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 Rough Structure

- Abstract: A comparative analysis and implementation of various methods for synchronizing Linux memory options over a network
- Introduction
  - Examining Linux's memory management and relevant APIs
  - Use cases for memory region synchronization
- Option 1: Handling page faults in userspace with `userfaultfd`
  - Introduction to userfaultfd
  - Implementing userfaultfd handlers and registration in Go
  - Transferring sockets between processes
  - Examples of handler and registration interfaces (byte slice, file, S3 object)
  - Performance assessment of this approach
- Option 2: Utilizing `mmap` for change notifications
  - Concept: `mmap` a memory region with MMAP_SHARED to track changes in a file
  - Method 1 for detecting file changes: inotify
  - Limitations: `mmap` does not generate WRITE events
- Option 3: Hash-based change detection
  - 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
  - Multiplexing synchronization streams
  - The function of `msync`
  - Performance assessment of this approach
- Option 4: Detecting changes with a custom filesystem implementation
  - Intercepting writes to the `mmap`ed region using a custom filesystem
  - Exploring methods for creating a new, custom Linux filesystem
    - In the kernel
    - NBD
    - CUSE
    - BUSE
    - FUSE
    - Upcoming options (ublk, etc.)

  - 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
  - Asynchronous writeback protocol and caching mechanism
  - Performance assessment of this approach
- Summary:
  - Comparing options in terms of ease of implementation, CPU load, and network traffic
  - Identifying the optimal solution for specific use cases: data change frequency, kernel/OS compatibility, etc.

2 Sections/Research Questions/Ideas Brainstorming

- Usecases: Direct Mount vs. Managed Mount vs. Migration
- Effects of high latency on different pull methods (esp. direct vs. managed)
- Effects of slow local disks or RAM on pull methods
- 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
- Mount backend API vs. seeder API
- Preemptive pulls and parallelized startups (n MB saved)
- Background pulling system and interface (rwat), % of availability
- Chunking system/non-aligned reads and writes, checking for correct chunking behavior
- Local vs. remote chunking
- Backend implementations, performance and usecases: File, memory, directory, dudirekta, gRPC, fRPC, Redis, S3, Cassandra
- Transport layers: Dudirekta, gRPC, fRPC (esp. benefits and problems with concurrent RPCs, connection pooling like with dRPC, benchmarks with latency and throughput etc.)
- 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
- Migration API lifecycle & lockable rwats

- Path vs. file vs. slice mounts/migrations
- Migration protocol actors, phases and state machine
- Managed mount protocol actors, phases, sequence and state machine
- Performance tuning parameters (chunk size, push/pull workers)
- P2P vs. relay systems/hub and spoke systems
- 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
- Performance of different hashing algorithms for detecting changes to a `mmap`ed region
- Usage of the r3map's API vs. e.g. "Remote regions" paper
- Tapisk as an example of using the mount APIs for a filesystem; esp. with support for writebacks in the future, and comparing this to STFS
- `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
- Minimum acceptable downtime
- Concurrent access/consistency guarantees for mounts vs. migrations etc. - `Track()`, why we can't modify a mount's source
- Usage of QUIC, UDP and other protocols for skipping on RTT to improve minimal latency
- Limitations and benefits of `mmap` for accessing a mount vs. a file (concurrent reads/writes etc.)
- Criticality in protocols (e.g. recovering from a network outage in mount vs. migration, finalization step can't be aborted etc.)
- Encryption of regions and the wire protocol, authn, DoS vulnerabilities
- Integration with existing app migration systems
- When to best `Finalize()` a migration; analyzing app usage patterns?
- How does Linux actually manage memory, O_DIRECT vs `mmap`, RAM vs Swap etc.
- 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
- 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

3 Alternative Outline

1. Abstract

- A comparative analysis and implementation of various methods for synchronizing Linux memory options over a network

2. Introduction
    1. Background: Examining Linux's memory management and relevant APIs
    2. Purpose: Use cases for memory region synchronization (Direct Mount, Managed Mount, Migration)
        - Discuss potential effects of high latency, slow local disks or RAM on different pull methods
3. Methodologies
    1. Option 1: Handling page faults in userspace with `userfaultfd`
        1. Explanation of userfaultfd and its implementation
        2. Description of userfaultfd handlers and registration in Go
        3. The process of transferring sockets between processes
        4. Examples of handler and registration interfaces
        5. Performance assessment of this approach, with focus on pull methods and effects of system constraints
    2. Option 2: Utilizing `mmap` for change notifications
        1. Understanding mmap and its application
        2. Detecting file changes using inotify and its limitations
        3. Performance evaluation of this approach, with focus on preemptive pulls and parallelized startups
    3. Option 3: Hash-based change detection
        1. Comparing hashes of local and remote `mmap`ed regions
        2. Evaluation of different hashing algorithms
        3. Introduction to delta synchronization and the role of rsync
        4. Custom protocol for delta synchronization
        5. Multiplexing synchronization streams and the function of `msync`
        6. Performance assessment of this approach, with an analysis of the effects of various pull priority functions/heuristics
    4. Option 4: Detecting changes with a custom filesystem implementation
        1. Intercepting writes to the `mmap`ed region using a custom filesystem
        2. Understanding methods for creating a new, custom Linux filesystem (in the kernel, NBD, CUSE, BUSE, FUSE, etc.)
        3. Performance assessment of this approach
    5. Option 5: Block device with userspace backend

    1. Detailed analysis of the NBD protocol and its implementation in Go
    2. Server backend interface and example implementations
    3. Asynchronous writeback protocol and caching mechanism
    4. Performance assessment of this approach, with an analysis of Go NBD client and server implementation and the project scope

4. Evaluation and Comparison of Approaches

    1. Comparing options in terms of ease of implementation, CPU load, and network traffic
    2. Identifying the optimal solution for specific use cases: data change frequency, kernel/OS compatibility, etc.
    3. Analysis of Go NBD client and server implementation and the project scope
    4. Examination of different transport layers and their implications on performance and concurrency
    5. Discussion on migration protocol actors, phases and state machine
    6. Performance tuning parameters (chunk size, push/pull workers)
    7. Examination of P2P vs. relay systems/hub and spoke systems

5. Case Studies

    1. Tapisk as an example of using the mount APIs for a filesystem, esp. one with very low read-/write speeds and high latency
    2. `ram-dl` as an example of using a remote backend to provide more RAM/Swap
    3. Migrating app state (e.g. a TODO list) between two hosts in a universal (byte-slice/by using the underlying memory) manner
    4. Mounting remote filesystems and combining the benefits of traditional FUSE-based mounts (e.g. s3-fuse) with Google Drive/Nextcloud-style synchronization
    5. Using mounts for SQLite etc. database access without having to use range requests
    6. Streaming video using formats that usually don't support streaming, e.g. MP4, where an index is required
    7. 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)
    8. Executing remote binaries or scripts that don't need to be scanned first without fully downloading them

6. Conclusion

    1. Summary of findings and optimal solutions for specific use cases
    2. Discussion of minimum acceptable downtime
    3. Future recommendations for research and improvements

4 Story

- Introduction: How does memory in Linux work? Paging, swap etc.
- An introduction to mmap and how we can use it to map a file into memory/a byte slice, the role of `msync`
- 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
- Implementing pull-based memory sync with `userfaultfd`; and implementation and throughput limitations
- Implementing push-pull based memory sync with a FUSE; limitations and complexity (citing STFS)
- Implementing push-pull based memory sync with NBD; implementation of go-nbd
- 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
- 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, case studies and comparison of approaches, finding the one that is right for each one, and showing an implementation for each, benchmarks, performance tuning
- Conclusion with a summary of the different approaches, further research recommendations

5 Revised Structure

1. Introduction
2. Synchronization Strategies
3. Case Studies
4. Conclusion