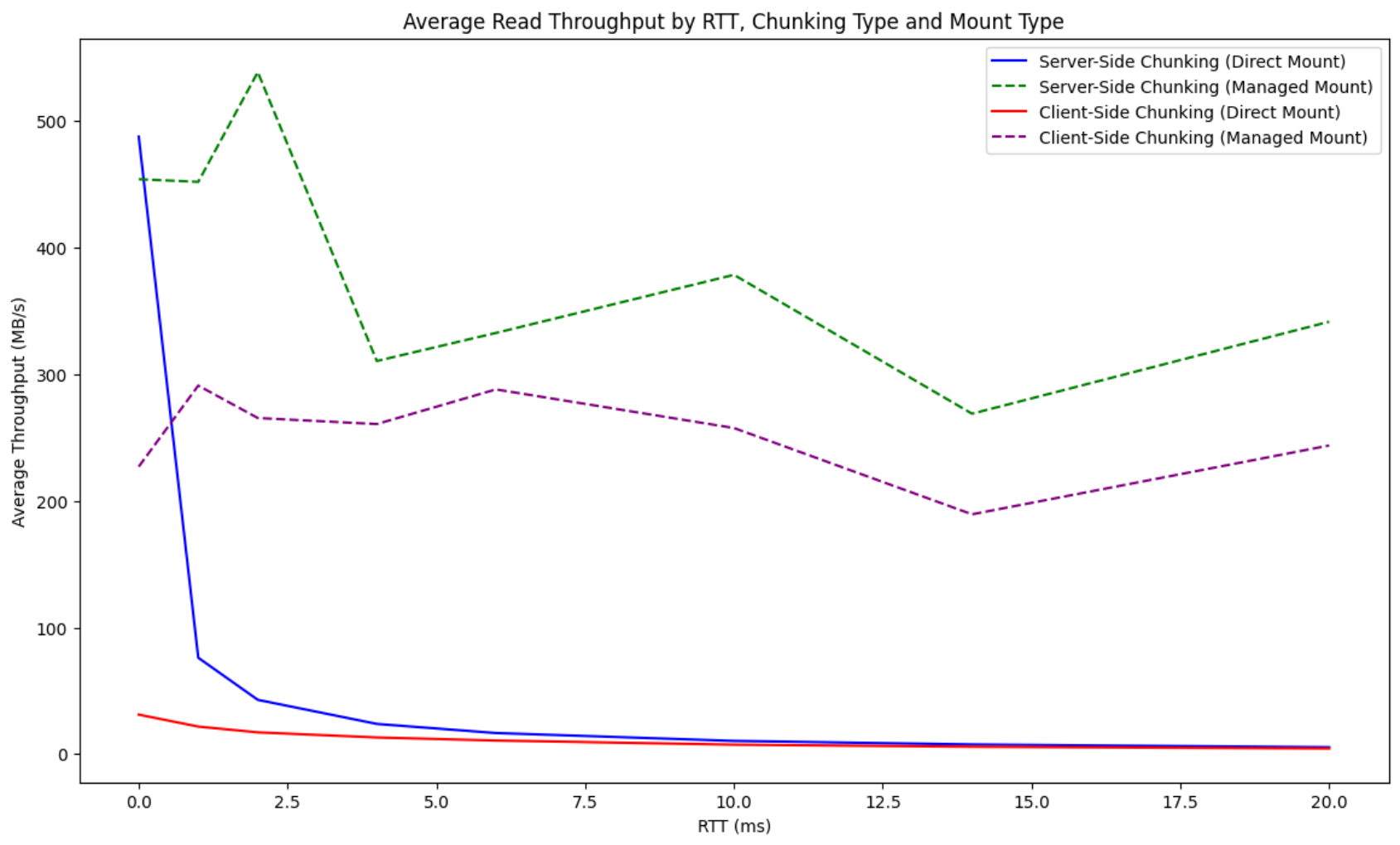


Figure 13: Average read throughput for server-side and client-side chunking, direct mounts and managed mounts by RTT

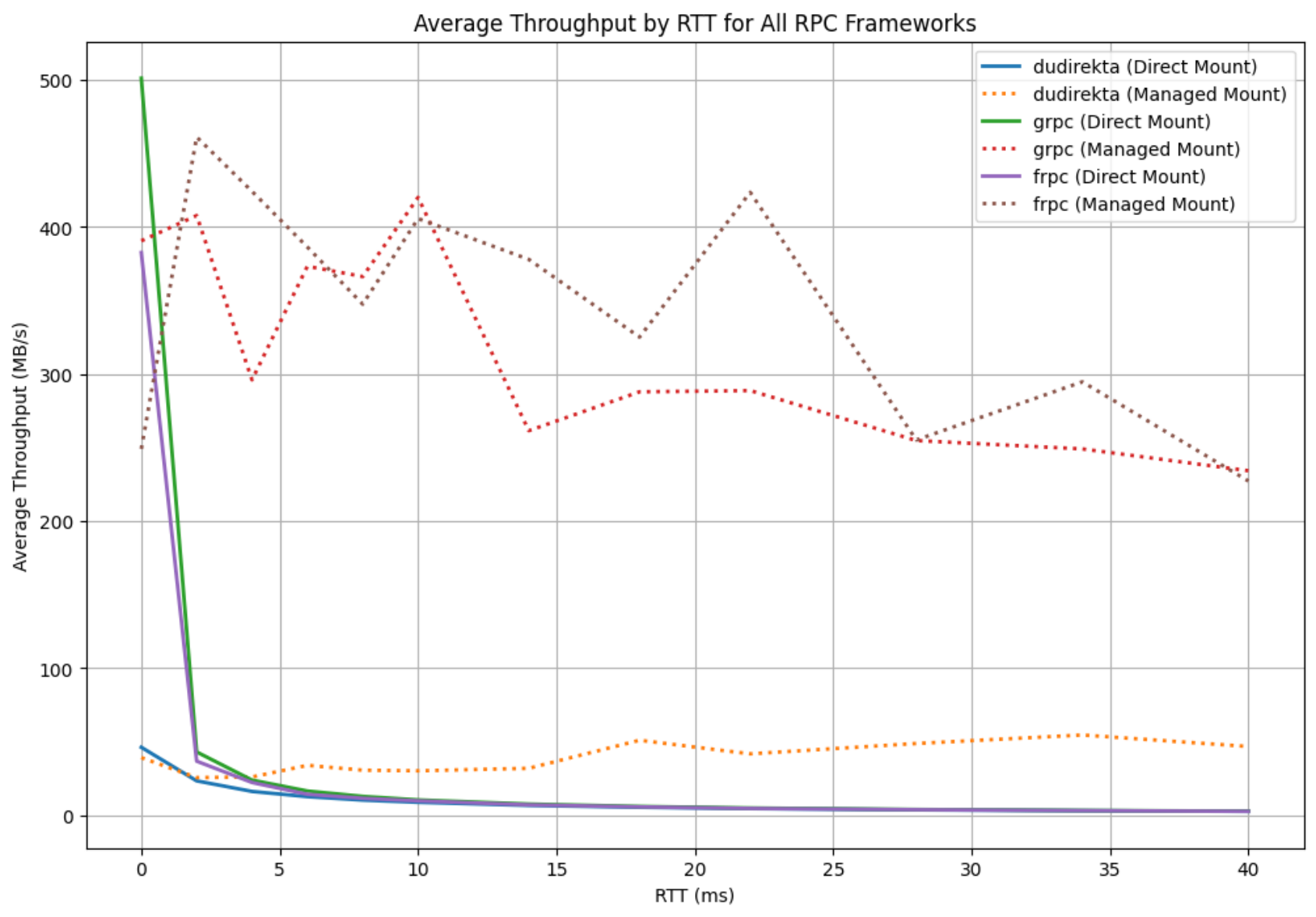
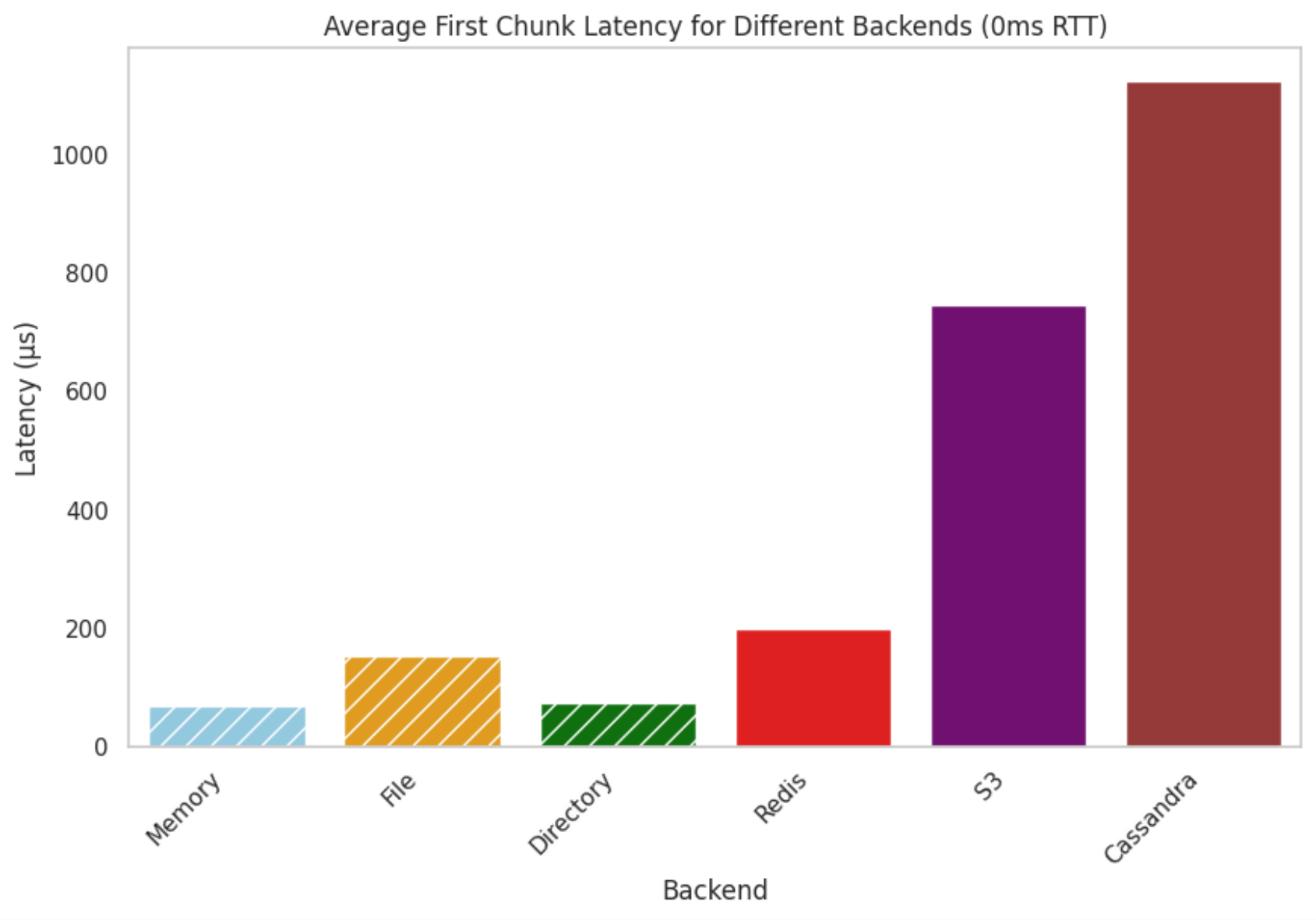


Figure 16: Average throughput by RTT for Dudirekta, gRPC and fRPC frameworks for direct and managed mounts.

Latency


Average First Chunk Latency for Different Backends (0ms RTT)
Latency (µs)
0
200
400
600
800
1000
Memory
File
Directory
Redis
S3
Cassandra
Backend

Throughput

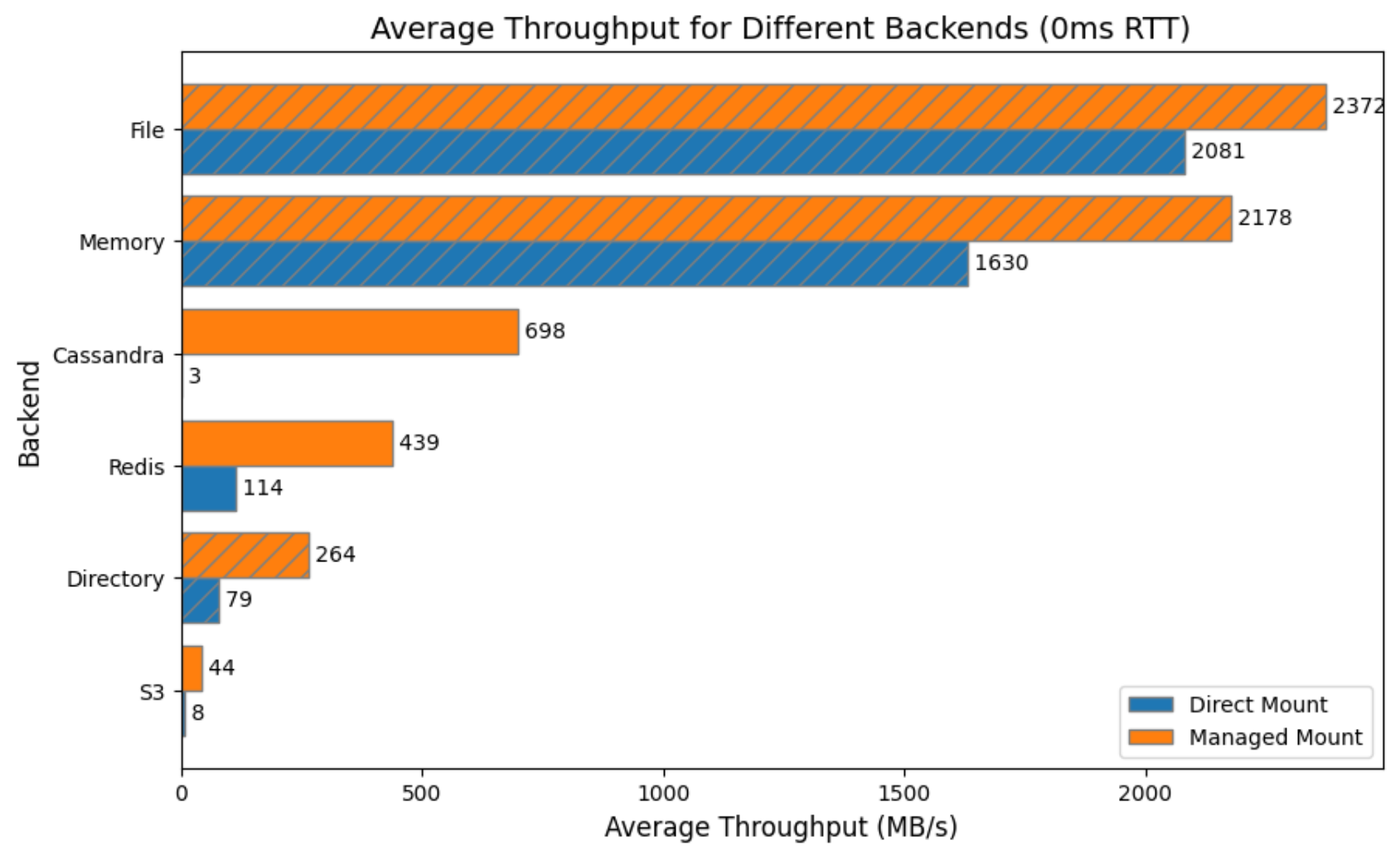


Figure 21: Average throughput for memory, file, directory, Redis, S3 and ScylllaDB backends for direct and managed mounts (0ms RTT)

Discussion

Userfaults

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

Due to these limitations, userfaultfd is essentially limited to read-only mounts of remote resources, not synchronization. While it could be a viable solution for post-copy migration, it also prevents pullling chunks from being pulled before they have been accesed without adding an additional layer of indirection. The userfaultfd API socket is also

File-Based Synchronization

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

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

FUSE

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

Direct Mounts

Direct mounts have a high spread when it comes to first chunk latency at 0 ms RTT(see figure 2), but are more predictable when it comes to their throughput (see figure 7). Similarly to the drawbacks of userfaultfd, it's first chunk latency grows linearly as the RTT increases (see figure 3), due to the lack of pre-emptive pulls. Despite this, it has the highest throughput at 0 ms RTT, even higher than managed mounts (see figure 5) due to it having less expensive internal I/O operations because of the lack of this pull system. While compared to userfaultfd, it's read throughput doesn't drop as rapidly as RTT increases (see figure 8). It's write speed is heavily influenced by RTT (see figure 10) since writes need to be written to the remote as they happen, as there is no background push system either. These characteristics make direct mounts a good access method to choose if the internal overhead of using managed mounts is higher than the overhead caused for direct mounts by the RTT, which can be the case in LAN or other very low-latency networks.

Managed Mounts

Managed mounts have a internal overhead to due to the duplicate I/O operations required for background pull, resulting in a worse throughput for low RTT scenarios compared to direct mounts (esp. for 0 ms RTT; see figure 5), as well as higher first chunk latencies (see figure 1). As soon as the RTT reaches levels more typical for a WAN environment however, this overhead becomes negligible compared to the benefits gained over the other access methods thanks to the background push and pull systems (see figure 3 and 8).

Tuning the background workers to the specific environment can substantially increase a managed mount's performance (see figure 4 and 9), since data can be fetched in parallel. The pull priority function can allow for even more optimized pulls, and pre-emptive pulls can significantly reduce initial chunk latency since data can be pulled asynchronously before the device is even available. Higher worker counts to increase the amount of data that is being pulled preemptively (see

Chunking

In general, server-side chunking should almost always be the preferred technology due to the much better throughput compared to client-side chunking (see figure 13). For direct mounts, due to their linear/synchronous access pattern, the throughput is low for both server- and client-side chunking as RTT increases, but even with linear access server-side chunking still outperforms the alternative (see figure 14). For managed mounts, client-side chunking can still halve the throughput of a mount compared to server-side chunking (see figure 15). If the data chunks are smaller than the NBD block size, it reduces the number of chunks that can be fetched if the number of workers remains the same. This isn't the case with server-side chunking because it doesn't need an extra worker on the client side for each additional chunk that needs to be fetched. This allows the background pull system to fetch more, thus increasing throughput.

Out of the frameworks tested, Dudirekta consistently has lower throughput than the alternatives (see figure 16). It performs better for managed mounts than direct mounts thanks to it's support for concurrent RPCs and is less sensitive to RTT compared to gRPC and fRPC for managed mounts, but even for the latter, it's throughput is much lower compared to both alternatives (see figure 18) due to it's lack of connecting pooling. Despite these drawbacks however, Dudirekta remains an interesting option for prototyping due to the decreased friction in developer overhead, bi-directional PC support and transport layer independence.

gRPC offers considerably faster throughput compared to Dudirekta for both managed and direct mounts (see figure 16). It has support for connection pooling, giving it a significant performance benefit over Dudirekta for managed mounts (see figure 18), do to it being able to more efficiently pull chunks in the background concurrently. It also has good throughput for 0 ms RTT scenarios, and is essentially an industry standard.

Redis is the network-capable backend with the the lowest amount of initial chunk latency at a 0 ms RTT (see figure 19). When used for direct mounts, it has a lower throughput compared to managed mounts (see figure 21), showing good support for concurrent chunk access; it also has the highest thoughput for direct mounts by a significant margin (see figure 22) due to it's optimized wire protocol and fast key lookups. It also has good throughput in managed mounts due to these optimizations (see figure 24), making it a good choice for ephemeral data like caches, where quick access times are necessary or the direct mount API provides benefits, i.e. in LAN deployments.

ScylllaDB has the highest throughput for 0 ms RTT deployments for managed mounts, showing a very good concurrent access performance (see figure 24). It does however fall short when it comes to usage in direct mounts, where the performance is worse than any other backend (see figure 22), showing the database's high read latency overhead for looking

Limitations

While the mount APIs are functional for most usecases, there are performance and usability issues due it being implemented in Go. Go is a garbage collected language, and if the garbage collector is active, it has to stop goroutines. If the mmap API is used to access a managed mount or a direct mount, it is possible that the garbage collector tries to manage an object with a reference to the exposed slice, or tries to release memory as data is being copied from the NBD device. If the garbage collector then tries to access the slice, it can stop the goroutine providing the slice in the form of the NBD server, causing the deadlock. One workaround for this is to lock the mmaped region into memory, but this will also cause all chunks to be fetched from the remote into memory, which leads to a high Open() latency; as a result, the current workaround for this is to simply start the NBD server in a separate process, so as to prevent the garbage collector from stopping the NBD server and trying to access the slice at the same time. Another workaround for this issue could be to instead use

Using Mounts for Remote Swap with ram-dl

ram-dl[61] is an experimental tech demo built to demonstrate how the mount API can be used. It uses the fRPC mount backend to expand local system memory, enabling a variety of usecases such as mounting a remote system's RAM locally or easily inspecting a remote system's memory contents.

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

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

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

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

tapisk[62] is a tool that exposes a tape drive as a block device. While seemingly unrelated to memory synchronization, it does serve as an interesting usecase due to the similarities to STFS (mentioned earlier in the FUSE section), which exposed a tape drive as a file system, and serves as an interesting example for how even seemingly incompatible backends for can be used to store and synchronize memory.

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

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

Since chunking works differently for tapes than for block devices, and tapes are append-only devices where overwriting a section prior to the end would result in all following data being overwritten, too, an index must be used to simulate the offsets of the block device locations to their physical location on the tape, which the bbolt database is used for. In order to make non-aligned reads and writes to the tape possible, the

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

Being able to map a tape into memory without having to read the entire contents first can have a variety of usecases. Tapes can store a large amount of data, in the case of LTO-9, up to 18TB on a single tape[63]; being able to access such a large amount of data directly in memory, instead of having to work with tooling like tar, can significantly improve developer

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

Hybrid Approach

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

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

Streaming Access to Remote Databases

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

Since reads are cached using the local backend with the managed mount API, only the first read should potentially have a performance impact (if it has not been pulled first by the background pull system); similarly so, since writes are written to the local backend first, and then

Making Arbitrary File Formats Streamable

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

TODO: Add graphic with MP4 metadata

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

Streaming App and Game Assets

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

By using the managed mount API to stream in the assets, the overhead of

Modelling State

Synchronization of app state is a fairly complex problem, and even for simple scenarios, a custom protocol is built for simple apps. While it possible to real-time databases like Firebase to synchronize some application state, it and similar solutions to it are usually limited in which data structures they can store and require specific APIs to synchronize them. Usually, even for a simple migration of state between two hosts, synchronization requires state to be manually marshalled, sent over a network, received on a destination host, and unmarshalled. This requires a complex synchronization protocol, and decisions such as when to synchronize state and when to start pulling from the remote need to be made manually, resulting in a database on a third host being used even for simple migrations from one host to another. Almost all of these data structures can ultimately be represented by a byte array; by allocating them from a slice mmaped by r3map, we can use the managed mount, direct mount or migration APIs to implement a universal way of both

Mounting State

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

Migrating State

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

It is important to note that there are a few limitations with synchronizing and migrating an application's internal stateful data structures this way; locking is not handled by r3map and would need to be done using a higher-level protocol; moreover, this assumes that the in-memory representation of the data structure is consistent across all, something which is not necessarily the case with programming languages such as Go with multiple processor architectures being involved. While projects such as Apache Arrow[64] allow for application state to represented in a language- and CPU architecture-independent way, this comes with some of the same restrictions on which state can be synchronized as with other solutions such as Firebase.

In order to keep the possibility of migrating arbitrary state, but also allow for cross-architecture compatibility, VMs can be used. Keeping with the TODO app example, if the resulting app is compiled to Wasm, instead of having to allocate all memory that is to be synchronized from the

Summary

Summary

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

When it comes to access methods, userfaultfd is an interesting API that is idiomatic to both Linux in as a first-party solution and Go due to it's fairly low implementation overhead. This approach however falls short when it comes to throughput, especially when used in WAN, where other options can provide better performance. The delta synchronization method for mmaped files provides a simple way of memory synchronization for specific scenarios, but does have a very significant I/O and compute overhead due to it's polling and hashing requirements that make it unsuitable for most applications; similarly so, FUSE provides an extensive API for implementing a complete file system in user space, but has

Conclusion

Conclusion

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

ram-dl demonstrates how minimal the implementation overhead is by implementing a system to share and mount a remote system's memory in under 300 source lines of code, while tapisk shows that the APIs can be used to efficiently map almost any resource, including a linear-access tape drive, to the concepts provided. Aside from these examples, the solution also makes many entirely new use cases that were previously thought of as extraodinarly hard to achieve possible, such as file synchronization that can combine the benefits of NBD with those of existing cloud storage

Bibliography

Bibliography

[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, pp. 27–29.

[2]
D. DeVault, “A hare code generator for finding ioctl numbers.” 2022. Accessed: Jul. 28, 2023. [Online]. Available: https://drewdevault.com/2022/05/14/generating-ioctls.html

[3]
Kernel Development Community, “Quick start.” 2023. Accessed: Jul. 19, 2023. [Online]. Available: https://www.kernel.org/doc/html/next/rust/quick-start.html

[4]
R. Love, *Linux kernel development*, 3rd ed. Pearson Education, Inc., 2010, pp. 8, 343–344.