
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	7

1 Unsorted Research Questions

2 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
- Pull-Based Memory Synchronization with `userfaultfd`
 - Plain language description of `userfaultfd` (what are page faults)
 - Exploring an alternative method by handling page faults using signals
 - Handlers and registration
 - History of `userfaultfd`
 - Allocating the shared region
 - Maximum shared region size is limited by available physical memory
 - Transferring handler sockets between processes
 - Implementing `userfaultfd` bindings in Go
 - Example usages of `userfaultfd` in Go (byte slice, file, S3 object)
 - Implications of not being able to catch writes to the region (its read-only)
 - Design of a `userfaultfd` backend (`io.ReaderAt`)
 - Limitations: ~50MB/s of throughput
 - Limitations of only being able to catch the first page fault (no way of updating the region)
 - Implications of not being able to pull chunks before they are being accessed
 - Limitations of only being able to pull chunks synchronously
 - Benefits of minimal registration and latency overhead
 - Benchmark: Sensitivity of `userfaultfd` to network latency and throughput
- Push-Based Memory Synchronization with `mmap` and Hashing
 - Plain language description of this approach (mapping a file into memory, then syncing the file)
 - Paging and swap in Linux
 - Introduction to `mmap` to map a file into a memory region
 - `MAP_SHARED` for writing changes back from the memory region to a file
 - Caching with `mmap`, why `O_DIRECT` doesn't work and what the role of `msync` is

- Detecting writes to a file with `inotify` and why this does not work for `mmap`
- Hashing (chunks of) the backing file in order to detect writes on a periodic basis
- Benchmark: Performance of different Go hashing algorithms for detecting writes
- Effects on maximum acceptable downtime due to CPU-bound limitations in having to calculate hashes and polling
- Introduction to delta synchronization protocols like `rsync`
- Implementation of a custom delta synchronization protocol with chunk support
- Multiplexing different synchronization streams in the protocol
- Benchmark: Throughput of this custom synchronization protocol vs. `rsync` (which hashes entire files)
- Using a central forwarding hub for star-based architectures with multiple destinations
- Limitations of only being able to catch writes, not reads to the region (its write-only, can't add hosts later on)
- Push-Pull Memory Synchronization with a FUSE
 - Plain language description of this approach (mapping a file into memory, catching reads/writes to/from the file with a custom filesystem)
 - Methods for creating a new, custom file system (FUSE vs. in the kernel)
 - STFS shows that it is possible to create file systems backed by complex data stores, e.g. a tape/non-random access stores
 - Is not the best option for implementing this due to significant overhead and us only needing a single file to map
- Pull-Based Memory Synchronization with NBD
 - Plain language description of this approach (mapping a block device into memory, catching reads/writes with a NBD server)
 - Why NBD is the better choice compared to FUSE (much less complex interface)
 - Overview of the NBD protocol
 - Phases, actors and messages in the NBD protocol (negotiation, transmission)
 - Minimal viable NBD protocol needs, and why only this minimal set is implemented
 - Listing exports
 - Limitations of NBD, esp. the kernel client and message sizes
 - Reduced flexibility of NBD compared to FUSE (can still be used for e.g. linear media, but will offer fewer interfaces for optimization)
 - Server backend interface design
 - File backend example
 - How the servers handling multiple users/connections
 - Server handling in the NBD protocol implementation

- Using the kernel NBD client without CGO/with ioctls
- Finding an unused NBD device using `sysfs`
- Benchmark: `nbd` kernel module quirks and how to detect whether a NBD device is open (polling `sysfs` vs. `udev`)
- Caching mechanisms and limitations (aligned reads) when opening the block device (`O_DIRECT`)
- Future outlook: Using `ublk` instead of NBD, allowing for potentially much faster concurrent access thanks to `io_uring`
- Why using BUSE to implement a NBD server would be possible but unrealistic (library & docs situation, CGo)
- `go-buse` as a preview of how such a CGo implementation could still work
- Alternatively implementing a new file system entirely in the kernel, only exposing a single file/block device to `mmap` and optimizing the user space protocol for this
- Push-Pull Memory Synchronization with Mounts
 - Plain language description of this approach (like NBD, but starting the client and server locally, then connecting the *server's* backend to a backend)
 - Benefits: Can use a secure wire protocol and more complex/abstract backends
 - Mounting the block device as a path vs. file vs. slice: Benefits of `mmap` (concurrent reads/writes)
 - Alternative approach: Formatting the block device as e.g. EXT4, then mounting the filesystem, and `mmap`ing a file on the file system (allows syncing multiple regions with a single file system, but has FS overhead)
 - Mount protocol actors, phases and state machine
 - Chunking system for non-aligned reads/writes (arbitrary rwat and chunked rwat)
 - Benchmark: Local vs. remote chunking
 - Optimizing the mount process with the Managed Mount interface
 - Pre- and post-copy systems and why we should combine them (see Optimizing Virtual Machine Live Migration without Shared Storage in Hybrid Clouds)
 - Asynchronous background push system interface and how edge cases (like accessing dirty chunks as they are being pulled or being written to) are handled
 - Preemptive background pulls interface
 - Syncer interface
 - Benchmark: Parallelizing startups and pulling n MBs as the device starts
 - Using a pull heuristic function to optimize which chunks should be scheduled to be pulled first
 - Internal rwat pipeline (create a graphic) for direct mounts vs. managed mounts
 - Unit testing the rwats

- Comparing this mount API to other existing remote memory access APIs, e.g. “Remote Regions” (“Remote regions: a simple abstraction for remote memory”)
- Complexities when `mmap`ing a region in Go as the GC halts the entire world to collect garbage, but that also stops the NBD server in the same process that tries to satisfy the region being scanned
- Potentially using Rust for this component to cut down on memory complexities and GC latencies
- Pull-Based Memory Synchronization with Migrations
 - Plain language description of this approach (like NBD, but two phases to start the device and pull, then only flush the latest changes to minimize downtime)
 - Inspired by live VM migration, where changes are continuously being pulled to the destination node until a % has been reached, after which the VM is migrated
 - Migration protocol actors (seeders, leechers etc.), phases and state machine
 - How the migration API is completely independent of a transport layer
 - Switching from the seeder to the leecher state
 - Using preemptive pulls and pull heuristics to optimize just like for the mounts
 - Lifecycle of the migration API and why lockable rwats are required
 - How a successful migration causes the `Seeder` to exit
 - The role of maximum acceptable downtime
 - The role of `Track()`, concurrent access and consistency guarantees vs. mounts (where the source must not change)
 - When to best `Finalize()` a migration and how analyzing app usage patterns could help (A Framework for Task-Guided Virtual Machine Live Migration, Reducing Virtual Machine Live Migration Overhead via Workload Analysis)
 - Benchmark: Maximum acceptable downtime for a migration scenario with the Managed Mount API vs the Migration API
- Optimizing Mounts and Migrations
 - Encryption of memory regions and the wire protocol
 - Authentication of the protocol
 - DoS vulnerabilities in the NBD protocol (large message sizes; not meant for the public internet) and why the indirection of client & server on each node is needed
 - Mitigating DoS vulnerabilities in the `ReadAt/WriteAt` RPCs with `maxChunkSize` and/or client-side chunking
 - Critical `Finalizing` state in the migration API and how it could be remedied
 - How network outages are handled in the mount and migration API
 - Analyzing the file and memory backend implementations

- Analyzing the directory backend
- Analyzing the dudirekta, gRPC and fRPC backends
- Benchmark: Latency till first n chunks *and* throughput for dudirekta, gRPC and fRPC backends (how they are affected by having/not having connection polling and/or concurrent RPCs)
- Benchmark: Effect of tuning the amount of push/pull workers in high-latency scenarios
- Analyzing the Redis backend
- Analyzing the S3 backend
- Analyzing the Cassandra backend
- Benchmark: Latency and throughput of all benchmarks on localhost and in a realistic latency and throughput scenario
- Effects of slow disks/memory on local backends, and why direct mounts can outperform managed mounts in tests on localhosts
- Using P2P vs. star architectures for mounts and migrations
- Looking back at all options and comparing ease of implementation, CPU load and network traffic between them
- Case Studies
 - [ram-dl](#) as an example of using the direct mount API for extending system memory
 - [tapisk](#) as an example of using the managed mount API for a file system with very high latencies, linear access and asynchronous to/from fast local memory vs. STFS
 - Migration app state (e.g. TODO list) between two hosts in a universal (underlying memory) manner
 - Mounting remote file systems as managed mounts and combining the benefits of traditional FUSE mounts (e.g. s3-fuse) with Google Drive-style synchronization
 - Using managed mounts for remote SQLite databases without having to download it first
 - Streaming video formats like e.g. MP4 that don't support streaming due to indexes/compression
 - Improving game download speeds by mounting the remote assets with managed mounts, using a pull heuristic that defines typical access patterns (like which levels are accessed first), making any game immediately playable without changes
 - Executing remote binaries or scripts that don't need to be scanned first without having to fully download them
- Conclusion
 - 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 Content

- 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 occurred, 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 pages 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 pages 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`d in order for them to be written to disks, `mmap`ed regions need to be `msync`d 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 continuously 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/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 `mmaped` 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/poijntfx/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-fsystem/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/poijntfx/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 `mmaped` 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
- How the server encodes/decodes messages
- Message type modelling
- Using the kernel NBD client without CGO/with `ioctl`s
- Finding an unused NBD device using `sysfs`
- Benchmark: `nbd` kernel module quirks and how to detect whether a NBD device is open (polling `sysfs` vs. `udev`)
- Caching mechanisms and limitations (aligned reads) when opening the block device (`O_DIRECT`)
- Future outlook: Using `ublk` instead of NBD, allowing for potentially much faster concurrent access thanks to `io_uring`
- Why using BUSE to implement a NBD server would be possible but unrealistic (library & docs situation, CGo)
- `go-buse` as a preview of how such a CGo implementation could still work
- Alternatively implementing a new file system entirely in the kernel, only exposing a single file/block device to `mmap` and optimizing the user space protocol for this