# 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 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)
  - Limiations: ~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 manged 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 scrips 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 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