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

A user-friendly approach to application-agnostic state synchronization

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Contents

1 Unsorted Research Questions **2**

2 Structure **2**

3 Content **2**

3.1 IMRAD (Introduction, Methods, Results and Discussion) Structure 37

1 Unsorted Research Questions

2 Structure

- Abstract
- Introduction
- Theory
- Implementation
- Results
- Conclusion

3 Content

- Introduction
  - Memory management in Linux
  - Memory as the universal storage API
  - What would be possible if memory would be the universal way to access resources?
  - Why efficient memory synchronization is the missing key component
  - High-level use cases for memory synchronization in the industry today
- Pull-Based Memory Synchronization with `userfaultfd`
  - Page faults occur when a process tries to access a memory region that has not yet been mapped into a process' address space
  - By listening to these page faults, we know when a process wants to access a specific piece of memory
  - We can use this to then pull the chunk of memory from a remote, map it to the address on which the page fault occured, thus only fetching data when it is required
  - Usually, handling page faults is something that the kernel does
  - In our case, we want to handle page faults in userspace
  - In the past, this used to be possible by handling the `SIGSEGV` signal in the process
  - In our case however, we can use a recent system called `userfaultfd` to do this in a more elegant way (available since kernel 4.11)
  - `userfaultfd` allows handling these page faults in userspace
  - Implementing this in Go was quite tricky, and it involves using `unsafe`
  - We can use the `syscall` and `unix` packages to interact with `ioctl` etc.

- We can use the `ioctl` syscall to get a file descriptor to the `userfaultfd` API, and then register the API to handle any faults on the region (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/register.go#L15)
- The region that should be handled can be allocated with e.g. `mmap`
- Once we have the file descriptor for the `userfaultfd` API, we need to transfer this file descriptor to a process that should respond with the chunks of memory to be put into the faulting address
- Passing file descriptors between processes is possible by using a UNIX socket (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/transfer/unix.go)
- Once we have received the socket we need to register the handler for the API to use
- If the handler receives an address that has faulted, it responds with the `UFFDIO_COPY` `ioctl` and a pointer to the chunk of memory that should be used on the file descriptor (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/handler.go
- A big benefit of using `userfaultfd` and the pull method is that we are able to simplify the backend of the entire system down to a `io.ReaderAt` (code snippet from https://pkg.go.dev/io#ReaderAt)
- That means we can use almost any `io.ReaderAt` as a backend for a `userfaultfd-go` registered object
- We know that access will always be aligned to 4 KB chunks/the system page size, so we can assume a chunk size on the server based on that
- For the first example, we can return a random pattern in the backend (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-abc/main.go) - this shows a great way of exposing truly arbitrary information into a byte slice without having to pre-compute everything or changing the application
- Since a file is a valid `io.ReaderAt`, we can also use a file as the backend directly, creating a system that essentially allows for mounting a (remote) file into memory (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-file/main.go)
- Similarly so, we can use it map a remote object from S3 into memory, and access only the chunks of it that we actually require (which in the case of S3 is achieved with HTTP range requests) (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-s3/main.go)
- As we can see, using `userfaultfd` we are able to map almost any object into memory
- This approach is very clean and has comparatively little overhead, but also has significant architecture-related problems that limit its uses
- The first big problem is only being able to catch page faults - that means we can only ever respond the first time a chunk of memory gets accessed, all future requests will return the memory directly from RAM on the destination host

  - This prevents us from using this approach for remote resources that update over
  - Also prevents us from using it for things that might have concurrent writers/shared resources, since there would be no way of updating the conflicting section
  - Essentially makes this system only usable for a read-only "mount" of a remote resource, not really synchronization
  - Also prevents pulling chunks before they are being accessed without layers of indirection
  - The `userfaultfd` API socket is also synchronous, so each chunk needs to be sent one after the other, meaning that it is very vulnerable to long RTT values
  - Also means that the initial latency will be at minimum the RTT to the remote source, and (without caching) so will be each future request
  - The biggest problem however: All of these drawbacks mean that in real-life usecases, the maximum throughput, even if a local process handles page faults on a modern computer, is ~50MB/s
  - Benchmark: Sensitivity of `userfaultfd` to network latency and throughput
  - In summary, while this approach is interesting and very idiomatic to Go, for most data, esp. larger datasets and in high-latency scenarios/in WAN, we need a better solution

- Push-Based Memory Synchronization with `mmap` and Hashing

  - This approach tries to improve on `userfaultfd` by switching to push-based synchronization method
  - Instead of reacting to page faults, this one a file to track changes to a memory region
  - By synchronizing the file representing the memory region between two systems, we can effectively synchronize the memory region itself
  - In Linux, swap space allows Linux to move chunks of memory to disk or other swap partition if the fast speed of RAM is not needed ("paging out")
  - Similarly to this, Linux can also load missing chunks from a disk
  - This works similarly to how `userfaultfd` handled page faults, except this time it doesn't need to go through user space, which can make it much faster
  - We can do this by using `mmap`, which allows us to map a file into memory
  - By default, `mmap` doesn't write changes from a file back into memory, no matter if the file descriptor passed to it would allow it to or not
  - We can however add the `MAP_SHARED` flag; this tells the kernel to write back changes to the memory region to the corresponding regions of the backing file
  - Linux caches reads to such a backing file, so only the first page fault would be answered by fetching from disk, just like with `userfaultfd`
  - The same applies to writes; similar to how files need to be `sync`ed in order for them to be written to disks, `mmap`ed regions need to be `msync`ed in order to flush changes to the backing file

- In order to synchronize changes to the region between hosts by syncing the underlying file, we need to have the changes actually be represented in the file, which is why `msync` is critical
- For files, you can use `O_DIRECT` to skip this kernel caching if your process already does caching on its own, but this flag is ignored by the `mmap`
- Usually, one would use `inotify` to watch changes to a file
- `inotify` allows applications to register handlers on a file's events, e.g. `WRITE` or `SYNC`. This allows for efficient file synchronization, and is used by many file synchronization tools
- It is also possible to filter only the events that we need to sync the writes, making it the perfect choice for this use case
- For technical reasons however (mostly because the file is represented by a memory region), Linux doesn't fire these events for `mmap`ed files though, so we can't use it
- The next best option are two: Either polling for file attribute changes (e.g. last write), or by continously hashing the file to check if it has changed
- Polling on its own has a lot of downsides, like it adding a guaranteed minimum latency by virtue of having to wait for the next polling cycle
- This negatively impacts a maximum allowed downtime scenario, where the overhead of polling can make or break a system
- Hashing the entire file is also a naturally IO- and CPU-intensive process because the entire file needs to be read at some point
- Still, polling & hashing is probably the only reliable way of detecting changes to a file
- When picking algorithms for this hashing process, the most important metric to consider is the throughput with which it can compute hashes, as well as the change of collisions
- Benchmark: Performance of different Go hashing algorithms for detecting writes
- Instead of hashing the entire file, then syncing the entire file, we can want to really sync only the parts of the file that have changed between two polling iterations
- We can do this by opening up the file multiple times, then hashing individual offsets, and aggregating the chunks that have changed
- If the underlying hashing algorithm is CPU-bound, this also allows for better concurrent processing
- Increases the initial latency/overhead by having to open up multiple file descriptors
- Benchmark: Hashing the chunks individually vs. hashing the entire file
- But this can not only increase the speed of each individual polling tick, it can also drastically decrease the amount of data that needs to be transferred since only the delta needs to be synchronized
- Hashing and/or syncing individual chunks that have changed is a common practice
- The probably most popular tool for file synchronization like this is rsync
- When the delta-transfer algorithm for rsync is active, it computes the difference between

the local and the remote file, and then synchronizes the changes
  - The delta sync algorithm first does file block division
  - The file on the destination is divided into fixed-size blocks
  - For each block in the destination, a weak and fast checksum is calculated
  - The checksums are sent over to the source
  - On the source, the same checksum calculation process is run, and compared against the checksums that were sent over (matching block identification)
  - Once the changed blocks are known, the source sends over the offset of each block and the changed block's data to the destination
  - When a block is received, the destination writes the chunk to the specified offset, reconstructing the file
  - Once one polling interval is done, the process begins again
  - We have implemented a simple TCP-based protocol for this delta synchronization, just like rsync's delta synchronization algorithm (code snippet from https://github.com/loopholelabs/darkmagyk/l orchestrator/main.go#L1337-L1411 etc.)
  - For this protocol specifically, we send the changed file's name as the first message when starting the synchronization, but a simple multiplexing system could easily be implemented by sending a file ID with each message
  - Similarly to `userfaultfd`, this system also has limitations
  - While `userfaultfd` was only able to catch reads, this system is only able to catch writes to the file
  - Essentially this system is write-only, and it is very inefficient to add hosts to the network later on
  - As a result, if there are many possible destinations to migrate state too, a star-based architecture with a central forwarding hub can be used
  - The static topology of this approach can be used to only ever require hashing on one of the destinations and the source instead of all of them
  - This way, we only need to push the changes to one component (the hub), instead of having to push them to each destination on their own
  - The hub simply forwards the messages to all the other destinations
  - Benchmark: Throughput of this custom synchronization protocol vs. rsync (which hashes entire files)

- Push-Pull Memory Synchronization with a FUSE
  - Since the push method requires polling and is very CPU and I/O intensive, and `userfaultfd-go` has too low of a throughput, a better solution is needed
  - What if we could still get the events for the writes and reads without having to use `userfaultfd-go` or hashing?

- We can create a custom file system in Linux and load it as a kernel module
- This file system could then intercept reads/writes to/from the `mmap`ed region, making it possible to respond to them with a custom backend
- But such a system would need to run in the kernel directly, which leads to a lot of potential drawbacks
- While it is possible to write kernel modules with Rust instead of C these days, a lot of problems remain
- Kernel modules aren't portable; they are built for a specific kernel, which makes them hard to distribute to users
- A kernel file system is able to skip having to go through user space and thus save on context switching, but that will also mean that it will run in the kernel address space, making for a poor level of isolation
- Iterating on a kernel module is much harder than iterating on a program running in user space
- If we want the user to be able to provide their own backends from/to which to pull/push, that will still require communication between user space and the kernel
- So while adding this implementation in the kernel would be possible, it would also be very complex
- In order to implement file systems in user space, we can use the FUSE API
- Here, a user space program registers itself with the FUSE kernel module
- This program provides callbacks for the file system operations, e.g. for `open`, `read`, `write` etc.
- When the user performs a file system operation on a mounted FUSE file system, the kernel module will send a request for the operation to the user space program, which can then reply with a response, which the FUSE kernel module will then return to the user
- This makes it much easier to create a file system compared to writing it in the kernel, as it can run in user space
- It is also much safer as no custom kernel module is required and an error in the FUSE or the backend can't crash the entire kernel
- Unlike a file system implemented as a kernel module, this layer of indirection makes the file system portable, since it only needs to communicate with the FUSE module
- It is possible to use even very complex and at first view non-compatible backends as a FUSE file system's backend
- By using a file system abstraction API like `afero.Fs`, we can separate the FUSE implementation from the actual file system structure, making it unit testable and making it possible to add caching in user space (code snippet from https://github.com/pojntfx/stfs/blob/main/pkg/fs/fil
- It is possible to map any `afero.Fs` to a FUSE backend, so it would be possible to switch between different file system backends without having to write FUSE-specific (code snip-

pet from https://github.com/JakWai01/sile-fystem/blob/main/pkg/filesystem/fs.go)
  - For example, STFS used a tape drive as the backend, which is not random access, but instead append-only and linear (https://github.com/pojntfx/stfs/blob/main/pkg/operations/update.go)
  - By using an on-disk index and access optimizations, the resulting file system was still performant enough to be used, and supported almost all features required for the average user
  - FUSE does however also have downsides
  - It operates in user space, which means that it needs to do context switching
  - Some advanced features aren't available for a FUSE
  - The overhead of FUSE (and implementing a completely custom file system) for synchronizing memory is still significant
  - If possible, the optimal solution would be to not expose a full file system to track changes, but rather a single file
  - As a result of this, the significant implementation overhead of such a file system led to it not being chosen

- Pull-Based Memory Synchronization with NBD
  - As hinted at before, a better API would be able to catch reads/writes to a single `mmap`ed file instead of having to implement a complete file system
  - It does however not have to be an actual file, a block device also works
  - In Linux, block devices are (typically storage) devices that support reading/writing fixed chunks (blocks) of data
  - We can `mmap` a block device in the same way that we can `mmap` a file
  - Similarly to how a file system can be implemented in a kernel module, a block device is typically implemented as a kernel module/in kernel space
  - However, the same security, portability and developer experience issues as with the former also apply here
  - Instead of implementing a FUSE to solve this, we can create a NBD (network block device) server that can be used by the kernel NBD module similarly to how the process that connected to the FUSE kernel module functioned
  - The difference between a FUSE and NBD is that a NBD server doesn't provide a file system
  - A NBD server provides the storage device that (typically) hosts a file system, which means that the interface for it is much, much simpler
  - The implementation overhead of a NBD server's backend is much more similar to how `userfaultfd-go` works, rather than a FUSE
  - NBD uses a protocol to communicate between a server (provided by user space) and a client (provided by the NBD kernel module)
  - The protocol can run over WAN, but is really mostly meant for LAN or localhost usage

- It has two phases: Handshake and transmission
- There are two actors in the protocol: One or multiple clients, the server and the virtual concept of an export
- When the client connects to the server, the server sends a greeting message with the server's flags
- The client responds with its own flags and an export name (a single NBD server can expose multiple devices) to use
- The server sends the export's size and other metadata, after which the client acknowledges the received data and the handshake is complete
- After the handshake, the client and server start exchanging commands and replies
- A command can be any of the basic operations needed to access a block device, e.g. read, write or flush
- Depending on the command, it can also contain data (such as the chunk to be written), offsets, lengths and more
- Replies can contain an error, success value or data depending on the reply's type
- NBD is however limited in some respects; the maximum message size is 32 MB, but the maximum block/chunk size supported by the kernel is just 4096 KB, making it a suboptimal protocol to run over WAN, esp. in high latency scenarios
- The protocol also allows for listing exports, making it possible to e.g. list multiple memory regions on a single server
- NBD is an older protocol with multiple different handshake versions and legacy features
- Since the purpose of NBD in this use case is minimal and both the server and the client are typically controlled, it makes sense to only implement the latest recommended versions and the baseline feature set
- The baseline feature set requires no TLS, the latest "fixed newstyle" handshake, the ability to list and choose an export, as well as the read, write and disc(onnect) commands and replies
- As such, the protocol is very simple to implement
- With this simplicity however also come some drawbacks: NBD is less suited for use cases where the backing device behaves very differently from a random-access store device, like for example a tape drive, since it is not possible to work with high-level abstractions such as files or directories
- This is, for the narrow memory synchronization use case, however more of a feature than a bug
- Due to the lack of pre-existing libraries, a new pure Go NBD library was implemented
- This library does not rely on CGo/a pre-existing C library, meaning that a lot of context switching can be skipped
- The backend interface for `go-nbd` is very simple and only requires four methods: `ReadAt`

, `WriteAt`, `Size` and `Sync`
- A good example backend that maps well to a block device is the file backend (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/backend/file.go)
- The key difference here to the way backends were designed in `userfaultfd-go` is that they can also handle writes
- `go-nbd` exposes a `Handle` function to support multiple users without depending on a specific transport layer (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go)
- This means that systems that are peer-to-peer (e.g. WebRTC), and thus don't provide a TCP-style `accept` syscall can still be used easily
- It also allows for easily hosting NBD and other services on the same TCP socket
- The server encodes/decodes messages with the `binary` package (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/server/nbd.go#L73-L76)
- To make it easier to parse, the headers and other structured messages are modeled as Go structs (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/protocol/negotiation.go)
- The handshake is implemented using a simple for loop, which either returns on error or breaks
- The actual transmission phase is done similarly, by reading in a header, switching on the message type and reading/sending the relevant data/reply
- The server is completely in user space, there are no kernel components involved here
- The NBD client however is implemented by using the kernel NBD client
- In order to use it, one needs to find a free NBD device first
- NBD devices are pre-created by the NBD kernel module and more can be specified with the `nbds_max` parameter
- In order to find a free one, we can either specify it directly, or check whether we can find a NBD device with zero size in `sysfs` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg
- Relevant `ioctl` numbers depend on the kernel and are extracted using `CGo` (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/ioctl/negotiation_cgo.go)
- The handshake for the NBD client is negotiated in user space by the Go program
- Simple for loop, basically the same as for the server (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L221-L288)
- After the metadata for the export has been fetched in the handshake, the kernel NBD client is configured using `ioctl`s (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L290-L328)
- The `DO_IT` syscall never returns, meaning that an external system must be used to detect whether the device is actually ready
- Two ways of detecting whether the device is ready: By polling `sysfs` for the size parameter, or by using `udev`

- `udev` manages devices in Linux
- When a device becomes available, the kernel sends a `udev` event, which we can subscribe to and use as a reliable and idiomatic way of waiting for the ready state (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L104C10-L138)
- In reality however, polling `sysfs` directly can be faster than subscribing to the `udev` event, so we give the user the option to switch between both options (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/client/nbd.go#L140-L178)
- When `open`ing the block device that the client has connected to, usually the kernel does provide a caching mechanism and thus requires `sync` to flush changes
- By using `O_DIRECT` however, it is possible to skip the kernel caching layer and write all changes directly to the NBD client/server
- This is particularly useful if both the client and server are on the local system, and if the amount of time spent on `sync`ing should be as small as possible
- It does however require reads and writes on the device node to be aligned to the system's page size, which is possible to implement with a client-side chunking system but does require application-specific code
- NBD is a battle-tested solution for this with fairly good performance, but in the future a more lean implemenation called `ublk` could also be used
- `ublk` uses `io_uring`, which means that it could potentially allow for much faster concurrent access
- It is similar to NBD; it also uses a user space server to provide the block device backend, and a kernel `ublk` driver that creates `/dev/ublkb*` devices
- Unlike as it is the case for the NBD kernel module, which uses a rather slow UNIX or TCP socket to communicate, `ublk` is able to use `io_uring` pass-through commands
- The `io_uring` architecture promises lower latency and better throughput
- Because it is however still experimental and docs are lacking, NBD was chosen
- Another option of implementing a block device is BUSE (block devices in user space)
- BUSE is similar to FUSE in nature, and similarly to it has a kernel and user space server component
- Similarly to `ublk` however, BUSE is experimental
- Client libraries in Go are also experimental, preventing it from being used as easily as NBD
- Similarly so, a CUSE could be implemented (char device in user space)
- CUSE is a very flexible way of defining a char (and thus block) device, but also lacks documentation
- The interface being exposed by CUSE is more complicated than that of e.g. NBD, but allows for interesting features such as custom `ioctl`s (code snippet from https://github.com/pojntfx/webpipe/blob/main/pkg/cuse/device.go#L3-L15)
- The only way of implementing it without too much overhead however is CGo, which comes

with its own overhead
  - It also requires calling Go closures from C code, which is complicated (code snippet from https://github.com/pojntfx/webpipe/blob/main/pkg/cuse/bindings.go#L79-L132)
  - Implementing closures is possible by using the `userdata` parameter in the CUSE C API (code snippet from https://github.com/pojntfx/webpipe/blob/main/pkg/cuse/cuse.c#L20-L22)
  - To fully use it, it needs to first resolve a Go callback in C, and then call it with a pointer to the method's struct in user data, effectively allowing for the use of closures (code snippet from https://github.com/pojntfx/webpipe/blob/main/pkg/cuse/bindings.go#L134-L162)
  - Even with this however, it is hard to implement even a simple backend, and the CGo overhead is a significant drawback (code snippet from https://github.com/pojntfx/webpipe/blob/main/pkg/de
  - The last alternative to NBD devices would be to extend the kernel with a new construct that allows for essentially a virtual file to be `mmap`ed, not a block device
  - This could use a custom protocol that is optimized for this use case instead of a full block device
  - Because of the extensive setup required to implement such a system however, and the possibility of `ublk` providing a performant alternative in the future, going forward with NBD was chosen for now

- Push-Pull Memory Synchronization with Mounts
  - Usually, the NBD server and client don't run on the same system
  - NBD was originally designed to used as a LAN protocol to access a remote hard disk
  - As mentioned before, NBD can run over WAN, but is not designed for this
  - The biggest problem with running NBD over a public network, even if TLS is enabled is latency
  - Individual chunks would only be fetched to the local system as they are being accessed, adding a guaranteed minimum latency of at least the RTT
  - Instead of directly connecting a client to a remote server, we add a layer of indirection, called a `Mount` that consists of both a client and a server, both running on the local system
  - We then connect the server to the backend with an API that is better suited for WAN usage
  - This also makes it possible to implement smart pull/push strategies instead of simply directly writing to/from the network ("managed mounts")
  - The server and client are connected by creating a connected UNIX socket pair (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_direct.go#L59-L62)
  - By building on this basic direct mount, we can add a file (code snippet from https://github.com/pojntfx/r3m and slice (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/slice_direct.go) mount API, which allows for easy usage and integration with `sync`/`msync` respectively

- Using the `mmap`/slice approach has a few benefits
- First, it makes it possible to use the byte slice directly as though it were a byte slice allocated by `make`, except its transparently mapped to the (remote) backend
- `mmap`/the byte slices also swaps out the syscall-based file interface with a random access one, which allows for faster concurrent reads from the underlying backend
- Alternatively, it would also be possible to format the server's backend or the block device using standard file system tools
- When the device then becomes ready, it can be mounted to a directory on the system
- This way it is possible to `mmap` one or multiple files on the mounted file system instead of `mmap`ing the block device directly
- This allows for handling multiple remote regions using a single server, and thus saving on initialization time and overhead
- Using a proper file system however does introduce both storage overhead and complexity, which is why e.g. the FUSE approach was not chosen
- The simplest form of the mount API is the direct mount API
- This API simply swaps out NBD for a transport-independent RPC framework, but does not do additional optimizations
- It has two simple actors: A client and a server, with only the server providing methods to be called (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/backend.go#L14-L19)
- The protocol as such is stateless, as there is only a simple remote read/write interface (add state machine and sequence diagram here)
- One additional layer that needs to be implemented however is proper chunking support
- While we can specify a chunk size for the NBD client in the form of a block size, we can only go up to 4 KB chunks
- For scenarios where the RTT between the backend and server is large, it might make sense to use a much larger chunk size for the actual networked transfers
- Many backends also have constraints that prevent them from functioning without a specific chunk size or aligned offsets, such as using e.g. tape drives, which require setting a block size and work best when these chunks are multiple MBs instead of KBs
- Even if there are no constraints on chunking on the backend side (e.g. when a file is used as the backend), it might make sense to limit the maximum supported message size between the client and server to prevent DoS attacks by forcing the backend to allocate large chunks of memory to satisfy requests, which requires a chunking system to work
- In order to implement the chunking system, we can use a abstraction layer that allows us to create a pipeline of readers/writers - the `ReadWriterAt`, combining an `io.ReaderAt` and a `io.WriterAt`
- This way, we can forward the `Size` and `Sync` syscalls directly to the underlying back-

end, but wrap a backend's `ReadAt` and `WriteAt` methods in a pipeline of other `ReadWriterAt`s
- One such `ReadWriterAt` is the `ArbitraryReadWriterAt` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/arbitrary_rwat.go)
- It allows breaking down a larger data stream into smaller chunks
- In `ReadAt`, it calculates the index of the chunk that the offset falls into and the position within the offsets
- It then reads the entire chunk from the backend into a buffer, copies the necessary portion of the buffer into the input slice, and repeats the process until all requested data is read
- Similarly for the writer, it calculates the chunk's index and offset
- If an entire chunk is being written to, it bypasses the chunking system, and writes it directly to not have to unnecessarily copy the data twice
- If only parts of a chunk need to be written, it first reads the complete chunk into a buffer, modifies the buffer with the data that has changed, and writes the entire chunk back, until all data has been written
- This simple implementation can be used to allow for writing data of arbitrary length at arbitrary offsets, even if the backend only supports a few chunks
- In addition to this chunking system, there is also a `ChunkedReadWriterAt`, which ensures that the limits concerning the maximum size supported by the backend and the actual chunks are being respected
- This is particularly useful when the client is expected to do the chunking, and the server simply checks that the chunking system's chunk size is respected
- In order to check if a read or write is valid, it checks whether a read is done to an offset multiple of the chunk size, and whether the length of the slice of data to read/write is the chunk size (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/chunked_rwat.go)
- It is possible to do the chunking in two places; on the mount API's side, and on the (potentially remote) backend's side
- Doing the chunking on the backend's side is usually much faster than on the mount API's side, as writes with lengths smaller than the chunk size will mean that the remote chunk needs to be fetched first, significantly increasing the latency esp. in scenarios with high RTT
- Benchmark: Local vs. remote chunking
- While these systems already allow for some optimizations over simply using the NBD protocol over WAN, they still mean that chunks will only be fetched as they are being needed, which means that there still is a guaranteed minimum downtime
- In order to improve on this, a more advanced API (the managed mount API) was created
- A field that tries to optimize for this use case is live migration of VMs
- Live migration refers to moving a virtual machine, its state and connected devices from

one host to another with as little downtime as possible
- There are two types of such migration algorithms; pre-copy and post-copy migration
- Pre-copy migration works by copying data from the source to the destination as the VM continues to run (or in the case of a generic migration, app/other state continues being written to)
- First, the initial state of the VM's memory is copied to the destination
- If, during the push, chunks are being modified, they are being marked as dirty
- These dirty chunks are being copied over to the destination until the number of remaining chunks is small enough to satisfy a maximum downtime criteria
- Once this is the case, the VM is suspended on the source, and the remaining chunks are synced over to destination
- Once the transfer is complete, the VM is resumed on the destination
- This process is helpful since the VM is always available in full on either the source or the destination, and it is resilient to a network outage occurring during the synchronization
- If the VM (or app etc.) is however changing too many chunks on the source during the migration, the maximum acceptable downtime criteria might never get reached, and the maximum acceptable downtime is also somewhat limited by the available RTT
- An alternative to pre-copy migration
- In this approach, the VM is immediately suspended on the source, moved to the destination with only a minimal set of chunks
- After the VM has been moved to the destination, it is resumed
- If the VM tries to access a chunk on the destination, a page fault is raised, and the missing page is fetched from the source, and the VM continues to execute
- The benefit of post-copy migration is that it does not require re-transmitting dirty chunks to the destination before the maximum tolerable downtime is reached
- The big drawback of post-copy migration is that it can result in longer migration times, because the chunks need to be fetched from the network on-demand, which is very latency/RTT-sensitive
- For the managed mount API, both paradigms were implemented
- The managed mount API is primarily intended as an API for reading from a remote resource and then syncing changes back to it however, not migrating a resource between two hosts
- The migration API (will be discussed later) is a more well-optimized version for this use case
- The pre-copy API is implemented in the form of preemptive pulls based on another `ReaderAt`
- The `Puller` component asynchronously pulls chunks in the background (code snipped from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/puller.go)
- It allows passing in a pull heuristic function, which it uses to determine which chunks

should be pulled in which order
- Many applications commonly access certain bits of data first
- If a resource should be available locally as quickly as possible, then using the correct pull heuristic can help a lot
- For example, if the data pulled consists of a header, then using a pull heuristic that pulls these header chunks first can be of help
- If a file system is being synchronized, and the superblocks of the file system are being stored in a known pattern, the pull heuristic can be used to fetch these superblocks first
- If a format like MP4, which has an index, is used then said index can be fetched first, be accessed first and during the time needed to parse the index, the remaining data can be pulled in the background
- After sorting the chunks, the puller starts a fixed number of worker threads in the background, each of which ask for a chunk to pull
- Note that the puller itself does not copy to/from a destination; this use case is handled by a separate component
- It simply reads from the provided `ReaderAt`, which is then expected to handle the actual copying on its own
- The actual copy logic is provided by the `SyncedReadWriterAt` instead
- This component takes both a remote reader and a local `ReadWriterAt`
- If a chunk is read, e.g. by the puller component calling `ReadAt`, it is tracked and market as remote by adding it to a local map
- The chunk is then read from the remote reader and written to the local `ReadWriterAt`, and is then marked as locally available, so that on the second read it is fetched locally directly
- A callback is then called which can be used to monitor the pull process
- Note that if it is used in a pipeline with the `Puller`, this also means that if a chunk which hasn't been fetched asynchronously yet will be scheduled to be pulled immediately
- WriteAt also starts by tracking a chunk, but then immediately marks the chunk as available locally no matter whether it has been pulled before
- The combination of the `SyncedReadWriterAt` and the `Puller` component implements the pull post-copy system in a modular and testable way
- Unlike the usual way of only fetching chunks when they are available however, this system also allows fetching them pre-emptively, gaining some benefits of pre-copy migration, too
- Using this `Puller` interface, it is possible to implement a read-only managed mount
- This is very similar the `rr+` prefetching mechanism from "Remote Regions" (reference atc18-aguilera)
- In order to also be able to write back however, it needs to have a push system as well
- This push system is being started in parallel with the pull system

- It also takes a local and a remote `ReadWriterAt`
- Chunks that have changed/are pushable are marked with `MarkOffsetPushable` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L171C24-L185)
- This integrates with the callbacks supplied by the syncer, which ensures that we don't sync back changes that have been pulled but not modified, only the ones that have been changed locally
- Once opened, the pusher starts a new goroutine in the background which calls `Sync` in a set recurring interval (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/pushe
- Once sync is called by this background worker system or manually, it launches workers in the background
- These workers all wait for a chunk to handle
- Once a worker receives a chunk, it reads it from the local `ReadWriter`, and copies it to the remote
- Safe access is ensures by individually locking each chunk
- The pusher also serves as a step in the `ReadWriterAt` pipeline
- In order to do this, it exposes `ReadAt` and `WriteAt`
- `ReadAt` is a simple proxy, while `WriteAt` also marks a chunk as pushable (since it mutates data) before writing to the local `ReadWriterAt`
- For the direct mount, the NBD server was directly connected to the remote, while in this setup a pipeline of pullers, pushers, a syncer and an `ArbitraryReadWriter` is used (graphic of the four systems and how they are connected to each other vs. how the direct mounts work)
- For a read-only scenario, the `Pusher` step is simply skipped (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L142-L169)
- If no background pulls are enabled, the creation of the `Puller` is simply skipped (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go#L187-L222)
- This setup allows pulling from the remote `ReadWriterAt` before the NBD device is open
- This means that we can start pulling in the background as the NBD client and server are still starting (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_managed.go
L272)
- These two components typically start fairly quickly, but can still take multiple ms
- Often, it takes as long as one RTT, so parallelizing this startup process can significantly reduce the initial latency and pre-emptively pull quite a bit of data
- Benchmark: Parallelizing startups and pulling n MBs as the device starts
- Using this benchmark and simple interface also makes the entire system very testable

- In the tests, a memory reader or file can be used as the local or remote `ReaderWriterAt`s and then a simple table-driven test can be used (code snippet from https://github.com/pojntfx/r3map/blo
- With all of these components in place, the managed mounts API serves as a fast and efficient option to access almost any remote resource in memory
- Similarly to how the direct mounts API used the basic path mount to build the file and `mmap` interfaces, the managed mount builds on this interface in order to provide the same interfaces
- It is however a bit more complicated for the lifecycle to work
- For example, in order to allow for a `Sync()` API, e.g. the `msync` on the `mmap`ed file must happen before `Sync()` is called on the syncer
- This is done through a hooks system (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg L37)
- The same hooks system is also used to implement the correct lifecycle when `Close`ing the mount
- While the managed mounts API mostly works, there are some issues with it being implemented in Go
- This is mostly due to deadlocking issues; if the GC tries to release memory, it has to stop the world
- If the `mmap` API is used, it is possible that the GC tries to manage the underlying slice, or tries to release memory as data is being copied from the mount
- Because the NBD server that provides the byte slice is also running in the same process, this causes a deadlock as the server that provides the backend for the mount is also frozen (https://github.com/pojntfx/r3map/blob/main/pkg/mount/slice_managed.go#L70-L93)
- A workaround for this is to lock the `mmap`ed region into memory, but this will also cause all chunks to be fetched, which leads to a high `Open()` latency
- This is fixable by simply starting the server in separate thread, and then `mmap`ing
- Issues like this however are hard to fix, and point to Go potentially not being the correct language to use for this part of the system
- In the future, using a language without a GC (such as Rust) could provide a good alternative
- While the current API is Go-specific, it could also be exposed through a different interface to make it usable in Go
- A similar approach was made in RegionFS (reference atc18-aguilera)
- RegionFS is implemented as a kernel module, but it is functionally similar to how this API exposes a NBD device for memory interaction
- In RegionFS, the regions file system is mounted to a path, which then exposes regions as virtual files
- Instead of using a custom configuration (such as configuring the amount of pushers to make a mount read-only), such an approach makes it possible to use `chmod` on the virtual

file for a memory region to set permissions
  - By using standard utilities like `open` and `chmod`, this API usable from different programming languages with ease
  - Unlike the managed mounts API however, the system proposed in Remote Regions is mostly intended for private usecases with a limited amount of hosts and in LAN, with low-RTT connections
  - It is also not designed to be used for a potential migration scenarios, which the modular approach of r3map allows for
  - While Remote Regions' file system approach does allow for authorization based on permissions, it doesn't specify how authentication could work
  - In terms of the wire protocol, Remote Regions also seems to target mostly LAN with protocols like RDMA comm modules, while r3map targets mostly WAN with a pluggable transport protocol interface

- Pull-Based Memory Synchronization with Migrations
  - We have now implemented a managed mounts API
  - This API allows for efficient access to a remote resource through memory
  - It is however not well suited for a migration scenario
  - For migrations, more optimization is needed to minimize the maximum acceptable downtime
  - The flexible architecture of the `ReadWriterAt` components allow the reuse of lots of code for both use cases
  - For the migration, the process is split into two distinct phases
  - The same preemptive background pulls and parallelized device/syncer startup can be used, but the push process is dropped
  - The two phases allow pulling the majority of the data first, and only finalize the move later with the remaining data
  - This is inspired by the pre-copy approach to VM live migration, but also allows for some of the benefits of the post-copy approach as we'll see later
  - Why is this useful? A constraint for the mount-based API that we haven't mentioned before is that it doesn't allow safe concurrent access of a resource by two readers or writers at the same time
  - This poses a problem for migration, where the downtime is what should be optimized for, as the VM or app that is writing to the source device would need to be suspended before the transfer could begin
  - This adds very significant latency, which is a problem
  - The mount API was also designed in such a way as to make it hard to share a resource this way

- The remote backend for example API doesn't itself provide a mount to access the underlying data, which further complicates migration by not implementing a migration lifecycle
- To fix this, the migration API defines two new actors: The seeder and the leecher
- The seeder represents a resource that can be migrated from/a host that exposes a migratable resource
- The leecher represents a client that wants to migrate a resource to itself
- Initially, the protocol starts by running an application with the application's state on the seeder's mount
- When a leecher connects to the seeder, the seeder starts tracking any writes to it's mount
- The leecher starts pulling chunks from the seeder to it's local backend
- Once the leecher has received a satisfactory level of locally available chunks, it as the seeder to finalize, which then causes the seeder to stop the remote app, `msync`/flushes the drive, and returns the chunks that were changed between it started tracking and finalizing
- The leecher then marks these chunks as remote, immediately resumes the VM, and queues them to be pulled immediately
- By splitting the migration into these two distinct phases, the overhead of having to start the device on the leecher can be skipped and additional app initialization that doesn't depend on the app's state (e.g. memory allocation, connecting to databases, loading models etc.) can be performed before the application needs to be suspended
- This solution combines both the pre-copy algorithm (by pulling the chunks from the seeder ahead of time) and the post-copy algorithm (by resolving dirty chunks from the seeder after the VM has been migrated) into one coherent protocol (add state machine diagram here)
- This way, the maximum tolerable downtime can be drastically reduced, and dirty chunks don't need to be re-transmitted multiple times
- Effectively, it drops the maximum guaranteed downtime to the time it takes to `msync` the seeder's app state, the RTT and, if they are being accessed immediately, how long it takes to fetch the chunks that were written in between starting to track and finalize
- To achieve this, the seeder defines a simple read-only API with the familiar `ReadAt` methods, but also new APIs such as returning dirty chunks from `Sync` and adding a `Track` method (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go#L15-L21)
- Unlike the remote backend, a seeder also exposes a mount through a path, file or byte slice, so that as the migration is happening, the underlying data can still be accessed by the application
- The tracking aspect is implemented in the same modular and composable way as the syncer etc. - by using a `TrackingReadWriterAt` that is connected to the seeder's

`ReadWriterAt` pipeline (graphic of the pipeline here)
- Once activated by `Track`, the tracker intercepts all `WriteAt` calls and adds them to a local de-duplicated store (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/trac L40)
- When `Sync` is called, the changed chunks are returned and the de-duplicated store is cleared
- A benefit of the protocol being defined in such a way that only the client ever calls an RPC, not the other way around, is that the transport layer/RPC system is completely interchangeable
- This works by returning a simple abstract `service` utility struct struct from `Open` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go)
- This abstract struct can then be used as the backend for any RPC framework, e.g. for gRPC (code snippet https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder_grpc.go)
- The leecher then takes this abstract API as an argument
- As soon as the leecher is opened, it calls track on the seeder, and in parallel starts the device (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/migration/path_leecher.go#L1 L203)
- The leecher introduces a new component, the `LockableReadWriterAt`, into its internal pipeline (add graphic of the internal pipeline)
- This component simply blocks all read and write operations to/from the NBD device until `Finalize` has been called (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks L37)
- This is required because otherwise stale data (since `Finalize` did not yet mark the changed chunks) could have poisoned the cache on the e.g. `mmap`ed device
- Once the leecher has started the device, it sets up a syncer (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/migration/path_leecher.go#L214-L252)
- A callback can be used to monitor the pull process (code snippet from https://github.com/pojntfx/r3map/b migration-benchmark/main.go#L544-L548)
- As described before, after a satisfactory local availability level has been reached, `Finalize` can be called
- `Finalize` then calls `Sync()` on the remote, marks the changed chunks as remote, and schedules them also be pulled in the background (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/migration/path_leecher.go#L257-L280)
- As an additional measure aside from the lockable `ReadWriterAt` to make accessing the path/file/slice too early harder, only `Finalize` returns the managed object, so that the happy path can less easily lead to deadlocks

- After a leecher has successfully reached 100% local availability, it calls `Close` on the seeder and disconnects the leecher from the seeder, causing both to shut down (code snippet from https://github.com/pojntfx/r3map/blob/main/cmd/r3map-migration-benchmark-server/main.go#L137)
- Once the leecher has exited, a seeder can be started, to allow for migrating from the destination to another destination again
- A interesting question to ask with the two-step migration API is when to start the finalization step
- As is visible from the migration API protocol state machine showed beforehand, the finalization stage is critical and hard or impossible to recover from depending on the implementation
- While for the memory sync on its own, one could just call `Finalize` multiple times to restart it
- But since `Finalize` needs to return a list of dirty chunks, it requires the VM or app on the source device to be suspended before `Finalize` can return
- While not necessarily the case, such a suspend operation is not idempotent (since it might not just be a suspension that is required, but also a shutdown of dependencies etc.)
- "Reducing Virtual Machine Live Migration Overhead via Workload Analysis" provides an interesting analysis of options on how this decision of when to migrate can be made
- While being designed mostly for use with virtual machines, it could serve as a basis for other applications or migration scenarios, too
- The proposed method identifies workload cycles of VMs and uses this information to postpone the migration if doing so is beneficial
- This works by analyzing cyclic patters that can unnecessarily delay a VM's migration, and identifies optimal cycles to migrate VMs in from this information
- For the VM use case, such cycles could for example be the GC of a large application triggering a lot of changes to the VMs memory etc.
- If a migration is proposed, the system checks for whether it is currently in a beneficial cycle to migrate in in which case it lets the migration proceed; otherwise, it postpones it until the next cycle
- The algorithm uses a Bayesian classifier to identify a favorable or unfavorable cycle
- Compared to the alternative, which is usually waiting for a significant percentage of the chunks that were not changed before tracking started to be synced first, this can potentially yield a lot of improvements
- The paper has found an improvement of up to 74% in terms of live migration time/downtime and 43% in terms of the amount of data transferred over the network
- While such a system was not implemented for r3map, using r3map with such a system would certainly be possible

  - With these results in mind, it is interesting to look at how the migration API performs compared to the single-phase mount API
  - Benchmark: Maximum acceptable downtime for a migration scenario with the Managed Mount API vs the Migration API

- Optimizing Mounts and Migrations

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

pull-only architecture
- Usually, a pre-copy system pushes changes to the destination in the migration API
- This however makes such a system hard to use in a scenario where NATs exist, or a scenario in which the network might have an outage during the migration
- With a pull-only system emulating the pre-copy setup, the client can simply keep track of which chunks it still needs to pull itself, so if there is a network outage, it can just resume pulling like before, which would be much harder to implement with a push system as the server would have to track this state for multiple clients and handle the lifecycle there
- The pull-only system also means that unlike the push system that was implemented for the hash-based synchronization, a central forwarding hub is not necessary
- In the push-based system for the hash-based solution, the topology had to be static/all destinations would have to have received the changes made to the remote app's memory since it was started
- In order to cut down on unnecessary duplicate data transmissions, a central forwarding hub was implemented
- This central forwarding hub does however add additional latency, which can be removed completely with the migration protocol's P2P, pull-only algorithm
- In WAN, where latency is high, the ability to fetch chunks concurrently is very important
- Without concurrent background pulls, latency adds up very quickly as every memory request would have at least the RTT as latency
- The first prerequisite for supporting this is that the remote backend has to be able to read from multiple regions without locking the backend globally
- For the file backend for example, this is not the case, as the lock needs to be acquired for the entire file before an offset can be accessed (code snippet from https://github.com/pojntfx/go-nbd/blob/main/pkg/backend/file.go#L17-L25)
- For high-latency scenarios, this can quickly become a bottleneck
- While there are many ways to solve this, one is to use the directory backend
- Instead of using just one backing file, the directory backend is a chunked backend that uses a directory with one file for each chunk instead of a global file
- This means that the directory backend can lock each file individually, speeding up concurrent access
- This also applies to writes, where even concurrent writes to different chunks can be done at the same time as they are all backed by a separate file
- The directory backend keeps track of these chunks by using an internal map of locks (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/directory.go#L22-L24)
- When a chunk is first accessed, a new file is created for the chunk (code snipped from https://github.com/pojntfx/r3map/blob/main/pkg/backend/directory.go#L77-L94)

- If the chunk is being read, the file is also truncated to one chunk length
- Since this could easily exhaust the number of maximum allowed file descriptors for a process, a check is added (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/ L75)
- If the maximum allowed number of open files is exceeded, the first file is closed and removed from the map, causing it to be reopened on a subsequent read
- These optimizations add an initial overhead to operations, but can significantly improve the pull speed in scenarios where the backing disk is slow or the latency is high
- Benchmark: File vs. directory backend performance
- 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 the migration APIs are transport-independent
- A simple RPC framework to use is dudirekta
- Dudirekta is reflection-based, which makes it very simple to use to iterate on the protocol quickly
- To use it, a simple wrapper struct with the needed RPC methods is created (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/backend.go#L41-L61)
- This wrapper struct simply calls the backend (or seeder etc.) functions
- The wrapper struct is then passed as the local function struct into a registry, which creates the RPC server (code snippet from https://github.com/pojntfx/r3map/blob/main/cmd/r3map-mount-benchmark-server/main.go#L146-L166)
- When the transport protocol, in this case TCP, `accept`s a client it is linked to the registry (code snippet from https://github.com/pojntfx/r3map/blob/main/cmd/r3map-mount-benchmark-server/main.go#L198-L200)
- The used protocol is very simple (code snippet from https://github.com/pojntfx/dudirekta#protocol)
- If an RPC, such as `ReadAt`, is called, it is looked up via reflection and validated (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L323-L357)
- The arguments, which have been supplied as JSON, are then unmarshalled into their native types, and the local wrapper struct's method is called in a new goroutine (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L417-L521)
- This allows for one important feature: Concurrent RPC calls
- Many simple RPC frameworks only support one RPC call at a time, because no demuxing is implemented
- For example, when dRPC (https://github.com/storj/drpc) was used, drastic performance issues were noted compared to gRPC etc., because no support for concurrent RPCs was implemented
- To use a RPC backend on the destination side, the wrapper struct's remote representation

is used (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/backend.go#L14-L19)
- For the destination site, the remote representation's fields are iterated over, and replaced by functions which marshal and unmarshal the function calls into the dudirekta JSON protocol (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L228-L269)
- To do this, the arguments are marshalled into JSON, and a unique call ID is generated (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pkg/rpc/registry.go#L109-L130)
- Once the remote has responded with a message containing the unique call ID, it unmarshalls the arguments, and returns (code snippet from https://github.com/pojntfx/dudirekta/blob/main/pk L217)
- This makes both calling RPCs and defining them completely transparent to the user
- Since dudirekta has a few limitations (such as the fact that slices are passed as copies, not references, and that context needs to be provided), the resulting remote struct can't be used directly
- To work around this, the standard `go-nbd` backend interface is implemented for the remote representation, creating a universally reusable, generic RPC backend wrapper (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/rpc.go)
- The same backend implementation is also done for the seeder protocol (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder.go)
- While the dudirekta RPC serves as a good reference implementation of the basic RPC protocol, it does not scale particularly well
- This mostly stems from two aspects of how it is designed
- JSON(L) is used for the wire format, which while simple and easy to analyze, is slow to marshal and unmarshal
- Dudirekta supports defining functions on both the client and the server
- This is very useful for implementing e.g. a pre-copy protocol where the source pushes chunks to the destination by simply calling a RPC on the destination
- Usually, RPCs don't support exposing or calling RPCs on the client, too, only on the server
- This would mean that in order to implement a pre-copy protocol with pushes, the destination would have to be `dial`able from the source
- In a LAN scenario, this is easy to implement, but in WAN it is complicated and requires authentication of both the client and the server
- Dudirekta fixes this by making the protocol itself bi-directional (example and code snippet from https://github.com/pojntfx/dudirekta/tree/main#1-define-local-functions)
- In addition to this, dudirekta works over any `io.ReadWriter`
- This does however come at the cost of not being able to do connection pooling, since each

client `dial`ing the server would mean that the server could not reference the multiple client connections as one composite client without changes to the protocol
- While implementing such a pooling mechanism in the future could be interesting, it turned out to not be necessary thanks to the pull-based pre-copy solution described earlier
- Instead, only calling RPCs exposed on the server from the client is the only requirement for an RPC framework, and other, more optimized RPC frameworks can already offer this
- Dudirekta uses reflection to make the RPCs essentially almost transparent to use
- By switching to a well-defined protocol with a DSL instead, we can gain further benefits from not having to use reflection and generating code instead
- One popular such framework is gRPC
- gRPC is a high-performance RPC framework based on Protocol Buffers
- Because it is based on Protobuf, we can define the protocol itself in the `proto3` DSL
- The DSL also allows to specify the order of each field in the resulting wire protocol, making it possible to evolve the protocol over time without having to break backwards compatibility
- This is very important given that r3map could be ported to a language with less overhead in the future, e.g. Rust, and being able to re-use the existing wire protocol would make this much easier
- While dudirekta is a simple protocol that is easy to adapt for other languages, currently it only supports Go and JS, while gRPC supports many more
- A fairly unique feature of gRPC are streaming RPCs, where a stream of requests can be sent to/from the server/client, which, while not used for the r3map protocol, could be very useful to implementing a pre-copy migration API with pushes similarly to how dudirekta does it by exposing RPCs from the client
- As mentioned before, for WAN migration or mount scenarios, things like authentication, authorization and encryption are important, which gRPC is well-suited for
- Protobuf being a proper byte-, not plaintext-based wire format is also very helpful, since it means that e.g. sending bytes back from `ReadAt` RPCs doesn't require any encoding (wereas JSON, used by dudirekta, `base64` encodes these chunks)
- gRPC is also based on HTTP/2, which means that it can benefit from existing load balancing tooling etc. that is popular in WAN for web uses even today
- This backend is implemented by first defining the protocol in the DSL (code snippet from https://github.com/pojntfx/r3map/blob/main/api/proto/migration/v1/seeder.proto)
- After generating the bindings, the generated backend interface is implemented by using the dudirekta wrapper struct as the abstraction layer (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder_grpc.go)
- Unlike dudirekta, gRPC also implements concurrent RPCs and connection pooling
- Similarly to how having a backend that allows concurrent reads/writes can be useful to

speed up the concurrent push/pull steps, having a protocol that allows for concurrent RPCs can do the same
- Connection pooling is another aspect that can help with this
- Instead of either using one single connection with a multiplexer (which is possible because it uses HTTP/2) to allow for multiple requests, or creating a new connection for every request, gRPC is able to intelligently re-use existing connections for RPCs or create new ones, speeding up parallel requests
- Despite these benefits, gRPC is not perfect however
- Protobuf specifically, while being faster than JSON, is not the fastest serialization framework that could be used
- This is especially true for large chunks of data, and becomes a real bottleneck if the connection between source and destination would allow for a high throughput
- This is where fRPC, a RPC library that is easy to replace gRPC with, becomes useful
- fRPC is 2-4x faster than gRPC, and especially in terms of throughput (insert graphics from https://frpc.io/performance/grpc-benchmarks)
- Because throughput and latency determine the maximum acceptable downtime of a migration/the initial latency for mounts, choosing the right RPC protocol is an important decision
- fRPC also uses the same proto3 DSL, which makes it an easy drop-in replacement, and it also supports multiplexing and connection polling
- Because of these similarities, the usage of fRPC in r3map is extremely similar to gRPC (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/services/seeder_frpc.go)
- A good way to test how well the RPC framework scales for concurrent requests is scaling the amount of pull workers, where dudirekta only gains marginally from increasing their number, whereas both gRPC and fRPC can increase throughput and decrease the initial latency by pulling more chunks pre-emptively
- Benchmark: Effect of tuning the amount of push/pull workers in high-latency for these three backends on latency till first n chunks and throughput
- These backends provide a way to access a remote backend
- This is useful, esp. if the remote resource should be protected in some way or if it requires some kind of authorization
- Depending on the use case however, esp. for the mount API, having access to a remote backend without this level of indirection can be useful
- Fundamentally, a mount maps fairly well to a remote random-access storage device
- Many existing protocols and systems provide a way to access essentially this concept over a network
- One of these is Redis, an in-memory key-value store with network access
- Chunk offsets can be mapped to keys, and bytes are a valid key type, so the chunk itself can

be stored directly in the KV store (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/ba
L63)
- Using Redis is particularly useful because it is able to handle the locking server-side in a very efficient way, and is well-tested for high-throughput etc. scenarious
- Authentication can also be handled using the Redis protocol, so can multi-tenancy by using multiple databases or a prefix
- Redis also has very fast read/write speeds due to its bespoke protocol and fast serialization
- While the Redis backend is very useful for read-/write usage, when deployment to the public internet is required, it might not be the best one
- The S3 backend is an good choice for mapping public information, e.g. media assets, binaries, large read-only filesystems etc. into memory
- S3 used to be an object storage service from AWS, but has since become a more or less standard way for accessing blobs thanks to open-source S3 implementations such as Minio
- Similarly to how files were used as individual files, one S3 object per chunk is used to store them
- S3 is based on HTTP, and like the Redis backend requires chunking due to it not supporting updates of part of a file
- In order to prevent having to store empty data, the backend interprets "not found" errors as empty chunks
- Another backend option is a NoSQL server such as Cassandra
- This is more of a proof of concept than a real usecase, but shows the versitility and flexibility of how a database can be mapped to a memory region, which can be interesting for accessing e.g. a remote database's content without having to use a specific client
- `ReadAt` and `WriteAt` are implemented using Cassandra's query language (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/backend/cassandra.go#L36-L63)
- Similarly to Redis, locking individual keys can be handled by the DB server
- But in the case of Cassandra, the DB server stores chunks on disk, so it can be used for persistent data
- In order to use Cassandra, migrations have to be applied for creating the table etc. (code snippet from https://github.com/pojntfx/r3map/blob/main/cmd/r3map-direct-mount-benchmark/main.go#L369-L396)
- Benchmark: These three backends on localhost and on remote hosts, where they could be of use
- Another interesting aspect of optimization to look at is the overhead of managed mounts
- It is possible for managed mounts (and migrations) to deliver signficantly lower performance compared to direct mounts
- This is because using managed mounts come with the cost of potential duplicate I/O oper-

ations
  - For example, if memory is being accessed linearly from the first to the last offset immediately after it being mounted, then using the background pulls will have no effect other than causing a write operation (to the local caching backend) compared to just directly reading from the remote backend
  - This however is only the case in scenarios with a very low latency between the local and remote backends
  - If latency becomes higher, then the ability to pull the chunks in the background and in parallel with the puller will offset the cost of duplicate I/O
  - The same applies to slow local backends, e.g. if slow disks or memory are being used, which can mean that offsetting the duplicate I/O will need a significantly higher latency to be worth it
  - Benchmark: Latency and throughput of all benchmarks on localhost and in a realistic latency and throughput scenario (direct mounts can outperform managed mounts in tests on localhosts)

- Case Studies
  - ram-dl is a fun experiment
  - Tech demo for r3map
  - Uses the fRPC backend to expand local system memory
  - Can allow mounting a remote system's RAM locally
  - Can be used to inspect a remote system's memory contents
  - Is based on the direct mount API
  - Uses mkswap, swapon and swapoff (code snippet from https://github.com/pojntfx/ram-dl/blob/main/cmd/ram-dl/main.go#L170-L190)
  - Enables paging out to the block device provided by the direct mount API
  - `ram-ul` "uploads" RAM by exposing a memory, file or directory-backed file over fRPC
  - `ram-dl` then does all of the above
  - Not really intended for real-world usecases, but does show that this can be used for interesting usecases such as real, remote RAM/Swap
  - Shows how easy it is to use the resulting library, as the entire project is just ~300 SLOC including backend configuration, flags and other boilerplate
  - tapisk is an interesting usecase because of how close it is to STFS, which provided the inspiration for the FUSE-based approach
  - Very high read/write backend latency (multiple seconds, up to 90s, due to seeking)
  - Linear access, no random reads
  - Can benefit a lot from asynchronous writes provided by managed mounts
  - Fast storage acts as a caching layer

- Backend is linear, so only one read/write possible at a time
- With local backend, writes are de-duplicated automatically and both can be asynchronous/concurrent
- Writes go to fast ("local") backend first, syncer then handles in both directions
- Chunking works on tape drive records and blocks
- Only one concurrent reader/writer makes sense
- Syncing intervals to/from can maybe be minutes or more to make it more efficient (since long, connected write intervals prevent having to seek to a different position on disk)
- Tape backends can also be used to access large amounts of linear data (terrabytes) from a tape as though it were in memory
- Makes it possible to access even large datasets or backups in a way that is comfortable
- If a file system (e.g. EXT4) is written to the tape, the tape can be mounted as a random-access device
- Tape-specific backend is implemented as the `backend.Backend` interface from go-nbd
- Backend can be re-used outside of tapisk, too (e.g. as a `ram-dl` backend)
- `bbolt` DB is used as an index
- Index maps simulated offsets to real tape records (code snippet from https://github.com/pojntfx/tapisk/blo L68)
- When reading, first the tape looks up the real tape record for the requested offset (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/backend/tape.go#L71-L78)
- Seeking can then use the accelerated `MTSEEK` ioctl to fast-forward to a record on the tape (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/mtio/tape.go#L25-L40)
- After seeking to the block, the chunk is read from the tape into memory
- When writing the drive seeks to the end of the tape (unless the last operation was a write already, in which case we're already at the end) (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/mtio/tape.go#L57-L72)
- After that, the current real tape block position is requested, stored in the index, and the offset is written to the tape (code snippet from https://github.com/pojntfx/tapisk/blob/main/pkg/backend/ta L119)
- This essentially makes the tape a chunked, reusable `ReadWriterAt`, in the same way as the directory backend
- By using a RPC backend, a remote tape can be accessed in the same way, making it possible to map e.g. a remote library's tape robot to your local memory
- `tapisk` is a unique usecase that shows the versitility of the approach chosen and how flexible it is
- E.g. the chunking system didn't have to be reimplemented (like with STFS) - we could just

use the managed mount API directly without changes
- Can actually be a real usecase to replace LTFS
- LTFS is a kernel module filesystem for tape drives that makes them a file system the same way as STFS
- But is its own filesystem, while tapisk allows using any existing and tested filesystem on top of the generic block device
- Doesn't support the caching, making it hard to use for memory mapping, too
- Tapisk also shows how minimal it is: While LTFS is 10s of thousands of SLOC, tapisk achieves the same and more in just under 350 SLOC
- Another potential usecase is using r3map to create a unique mountable remote filesystem
- Currently there are two choices on how these can be implemented
- Google Drive, Nextcloud etc. listen to file changes on a folder and synchronize files when they change
- The big drawback is that everything needs to be stored locally
- If a lot of data is stored (e.g. terabytes), the locally available files would need to be manually selected
- There is no way to dynamically download files this way as they are required
- It is however very efficient, since the filesystem is completely transparent to the user (writes are being synced back asynchronously)
- It also supports an offline usecase easily
- The other option is to use a FUSE, e.g. `s3-fuse`
- This means that files can be fetched on demand
- But comes with a heavy performance penalty
- Writes are usually done (esp. for `s3-fuse`) when they are communicated to the FUSE by the kernel, which makes writes very slow
- Offline usage is also hard/usually impossible
- Also usually not a fully featured system (e.g. not `inotify` support, missing permissions etc.)
- This leaves the choice between two imperfect systems, all with their own downsides
- Using r3map can combine both approaches by both not having to download all files in advance and being able to write back changes asynchronously
- It is also a fully-featured filesystem
- Files can also be downloaded preemptively for offline access, just like with e.g. the Google Drive approach
- This can be achieved by using the managed mount API
- The block device is formatted using any valid filesystem (e.g. EXT4) and then mounted
- If files should be downloaded in the background, the amount of pull workers can be set to anything `>0`

- Reads to blocks/files that haven't been read yet can be resolved from the remote backend, giving a FUSE-like experience
- Since read block are then written to the local one, making subsequent reads almost as fast as native disk reads (unlike in FUSE/like in the Nextcloud approach)
- Writes are done to the local backend and can then be synced up in periodic intervals like with the Nextcloud approach, making them much faster than a FUSE, too
- Similarly to how the mount API is used, the migration API could then be used to move the file system between two hosts in a highly efficient way and without a long downtime
- This essentially creates a system that bridges the gap between FUSE and file synchronization, once again showing how the memory synchronization tooling can be used to solve essentially the synchronization of any state, including disks
- Another usecase is accessing a remote database locally
- While using a database backend is one option of storing chunks, an actual database can also be stored in a mount as well
- Particularly interesting for in-memory or on-disk databases like SQLite
- Instead of having to download the entire SQLite database before using it, it can simply be mounted, and accessed as it is being used
- This allows very efficient network access, as only very rarely the entire database is needed
- Since reads are cached with the managed mount API, only the first read should potentially have a performance impact
- Similarly so writes to the database will be more or less the same throughput as to the local disk, since changes are written back asynchronously
- If the full database should eventually be accessible locally, the background pullers can be used
- If the location of e.g. indexes is known, then the pull heuristic can be specified to fetch these first to speed up initial queries
- SQLite itself doesn't have to support a special interface or changes for this to work, since it can simply use the mount's block device as it's backend, which shows how universally the concept is usable
- This concept doesn't just apply to SQLite however
- Other uses could for example be to make formats that are usually not streamable due to technical limitations
- One such format is famously MP4
- Usually, if someone downloads a MP4, they can't start playing it before they have finished downloading
- This is because MP4s store metadata at the end of the file (graphic with metadata)
- The reason for storing this at the end is that generating the metadata requires encoding the video first

- Usually, if the file is downloaded from start to finish, then this metadata wouldn't be accessible, and simply inverting the download order wouldn't work since the stream would still be stored in different offsets of the file
- While there are ways of changing this (for example by storing the metadata in the beginning after the entire MP4 has been transcoded), this is not possible for already existing files
- With r3map however, the pull heuristic function can be used to immediately pre-fetch the metadata, and the individual chunks of the MP4 file are then being fetched in the background or as they are being accessed
- This allows making a format such as MP4 streamable before it has been fully downloaded yet, or making almost any format streamable no matter the codec without any changes required to the video/audio player or the format
- Another use case of this in-place streaming approach could be in gaming
- Usually, all assets of a game are currently being downloaded before the game is playable
- For many new AAA games, this can be hundreds of gigabytes
- Many of these assets aren't required to start the game however
- Only some of them (e.g. the launcher, UI libraries, first level etc.) are
- In theory, one could design a game engine in such a way that these things are only fetched from a remote as they are required
- This would however require changing the way new games work, and would not work for existing games
- Playing games of e.g. a network drive or FUSE is also unrealistic due to performance limitations
- Using the managed mount API, the performance limitations of such an approach can be reduced, without changes to the game
- By using the background pull system, reads from chunks that have already been pulled are almost as fast as native disk reads
- By analyzing the data access pattern of an existing game, a pull heuristic function can be created which will then preemptively pull the game assets that are required first, keeping latency spikes as low as possible
- Using the chunk availability hooks, the game could also be paused for a loading screen until a certain local availability percentage is reached to prevent latency spikes from missing assets, while also still allowing for faster startup times
- The same concept could also be applied to launching any application
- For many languages, completely loading a binary or script into memory isn't required for it to start executing
- Without significant changes to most interpreters and language VMs however, loading a file from a file system is the only available interface for loading programs

- With the managed mount API, this interface can be re-used to add streaming execution support to any intepreter or VM, simply by using the managed mount either as a filesystem to stream multiple binaries/scripts/libraries from, or by using the block device as a file or `mmap`ing it directly to fetch it
- Synchronization of app state is hard
- Even for hand-off scenarios a custom protocol is built most of the times
- It is possible to use a database sometimes (e.g. Firebase) to synchronize things
- But that can't sync all data structures and requires using specific APIs to sync things
- What if you could mount and/or migrate any ressource?
- Usually these structures are marshalled, sent over a network, received on a second system, unmarshalled, and then are done being synced
- Requires a complex sync protocol, and when to sync, when to pull etc. is inefficient and usually happens over a third party (e.g. a database)
- Data structures can almost always be represented by am `[]byte`
- If the data structures are allocated from on a `[]byte` from the block device, we can use the managed mount/direct mount/migration APIs to send/receive them or mount them
- This makes a few interesting usecases possible
- For example, one could imagine a TODO app
- By default, the app mounts the TODO list as a byte slice from a remote server
- Since authentication is pluggable and e.g. a database backend with a prefix for the user can be used, this could scaly fairly well
- Using the pre-emptive background pulls, when the user connects, they could stream the byte slice in from the remote server as the app is accessing it, and also pre-emptively pull the rest
- If the TODO list is modified by changing it in memory, these changes are automatically being written to the local backend
- The asynchronous push algorithm can then take these changes and sync them up to the remote backend when required
- If the local backend is persistent, such a setup can even survive network outages
- Locking is not handled by this, only one user is supported
- The in-memory representation would also need to be universal
- This is not the case with Go and for different CPU architectures
- But if e.g. Go is compiled to Wasm, and the Wasm VM's linear memory is pointed to the NBD device, it is
- This would also allow storing in the entire app state in the Wasm remote, with no changes to the app itself
- More interesting possibly than this smart mount, where the remote struct is completely transparent to the user, could be the migration usecase

  - If a user has the TODO app running on a phone, but wants to continue writing on their desktop, they can migrate the app's state with r3map
  - In this usecase, the migration API could be used
  - The pre-copy phase could be automatically detected if the phone comes physically close to the computer
  - Since this would be a LAN scenario, the migration would be very fast and allow for interesting hand-off usecases
  - The migration API also provides hooks and the protocol that would ensure that the app would be closed on the phone before it would be resumed on the desktop, making state corruption during the migration much harder than with a third-party that changes are synced (where the remote state may not be up-to-date yet as it is resumed)
  - Migrating an app's TODO list or any internal data structure is one option
  - Using a toolkit like Apache Arrow can provide a universal format for this memory, but many other things are possible if a layer of indirection is added
  - As mentioned before, a Wasm VM's memory could also be migrated this way, allowing for any Wasm app to be transfered between hosts
  - Similarly so, the Wasm app's binary, a mounted WASI filesystem etc. could all be synchronized this way, too
  - Due to the flexible nature of remap, it is possible to bring the semantics of VM live migration - moving running VMs between hosts - to anything that has state
  - r3map can also be used for actual VM migration, with any hypervisor that supports mapping a VM's memory to a block device
  - This could also be other VMs (like e.g. the JVM), a browser's JS VM, or - if the processor architectures etc. match - an entire process's memory space
  - Using the mount API, it would also be possible to bring the concept of VM snapshots to almost any app
  - Thanks to the preemptive pull mechanism, restoring e.g. a Wasm VM outside a migration would also be a very fast operation
  - Essentially, its VM memory migration and snapshots, but for any kind of state
  - Its particularly interesting because it can do so without any specific requirements for the data structures of the state other than them being ultimately a `[]byte`, which by definition every state is (def. a process level or VM level etc.)

- Conclusion

  - Looking back at all synchronization options and comparing ease of implementation, CPU load and network traffic between them
  - Summary of the different approaches, and how the new solutions might make it possible to use memory as the universal access format

  - Further research recommendations (e.g. `ublk`)

3.1 IMRAD (Introduction, Methods, Results and Discussion) Structure

- Introduction
  - Memory management in Linux
  - Memory as the universal storage API
  - What would be possible if memory would be the universal way to access resources?
  - Why efficient memory synchronization is the missing key component
  - High-level use cases for memory synchronization in the industry today
- Prior Works
  - Pull-Based Memory Synchronization with `userfaultfd`
    * Page faults occur when a process tries to access a memory region that has not yet been mapped into a process' address space.
    * By listening to these page faults, we know when a process wants to access a specific piece of memory.
    * We can use this knowledge to pull the chunk of memory from a remote, mapping it to the address on which the page fault occurred, thus only fetching data when it is needed.
    * Ordinarily, handling page faults is a task for the kernel.
    * In our case, we want to handle page faults in user space.
    * In the past, this used to be possible by handling the `SIGSEGV` signal in the process.
    * However, we can use a recent system called `userfaultfd` to do this in a more elegant way (available since kernel 4.11).
    * `userfaultfd` allows userspace handling of these page faults.
  - Push-Based Memory Synchronization with `mmap` and Hashing
    * This approach tries to improve on `userfaultfd` by switching to push-based synchronization method
    * Instead of reacting to page faults, this one a file to track changes to a memory region
    * By synchronizing the file representing the memory region between two systems, we can effectively synchronize the memory region itself
    * In Linux, swap space allows Linux to move chunks of memory to disk or other swap partition if the fast speed of RAM is not needed ("paging out")
    * Similarly to this, Linux can also load missing chunks from a disk
  - Push-Pull Memory Synchronization with a FUSE

    * Since the push method requires polling and is very CPU and I/O intensive, and `userfaultfd-go` has too low of a throughput, a better solution is needed
    * What if we could still get the events for the writes and reads without having to use `userfaultfd-go` or hashing?
    * We can create a custom file system in Linux and load it as a kernel module
    * This file system could then intercept reads/writes to/from the `mmap`ed region, making it possible to respond to them with a custom backend
    * But such a system would need to run in the kernel directly, which leads to a lot of potential drawbacks
  - Pull-Based Memory Synchronization with NBD
    * As hinted at before, a better API would be able to catch reads/writes to a single `mmap`ed file instead of having to implement a complete file system
    * It does however not have to be an actual file, a block device also works
    * In Linux, block devices are (typically storage) devices that support reading/writing fixed chunks (blocks) of data
    * We can `mmap` a block device in the same way that we can `mmap` a file
    * Similarly to how a file system can be implemented in a kernel module, a block device is typically implemented as a kernel module/in kernel space
    * However, the same security, portability and developer experience issues as with the former also apply here
  - NBD (Network Block Device) Protocols and Shortcomings
    * Usually, the NBD server and client don't run on the same system
    * NBD was originally designed to used as a LAN protocol to access a remote hard disk
    * As mentioned before, NBD can run over WAN, but is not designed for this
    * The biggest problem with running NBD over a public network, even if TLS is enabled is latency
    * Individual chunks would only be fetched to the local system as they are being accessed, adding a guaranteed minimum latency of at least the RTT
  - Push-Pull Memory Synchronization and Existing Migration Schemes
    * A field that tries to optimize for this use case is live migration of VMs
    * Live migration refers to moving a virtual machine, its state and connected devices from one host to another with as little downtime as possible
    * There are two types of such migration algorithms; pre-copy and post-copy migration
    * Pre-copy migration works by copying data from the source to the destination as the VM continues to run (or in the case of a generic migration, app/other state continues being written to)
    * An alternative to pre-copy migration

    * This is very similar the `rr+` prefetching mechanism from "Remote Regions" (reference atc18-aguilera)
  - Push-Pull Memory Synchronization with Mounts
    * This setup allows pulling from the remote `ReadWriterAt` before the NBD device is open
    * This means that we can start pulling in the background as the NBD client and server are still starting (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_r L272)
    * Often, it takes as long as one RTT, so parallelizing this startup process can significantly reduce the initial latency and pre-emptively pull quite a bit of data
    * Similarly to how the direct mounts API used the basic path mount to build the file and `mmap` interfaces, the managed mount builds on this interface
  - Reducing Virtual Machine Live Migration Overhead via Workload Analysis
    * "Reducing Virtual Machine Live Migration Overhead via Workload Analysis" provides an interesting analysis of options on how this decision of when to migrate can be made
    * While being designed mostly for use with virtual machines, it could serve as a basis for other applications or migration scenarios, too
    * The proposed method identifies workload cycles of VMs and uses this information to postpone the migration if doing so is beneficial
    * This works by analyzing cyclic patters that can unnecessarily delay a VM's migration, and identifies optimal cycles to migrate VMs in from this information
    * For the VM use case, such cycles could for example be the GC of a large application triggering a lot of changes to the VMs memory etc.
    * If a migration is proposed, the system checks for whether it is currently in a beneficial cycle to migrate in in which case it lets the migration proceed; otherwise, it postpones it until the next cycle
    * The algorithm uses a Bayesian classifier to identify a favorable or unfavorable cycle
    * The paper has found an improvement of up to 74% in terms of live migration time/-downtime and 43% in terms of the amount of data transferred over the network

- Fundamentals
  - Implementing `userfaultfd` in Go
    * Implementing this in Go was tricky, requiring the use of `unsafe`.
    * The `syscall` and `unix` packages can be used to interact with `ioctl` etc.
    * The `ioctl` syscall can be used to get a file descriptor to the `userfaultfd` API, and the API can be registered to handle any faults on the region (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/register.go#L15).

    * The requisite region can be allocated with e.g. `mmap`.
  - Memory Synchronization Fundamentals
    * This works similarly to how `userfaultfd` handled page faults, except this time it doesn't need to go through user space, which can make it much faster
    * We can do this by using `mmap`, which allows us to map a file into memory
    * By default, `mmap` doesn't write changes from a file back into memory, no matter if the file descriptor passed to it would allow it to or not
    * We can however add the `MAP_SHARED` flag; this tells the kernel to write back changes to the memory region to the corresponding regions of the backing file
    * Linux caches reads to such a backing file, so only the first page fault would be answered by fetching from disk, just like with `userfaultfd`
    * The same applies to writes; similar to how files need to be `sync`ed in order for them to be written to disks, `mmap`ed regions need to be `msync`ed in order to flush changes to the backing file
    * In order to synchronize changes to the region between hosts by syncing the underlying file, we need to have the changes actually be represented in the file, which is why `msync` is critical
  - Kernel Modules and Custom File Systems
    * While it is possible to write kernel modules with Rust instead of C these days, a lot of problems remain
    * Kernel modules aren't portable; they are built for a specific kernel, which makes them hard to distribute to users
    * A kernel file system is able to skip having to go through user space and thus save on context switching, but that will also mean that it will run in the kernel address space, making for a poor level of isolation
    * Iterating on a kernel module is much harder than iterating on a program running in user space
  - Pull-Based Memory Synchronization with NBD
    * Instead of implementing a FUSE to solve this, we can create a NBD (network block device) server that can be used by the kernel NBD module similarly to how the process that connected to the FUSE kernel module functioned
    * The difference between a FUSE and NBD is that a NBD server doesn't provide a file system
    * A NBD server provides the storage device that (typically) hosts a file system, which means that the interface for it is much, much simpler
    * The implementation overhead of a NBD server's backend is much more similar to how `userfaultfd-go` works, rather than a FUSE

    * NBD uses a protocol to communicate between a server (provided by user space) and a client (provided by the NBD kernel module)
  - Idea of Mounts as a Layer of Indirection
    * Instead of directly connecting a client to a remote server, we add a layer of indirection, called a `Mount` that consists of both a client and a server, both running on the local system
    * We then connect the server to the backend with an API that is better suited for WAN usage
  - Memory Synchronization and Chunking
    * One such `ReadWriterAt` is the `ArbitraryReadWriterAt` (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/chunks/arbitrary_rwat.go)
    * In addition to this chunking system, there is also a `ChunkedReadWriterAt`, which ensures that the limits concerning the maximum size supported by the backend and the actual chunks are being respected
  - Understanding the Push-Pull System
    * In order to also be able to write back however, it needs to have a push system
    * This integrates with the callbacks supplied by the syncer, which ensures that we don't sync back changes that have been pulled but not modified
    * The pusher also serves as a step in the `ReadWriterAt` pipeline
  - Pull-Based Memory Synchronization with Migrations: Problem Definition
    * We have now implemented a managed mounts API
    * This API allows for efficient access to a remote resource through memory
    * It is however not well suited for a migration scenario
    * For migrations, more optimization is needed to minimize the maximum acceptable downtime
    * The flexible architecture of the `ReadWriterAt` components allow the reuse of lots of code for both use cases
    * For the migration, the process is split into two distinct phases
    * The same preemptive background pulls and parallelized device/syncer startup can be used, but the push process is dropped
    * The two phases allow pulling the majority of the data first, and only finalize the move later with the remaining data
    * This is inspired by the pre-copy approach to VM live migration, but also allows for some of the benefits of the post-copy approach as we'll see later

- Methods and Implementation
  - Implementation of Pull-Based Memory Synchronization

    * Once we have acquired the file descriptor for the `userfaultfd` API, we need to transfer this descriptor to the process that should respond with the chunks of memory to be put into the faulting address.
    * File descriptors can be passed between processes using a UNIX socket (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/transfer/unix.go).
    * Once we have received the socket, the handler for the API needs to be registered.
    * If the handler receives an address that has faulted, it responds with the `UFFDIO_COPY` `ioctl` and a pointer to the memory chunk that should be used on the file descriptor (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/pkg/mapper/handler.go).
  - Changes Detection and Synchronization
    * For files, you can use `O_DIRECT` to skip this kernel caching if your process already does caching on its own, but this flag is ignored by the `mmap`
    * Usually, one would use `inotify` to watch changes to a file
    * `inotify` allows applications to register handlers on a file's events, e.g. `WRITE` or `SYNC`. This allows for efficient file synchronization, and is used by many file synchronization tools
    * It is also possible to filter only the events that we need to sync the writes, making it the perfect choice for this use case
    * For technical reasons however (mostly because the file is represented by a memory region), Linux doesn't fire these events for `mmap`ed files though, so we can't use it
    * The next best option are two: Either polling for file attribute changes (e.g. last write), or by continously hashing the file to check if it has changed
    * Instead of hashing the entire file, then syncing the entire file, we can want to really sync only the parts of the file that have changed between two polling iterations
    * We have implemented a simple TCP-based protocol for this delta synchronization, just like rsync's delta synchronization algorithm (code snippet from https://github.com/loopholelabs/darkmagyk/blob/master/cmd/darkmagyk-orchestrator/main.go#L1337-L1411 etc.)
  - FUSE Implementation and File System Abstraction
    * In order to implement file systems in user space, we can use the FUSE API
    * Here, a user space program registers itself with the FUSE kernel module
    * This program provides callbacks for the file system operations, e.g. for `open`, `read`, `write` etc.
    * Unlike a file system implemented as a kernel module, this layer of indirection makes the file system portable, since it only needs to communicate with the FUSE module
    * By using a file system abstraction API like `afero.Fs`, we can separate the

FUSE implementation from the actual file system structure, making it unit testable and making it possible to add caching in user space (code snippet from https://github.com/pojntfx/stfs/blob/main/pkg/fs/file.go)
    * It is possible to map any `afero.Fs` to a FUSE backend, so it would be possible to switch between different file system backends without having to write FUSE-specific (code snippet from https://github.com/JakWai01/sile-fystem/blob/main/pkg/filesystem/fs.go)
  - Pull-Based Memory Synchronization with NBD
    * The protocol can run over WAN, but is really mostly meant for LAN or localhost usage
    * It has two phases: Handshake and transmission
    * There are two actors in the protocol: One or multiple clients, the server and the virtual concept of an export
    * When the client connects to the server, the server sends a greeting message with the server's flags
    * The client responds with its own flags and an export name (a single NBD server can expose multiple devices) to use
    * The server sends the export's size and other metadata, after which the client acknowledges the received data and the handshake is complete
    * After the handshake, the client and server start exchanging commands and replies
    * A command can be any of the basic operations needed to access a block device, e.g. read, write or flush
    * Depending on the command, it can also contain data (such as the chunk to be written), offsets, lengths and more
    * Replies can contain an error, success value or data depending on the reply's type
  - Implementation of Mount API and Direct Mount API
    * This also makes it possible to implement smart pull/push strategies instead of simply directly writing to/from the network ("managed mounts")
    * The server and client are connected by creating a connected UNIX socket pair (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/path_direct.go#L59-L62)
    * By building on this basic direct mount, we can add a file (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/file_direct.go) and slice (code snippet from https://github.com/pojntfx/r3map/blob/main/pkg/mount/slice_direct.go) mount API, which allows for easy usage and integration with `sync`/`msync` respectively
    * The protocol as such is stateless, as there is only a simple remote read/write interface (add state machine and sequence diagram here)
    * One additional layer that needs to be implemented however is proper chunking sup-

port

- Creation and Implementation of Managed Mount API
  * While these systems already allow for some optimizations over simply using the NBD protocol over WAN, they still mean that chunks will only be fetched as they are being needed, which means that there still is a guaranteed minimum downtime
  * In order to improve on this, a more advanced API (the managed mount API) was created
  * For the managed mount API, both paradigms were implemented
  * The managed mount API is primarily intended as an API for reading from a remote resource and then syncing changes back to it however, not migrating a resource between two hosts
  * The migration API (will be discussed later) is a more well-optimized version for this use case
  * Using this `Puller` interface, it is possible to implement a read-only managed mount
- Implementation of Managed Mount API
  * In order to allow for a `Sync()` API, e.g. the `msync` on the `mmap`ed file must happen before `Sync()` is called on the syncer
  * While the managed mounts API mostly works, there are some issues with it being implemented in Go
  * Unlike the managed mounts API, the system proposed in Remote Regions is mostly intended for private use cases with a limited number of hosts and in LAN, with low-RTT connections

- Results
  - Results of Pull-Based Memory Synchronization Implementation
    * One major benefit of using `userfaultfd` and the pull method is that it significantly simplifies the backend of the entire system to a `io.ReaderAt` (code snippet from https://pkg.go.dev/io#ReaderAt).
    * This means we can use almost any `io.ReaderAt` as a backend for a `userfaultfd-go` registered object.
    * The system can make an assumption that access will always be aligned to 4 KB chunks/the system page size, so the server can assume a chunk size based on that.
    * The first example of practical implementation returns a random pattern in the backend (code snippet from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-abc/main.go).
    * Notably, a file is a valid `io.ReaderAt`, so we can also use a file as the backend directly, allowing a (remote) file to essentially be mounted into memory (code snippet

from https://github.com/loopholelabs/userfaultfd-go/blob/master/cmd/userfaultfd-go-example-file/main.go).
    * Similarly, we can map a remote object from S3 into memory, accessing only required chunks (via HTTP range requests for S3) (code snippet from https://github.com/loopholelabs/userfaul go/blob/master/cmd/userfaultfd-go-example-s3/main.go).
    * Hence, using `userfaultfd`, we are able to map almost any object into memory.
    * Benchmark: Sensitivity of `userfaultfd` to network latency and throughput.
- Performance Benchmarks and Implementation Examples
    * Benchmark: Performance of different Go hashing algorithms for detecting writes
    * Benchmark: Hashing the chunks individually vs. hashing the entire file
    * Benchmark: Throughput of this custom synchronization protocol vs. rsync (which hashes entire files)
    * For this protocol specifically, we send the changed file's name as the first message when starting the synchronization, but a simple multiplexing system could easily be implemented by sending a file ID with each message
    * As a result, if there are many possible destinations to migrate state too, a star-based architecture with a central forwarding hub can be used
- Examples and Limitations
    * For example, STFS used a tape drive as the backend, which is not random access, but instead append-only and linear (https://github.com/pojntfx/stfs/blob/main/pkg/operations/update.g
    * By using an on-disk index and access optimizations, the resulting file system was still performant enough to be used, and supported almost all features required for the average user
    * FUSE does however also have downsides
    * It operates in user space, which means that it needs to do context switching
    * Some advanced features aren't available for a FUSE
    * The overhead of FUSE (and implementing a completely custom file system) for synchronizing memory is still significant
- Pull-Based Memory Synchronization with NBD
    * NBD is however limited in some respects; the maximum message size is 32 MB, but the maximum block/chunk size supported by the kernel is just 4096 KB, making it a suboptimal protocol to run over WAN, esp. in high latency scenarios
    * The protocol also allows for listing exports, making it possible to e.g. list multiple memory regions on a single server
    * NBD is an older protocol with multiple different handshake versions and legacy features
    * Since the purpose of NBD in this use case is minimal and both the server and the client

are typically controlled, it makes sense to only implement the latest recommended versions and the baseline feature set

  - Observations from Mount Implementation
    * Using the `mmap`/slice approach has a few benefits
    * Alternatively, it would also be possible to format the server's backend or the block device using standard file system tools
    * This allows for handling multiple remote regions using a single server, and thus saving on initialization time and overhead
  - Benchmarking and Observations from the Managed Mount API
    * Benchmark: Local vs. remote chunking
    * The benefit of post-copy migration is that it does not require re-transmitting dirty chunks to the destination before the maximum tolerable downtime is reached
    * The big drawback of post-copy migration is that it can result in longer migration times, because the chunks need to be fetched from the network on-demand, which is very latency/RTT-sensitive
  - Benchmarking and System Testing
    * Benchmark: Parallelizing startups and pulling n MBs as the device starts
    * Using this benchmark and simple interface also makes the entire system very testable
    * In the tests, a memory reader or file can be used as the local or remote `ReaderWriterAt`s

- Discussion
  - Discussion on Results
    * This approach is very clean and has comparatively little overhead, but it also has significant architecture-related limitations that restrict its uses.
    * The first major problem lies in being able only to catch page faults. Hence, we can only ever respond the first time a chunk of memory gets accessed; all future requests will return the memory directly from RAM on the destination host.
    * This limitation prevents us from using this approach for remote resources that update over time.
    * It also prevents the use of this technique for items that might have concurrent writers/shared resources, as there would be no way to update a conflicting section.
    * This effectively restricts this system's use to a read-only "mount" of a remote resource, rather than real synchronization.
    * It also hinders the pulling of chunks before they are being accessed without layers of indirection.

    - The `userfaultfd` API socket is also synchronous, meaning that each chunk needs to be sent one after another. This makes it highly vulnerable to long RTTs.
    - Each future request will have an initial latency at minimum the RTT to the remote source (without caching).
    - The most significant problem is this: taking all these drawbacks into account, under real-life use cases, the maximum throughput—even with a local process handling page faults on a modern computer—is only around ~50MB/s.
    - In summary, while this approach appears interesting and very idiomatic to Go, for most data—especially larger datasets and in high-latency scenarios/in WAN—a more effective solution is required.
  - Review of the Hashing Strategy
    - Polling on its own has a lot of downsides, like it adding a guaranteed minimum latency by virtue of having to wait for the next polling cycle
    - This negatively impacts a maximum allowed downtime scenario, where the overhead of polling can make or break a system
    - Hashing the entire file is also a naturally IO- and CPU-intensive process because the entire file needs to be read at some point
    - Still, polling & hashing is probably the only reliable way of detecting changes to a file
    - When picking algorithms for this hashing process, the most important metric to consider is the throughput with which it can compute hashes, as well as the change of collisions
    - Similarly to `userfaultfd`, this system also has limitations
    - While `userfaultfd` was only able to catch reads, this system is only able to catch writes to the file
    - Essentially this system is write-only, and it is very inefficient to add hosts to the network later on
    - The static topology of this approach can be used to only ever require hashing on one of the destinations and the source instead of all of them
    - This way, we only need to push the changes to one component (the hub), instead of having to push them to each destination on their own
    - The hub simply forwards the messages to all the other destinations
  - Analysis of the FUSE-based Approach
    - If possible, the optimal solution would be to not expose a full file system to track changes, but rather a single file
    - As a result of this, the significant implementation overhead of such a file system led to it not being chosen
  - Pull-Based Memory Synchronization with NBD

* The baseline feature set requires no TLS, the latest “fixed newstyle” handshake, the ability to list and choose an export, as well as the read, write and disc(onnect) commands and replies
* As such, the protocol is very simple to implement
* With this simplicity however also come some drawbacks: NBD is less suited for use cases where the backing device behaves very differently from a random-access store device, like for example a tape drive, since it is not possible to work with high-level abstractions such as files or directories
* This is, for the narrow memory synchronization use case, however more of a feature than a bug
* Due to the lack of pre-existing libraries, a new pure Go NBD library was implemented
* This library does not rely on CGo/a pre-existing C library, meaning that a lot of context switching can be skipped
* The backend interface for `go-nbd` is very simple and only requires four methods: `ReadAt`, `WriteAt`, `Size` and `Sync`
* A good example backend that maps well to a block device is the file backend
* The key difference here to the way backends were designed in `userfaultfd-go` is that they can also handle writes
* `go-nbd` exposes a `Handle` function to support multiple users without depending on a specific transport layer
* This means that systems that are peer-to-peer (e.g. WebRTC), and thus don’t provide a TCP-style `accept` syscall can still be used easily
* It also allows for easily hosting NBD and other services on the same TCP socket
* The server encodes/decodes messages with the `binary` package
* To make it easier to parse, the headers and other structured messages are modeled as Go structs
* The handshake is implemented using a simple for loop, which either returns on error or breaks
* The actual transmission phase is done similarly, by reading in a header, switching on the message type and reading/sending the relevant data/reply
* The server is completely in user space, there are no kernel components involved here
* The NBD client however is implemented by using the kernel NBD client
* In order to use it, one needs to find a free NBD device first
* NBD devices are pre-created by the NBD kernel module and more can be specified with the `nbds_max` parameter
* In order to find a free one, we can either specify it directly, or check whether we can find a NBD device with zero size in `sysfs`
* The handshake for the NBD client is negotiated in user space by the Go program

    * The `DO_IT` syscall never returns, meaning that an external system must be used to detect whether the device is actually ready
    * Two ways of detecting whether the device is ready: By polling `sysfs` for the size parameter, or by using `udev`
    * `udev` manages devices in Linux
    * When a device becomes available, the kernel sends a `udev` event, which we can subscribe to and use as a reliable and idiomatic way of waiting for the ready state
    * In reality however, polling `sysfs` directly can be faster than subscribing to the `udev` event, so we give the user the option to switch between both options
    * When `open`ing the block device that the client has connected to, usually the kernel does provide a caching mechanism and thus requires `sync` to flush changes
    * By using `O_DIRECT` however, it is possible to skip the kernel caching layer and write all changes directly to the NBD client/server
    * This is particularly useful if both the client and server are on the local system, and if the amount of time spent on `sync`ing should be as small as possible
    * It does however require reads and writes on the device node to be aligned to the system's page size, which is possible to implement with a client-side chunking system but does require application-specific code
    * NBD is a battle-tested solution for this with fairly good performance, but in the future a more lean implemenation called `ublk` could also be used
    * The only way of implementing it without too much overhead however is CGo, which comes with its own overhead
    * It also requires calling Go closures from C code, which is complicated
    * Implementing closures is possible by using the `userdata` parameter in the CUSE C API
    * To fully use it, it needs to first resolve a Go callback in C, and then call it with a pointer to the method's struct in user data, effectively allowing for the use of closures
    * Even with this however, it is hard to implement even a simple backend, and the CGo overhead is a significant drawback
    * The last alternative to NBD devices would be to extend the kernel with a new construct that allows for essentially a virtual file to be `mmap`ed, not a block device
    * This could use a custom protocol that is optimized for this use case instead of a full block device
    * Because of the extensive setup required to implement such a system however, and the possibility of `ublk` providing a performant alternative in the future, going forward with NBD was chosen for now
  - Potential Limitations and Future Improvements
    * Using a proper file system however does introduce both storage overhead and com-

plexity, which is why e.g. the FUSE approach was not chosen
    * For scenarios where the RTT between the backend and server is large, it might make sense to use a much larger chunk size for the actual networked transfers
    * Even if there are no constraints on chunking on the backend side (e.g. when a file is used as the backend), it might make sense to limit the maximum supported message size between the client and server to prevent DoS attacks by forcing the backend to allocate large chunks of memory to satisfy requests, which requires a chunking system to work
    * In order to implement the chunking system, we can use an abstraction layer that allows us to create a pipeline of readers/writers - the `ReadWriterAt`, combining an `io.ReaderAt` and a `io.WriterAt`
    * This way, we can forward the `Size` and `Sync` syscalls directly to the underlying backend, but wrap a backend's `ReadAt` and `WriteAt` methods in a pipeline of other `ReadWriterAt`s
  - Analyses and Potential Improvements of Managed Mount API
    * Many applications commonly access certain bits of data first
    * If a resource should be available locally as quickly as possible, then using the correct pull heuristic can help a lot
    * The combination of the `SyncedReadWriterAt` and the `Puller` component implements the pull post-copy system in a modular and testable way
    * Unlike the usual way of only fetching chunks when they are available however, this system also allows fetching them pre-emptively, gaining some benefits of pre-copy migration, too
  - Potential Improvements and Alternatives for Managed Mount API
    * While the current API is Go-specific, it could also be exposed through a different interface to make it usable in Go
    * In the future, using a language without a GC (such as Rust) could provide a good alternative
    * It is also not designed to be used for potential migration scenarios, which the modular approach of r3map allows for

- Conclusion
  - Looking back at all synchronization options and comparing ease of implementation, CPU load and network traffic between them
  - Summary of the different approaches, and how the new solutions might make it possible to use memory as the universal access format
  - Further research recommendations (e.g. `ublk`)