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**
  - How does memory in Linux work? Paging, swap etc.
  - Examining Linux's memory management and relevant APIs
  - An introduction to mmap and how we can use it to map a file into memory/a byte slice, the role of `msync`
  - Concept: `mmap` a memory region with MMAP_SHARED to track changes in a file
  - Use cases for memory region synchronization (rough overview, esp. what is currently popular in the industry)
- **Synchronization Strategies**
  - **Implementing push-based memory sync by tracking changes to a `mmap`ed slice with polled hashing of individual chunks**, why we can't use `inotify`, and the CPU-bound limitations of this approach
    - Detecting file changes: inotify; but `mmap` does not generate WRITE events
    - Comparing hashes of local and remote `mmap`ed regions
    - Evaluation of hashing algorithms
    - Introduction to delta synchronization (e.g., rsync)
    - Custom protocol for delta synchronization
    - CPU-bound bottlenecks
    - Multiplexing synchronization streams (central forwarding hub etc.)
    - Comparing hashes of local and remote `mmap`ed regions
    - O_DIRECT vs `mmap`, RAM vs Swap etc.
    - Function of `msync`, lack of `O_DIRECT` (and why it lacks this/doesn't support it)
    - All clients need to be connected from the beginning; strong risk of inconsistencies
    - Performance of different hashing algorithms for detecting changes to a `mmap`ed region
  - **Implementing pull-based memory sync with `userfaultfd`; and implementation and throughput limitations**
    - Introduction to userfaultfd
    - Implementing userfaultfd handlers and registration in Go
    - The function of `msync`
    - Transferring sockets between processes
    - Performance assessment of this approach

    * Examples of handler and registration interfaces (byte slice, file, S3 object)
    * Performance assessment of this approach
    * Explanation of userfaultfd and its implementation
    * Description of userfaultfd handlers and registration in Go
    * I/O-bound bottlenecks
    * No option to pre-emptively pull data
    * userfaultfd is read-only
    * userfaultfd can only be used to fetch the first (missing) chunk, not subsequent ones
    * userfaultfd is limited to ~50MB/s of throughput
    * Biggest benefit of userfaultfd: It has minimal registration overhead & latency
    * userfaultfd's interface is just an io.ReaderAt, making it extremely simple to use
- **Implementing push-pull based memory sync with a FUSE**; limitations and complexity (citing STFS)
    * Intercepting writes to the `mmap`ed region using a custom filesystem
    * Mounting the filesystem
    * Exploring methods for creating a new, custom Linux filesystem: FUSE, in the Kernel
    * Taking a look at STFS for how to implement this; tape-specific optimizations were possible, so we could also do this for files I presume
    * Why this approach is suboptimal (we need just one file, not a full filesystem)
- **Implementing push-pull based memory sync with NBD; implementation of go-nbd**
    * Detailed analysis of the NBD protocol (client & server)
    * Implementing the client and server in Go based on the protocol
    * Server backend interface and example implementations
    * Backends can use custom indexes to map linear media (e.g. tape drives) into memory by mapping the block device offset to a real, append-only record number and swapping it out for a new one when things get overwritten in the block device vs. FUSE, where its much more complicated/needs to be a full filesystem
    * Caching mechanism (mounting the fd with O_DIRECT vs. not)
    * Performance assessment of this approach
    * Project scope (why only the minimal protocol was implemented)
    * NBD protocol overview and limitations
    * NBD protocol phases
    * Minimal viable NBD protocol needs
    * Go NBD server implementation: Multiple clients, error handling
    * Go NBD client and server implementation: The kernel's NBD client, CGo for `ioctl` numbers
    * Finding unused NBD devices, detecting client availability (polling sysfs vs. udev; add

benchmarks)
    * go-nbd pluggable backend API design/interface
    * go-nbd project scope & keeping it maintainable, esp. vs other NBD implementations
    * Using `ublk` instead of NBD in the future; should be much faster than NBD for concurrency/random-access as it supports
    * Using a BUSE as a NBD server (library/CGo limitations)
    * Extending the kernel with a new resource or a filesystem like FUSE, but actually in the kernel with a more efficient user-space protocol optimized for random access
- **Using NBD directly as a mount-based sync system with the direct mount API**; limitations with latency etc., and improvements with background pulls and pushes, different backends etc., the mount wire protocol, pull heuristics
    * Use cases
    * Path vs. file vs. slice mounts/migrations
    * Limitations and benefits of `mmap` for accessing a mount vs. a file (concurrent reads/writes etc.)
    * Instead of `mmap`ing the block device, formatting the block device with e.g. EXT4 and then `mmap`ing a file on through that (would allow synchronization of multiple separate memory regions, e.g. multiple app states, over the network)
    * Discussion on mount protocol actors, phases and state machine
    * Managed mount protocol actors, phases, sequence and state machine
    * The asynchronous background push method (for mounts); how chunks are marked as dirty when they are being written to before the download has finished completely
    * Chunking system/non-aligned reads and writes, checking for correct chunking behavior
    * Making it unit testable; rwat pipeline
    * Local vs. remote chunking
    * Pull priority function/heuristic: Benchmarks when accessing from end of file to start vs. other way around, latency vs. throughput changes with/without heuristic, using LLMs etc. to analyze access patterns with `pullWorkers: 0` and then generating an automatic pull heuristic
    * Usage of the r3map's API vs. e.g. "Remote regions" paper
    * Using Rust for a future implementation; esp. to prevent `mmap` blocks from being scanned by the garbage collector
- **Taking inspiration from VM live migration and adding a migration system** for memory sync, two-phase commit, the sync wire protocol, minimum acceptable downtime as the metric to optimize for
    * Use cases

    * Discussion on migration protocol actors, phases and state machine
    * Migration protocol actors, phases and state machine
    * Examination of P2P vs. relay systems/hub and spoke systems
    * Preemptive pulls and parallelized startups (n MB saved)
    * Background pulling system and interface (rwat), % of availability
    * Migration API lifecycle & the role of lockable rwats
    * Minimum acceptable downtime
    * Concurrent access/consistency guarantees for mounts vs. migrations etc. - `Track()`, why we can't modify a mount's source
    * When to best `Finalize()` a migration; analyzing app usage patterns?
  - **Optimizing the Mount and Migration Implementations**
    * Encryption of regions and the wire protocol, authn, DoS vulnerabilities without max size
    * Criticality/critical phases in protocols (e.g. recovering from a network outage in mount vs. migration, finalization step can't be aborted etc.)
    * Performance tuning parameters (chunk size, push/pull workers)
    * Backend implementations: File, memory, directory, dudirekta, gRPC, fRPC, Redis, S3, Cassandra (a section for each)
    * Transport layers: Dudirekta, gRPC, fRPC (esp. benefits and problems with concurrent RPCs, connection pooling like with dRPC, benchmarks with latency and throughput etc.)
    * Usage of QUIC, UDP and other protocols for skipping on RTT to improve minimal latency
    * Examination of different transport layers and their implications on performance and concurrency
    * Discuss potential effects of high latency, slow local disks or RAM on different pull methods
    * Effects of slow local disks or RAM on pull methods
    * Comparing options in terms of ease of implementation, CPU load, and network traffic
    * Effects of high latency on different pull methods (esp. direct vs. managed)
    * Performance tuning parameters (chunk size, push/pull workers)
    * P2P vs. relay systems/hub and spoke systems

- **Case Studies**
  - Mount backend API vs. seeder API
  - Use cases: Direct Mount vs. Managed Mount vs. Migration
  - Use cases, case studies and comparison of approaches, finding the one that is right for each one, and showing an implementation for each, benchmarks, performance tuning

  - Identifying the optimal solution for specific use cases: data change frequency, kernel/OS compatibility, etc.
  - Tapisk as an example of using the mount APIs for a filesystem, esp. one with very low read-/write speeds and high latency; esp. with support for writebacks in the future, and comparing this to STFS which was a FUSE instead of a block device
  - `ram-dl` as an example of using a remote backend to provide more RAM/Swap
  - Migrating app state (e.g. a TODO list) between two hosts in a universal (byte-slice/by using the underlying memory) manner, integration with existing app migration systems
  - Mounting remote filesystems and combining the benefits of traditional FUSE-based mounts (e.g. s3-fuse) with Google Drive/Nextcloud-style synchronization
  - Using mounts for SQLite etc. database access without having to use range requests
  - Streaming video using formats that usually don't support streaming, e.g. MP4, where an index is required
  - Improving game download speeds by mounting the remote assets with managed mounts, using a pull heuristic that defines typical access patterns, e.g. by levels (making the game immediately playable)
  - Executing remote binaries or scripts that don't need to be scanned first without fully downloading them

- **Conclusion**
  - Conclusion with a summary of the different approaches, further research recommendations