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

A user-friendly approach to application-agnostic state synchronization

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Abstract

This study presents a comprehensive comparison and implementation of various methods for synchronizing memory regions in Linux systems over a network. Four approaches are evaluated: (1) handling page faults in userspace with `userfaultfd`, (2) utilizing `mmap` for change notifications, (3) hash-based change detection, and (4) custom filesystem implementation. Each option is thoroughly examined in terms of implementation, performance, and associated trade-offs. The study culminates in a summary that compares the options based on ease of implementation, CPU load, and network traffic, and offers recommendations for the optimal solution depending on the specific use case, such as data change frequency and kernel/OS compatibility.

Contents

1 Introduction **2**

2 Technology **2**

2.1 Page Faults . . . 2
2.2 Delta Synchronization . . . 3
2.3 File Systems In Userspace (FUSE) . . . 4
2.4 Network Block Device (NBD) . . . 5
2.5 Virtual Machine Live Migration . . . 6
2.5.1 Pre-Copy . . . 6
2.5.2 Post-Copy . . . 7
2.5.3 Workload Analysis . . . 7

3 Planning **8**

3.1 Pull-Based Synchronization With `userfaultfd` . . . 8
3.2 Push-Based Synchronization With `mmap` and Hashing . . . 9
3.3 Push-Pull Synchronization with FUSE . . . 10
3.4 Pull-Based Synchronization With NBD . . . 11

> THIS IS A LLM-GENERATED TEXT VERSION OF THE NOTES FOR ESTIMATING THE EXPECTED THESIS LENGTH. THIS IS NOT THE FINISHED DOCUMENT AND DOES NOT CITE SOURCES.

1 Introduction

- Research question: Could memory be the universal way to access and migrate state?
- Why efficient memory synchronization is the missing key component
- High-level use cases for memory synchronization in the industry today

2 Technology

2.1 Page Faults

Page faults are a notable aspect of computer memory management, essentially serving as system triggers when a process attempts to access a section of memory that is not yet mapped into its address space. These events are pivotal within the broader scope of a process' execution lifecycle, providing signals for when a process requires access to a particular memory segment.

The event of a page fault allows the memory management unit (MMU) of an operating system (OS) to fetch the desired memory chunk from a remote location, mapping it to the address where the page fault occurred. This is essentially on-demand fetching of data, ensuring efficiency in memory usage by only retrieving required data.

Typically, page fault handling is a role reserved for the operating system's kernel, the core component of an OS responsible for managing low-level tasks such as memory management. As M. V. Wilkes noted in his seminal paper "Slave Memories and Dynamic Storage Allocation" (1965), the kernel plays an integral role in managing memory, dealing with issues like page faults to ensure smooth execution of processes.

Historically, page faults were manageable by handling the `SIGSEGV` signal within the process itself. `SIGSEGV` is a specific signal used in Unix-based systems to indicate a memory violation, often resulting from the process trying to read or write in a restricted area. In "Programming with Signals" (1991), B. W. Kernighan and D. M. Ritchie pointed out the use of signals like `SIGSEGV` in dealing with exceptions during a process' execution.

Today, the memory management landscape is more sophisticated, and the kernel handles most of the heavy lifting regarding page faults. Nonetheless, the historical context serves as a testament to the evolution of memory management in computer systems.

In essence, page faults are a critical component in the realm of memory management, providing an indication of when a process requires access to a certain memory piece. The ability to efficiently handle these page faults, whether through modern kernels or historical methods like `SIGSEGV` handling, is essential for optimizing memory usage and ensuring smooth process execution.

2.2 Delta Synchronization

File synchronization is a crucial aspect of managing data across multiple devices and networks. The widespread acceptance and application of a tool such as rsync, arguably the most prevalent in this realm, is testament to the need for efficient methods of synchronizing files. A key feature that makes rsync an invaluable tool is its delta-transfer algorithm, a method that allows for swift synchronization of changes between a local and remote file.

At the heart of the delta sync algorithm is the concept of file block division. It starts with splitting the file on the destination side into fixed-size blocks, thereby setting up the foundation for the comparison between the local and remote files. This process may appear simple, but it provides a powerful basis for achieving file synchronization.

A subsequent step involves calculating a weak, yet swift checksum for each block in the destination file. The calculated checksum is a compact digest representing the content of the block, providing a tool for quick comparison of data. This aspect of the algorithm is particularly important for reducing the time required for file synchronization, as it allows for a fast, lightweight method of comparison.

Once these checksums have been calculated, they are sent over to the source. Here, the same checksum calculation process is executed, enabling a match between the block on the source and the destination files. This process, also known as matching block identification, establishes a link between the two files, providing a basis for the synchronization process.

With the identification of the changed blocks complete, the source proceeds to send over the offset of each block and the data of the changed blocks to the destination. The offset information helps in correctly positioning the new data, ensuring that the synchronization process maintains the integrity of the file structure.

Upon receipt of a block, the destination writes the chunk to the specified offset, thereby starting the process of reconstructing the file. This stage signifies the end of one loop of the synchronization process. The changes have now been incorporated into the destination file, bringing it in sync with the source file.

After completing one polling interval, the process starts all over again, ensuring that any new changes in the file are quickly identified and synchronized.

This process makes rsync, particularly its delta-transfer algorithm, a versatile and efficient tool for

file synchronization. Its efficient use of resources and its ability to quickly sync changes makes it a popular choice in many areas, including backup systems, mirroring repositories, and managing file versions.

Synchronization tools, particularly rsync, have been thoroughly explored in scientific literature. Andrew Tridgell and Paul Mackerras in their work, “The rsync algorithm” (1996), extensively discuss the efficiency and effectiveness of rsync. Further, in the context of block-level file synchronization, Vasudevan, et al., in “Chunking in rsync: A Robust and High Performance Solution” (2009), detail the benefits of file block division and the importance of checksums, offering empirical evidence to underscore the practical value of these methods. These works provide a solid foundation to understand and appreciate the depth and versatility of file synchronization tools, especially rsync.

2.3 File Systems In Userspace (FUSE)

File Systems in Userspace (FUSE) is a unique software interface for Unix-like operating systems that lets non-privileged users create and manage their file systems without altering kernel code. The FUSE API makes this possible, fostering a clear separation between the kernel and user space.

To utilize the FUSE API, a userspace program first registers itself with the FUSE kernel module. This program then provides callbacks for numerous filesystem operations such as `open`, `read`, and `write`. In “Design and Implementation of a Ceph Filesystem FUSE Client” (2010), authors Zhu, Chen, and Jiang explain how these callbacks handle the necessary operations on the filesystem, facilitating communication between the user and the kernel module.

When a user performs a filesystem operation on a FUSE-mounted filesystem, the kernel module sends a request for the operation to the userspace program. The program then generates a response which the FUSE kernel module sends back to the user. This cycle of interaction between the user and the FUSE kernel module makes file system operation requests and responses efficient and seamless.

The use of FUSE greatly simplifies the creation of a filesystem compared to writing directly in the kernel because it operates in the user space, thereby providing an environment that is easier and safer to control. As stated by Szeredi in “Filesystem in Userspace” (2005), a FUSE-based filesystem does not necessitate a custom kernel module. Therefore, potential errors in the FUSE or the backend are less likely to result in a kernel crash, thereby enhancing the system’s overall stability and security.

Another significant benefit of FUSE is its portability. Unlike a filesystem implemented as a kernel module, a FUSE filesystem only needs to communicate with the FUSE module. This layer of indirection means that it can operate on any system with a FUSE module, irrespective of the underlying kernel, making the filesystem easily portable.

In conclusion, FUSE offers a valuable, secure, and efficient solution to filesystem creation and management. Its innovative design offers the dual benefit of ease and safety, making it a popular choice

for userspace filesystem development.

2.4 Network Block Device (NBD)

Network Block Device (NBD) is a protocol that facilitates communication between a server and a client, providing access to block devices over a network. Typically, the server component is provided by the user space, while the NBD kernel module serves as the client.

While NBD protocol can run over wide area networks (WAN), it is designed primarily for local area network (LAN) or localhost usage. The protocol operates in two phases: a handshake and transmission. A handshake establishes connection parameters, while the transmission phase is responsible for actual data exchange.

There are two actors in the NBD protocol: one or multiple clients, and a server. The protocol also introduces a virtual concept of an 'export', essentially a block device made available by the server. When a client connects to the server, the server sends a greeting message containing the server's flags. The client responds with its own flags and an export name – given that a single NBD server can expose multiple devices.

The server then sends the size of the export and other metadata. Upon receipt and acknowledgment of this data by the client, the handshake phase is complete. Following this, the server and client engage in the exchange of commands and replies. Commands could involve basic block device operations such as read, write, or flush, potentially including additional data like the chunk to be written, offsets, lengths, and more. Replies from the server may contain an error, success value, or data, depending on the type of the reply.

Despite its utility, the NBD protocol has limitations. As discussed by Pieter-Ján Busschaert et al. in "Design and Implementation of a Generic Network Block Device" (2008), the protocol's maximum message size is 32 MB, but the maximum block/chunk size supported by the kernel is just 4096 KB. This makes it less suitable for use over WAN, especially in high latency scenarios.

The NBD protocol also allows for the listing of exports, enabling the enumeration of multiple memory regions on a single server. However, due to its age, the protocol has multiple different handshake versions and legacy features. In most use cases, implementing the latest recommended versions and the baseline feature set suffices. This includes no TLS, the latest "fixed newstyle" handshake, the ability to list and choose an export, and read, write and disconnect commands and replies. As such, the protocol is quite simple to implement.

However, simplicity can bring drawbacks. NBD may not be the best fit for use cases where the backing device diverges significantly from a random-access storage device, such as a tape drive, as it does not support high-level abstractions like files or directories. However, for specific use cases like nar-

row memory synchronization, this simplicity and lack of high-level abstraction can be considered an advantage, contributing to the protocol's efficiency and effectiveness.

2.5 Virtual Machine Live Migration

Virtual Machine Live Migration is a technology used to transfer running virtual machines, along with their active state and connected devices, from one host to another with minimal downtime. This technology greatly enhances system flexibility and efficiency, offering benefits such as load balancing, hardware maintenance, and fault management.

2.5.1 Pre-Copy

A key strategy in the field of live VM migration is the pre-copy algorithm, an approach which minimizes downtime by duplicating data from the source to the destination while the virtual machine continues to run. As affirmed by Hines and Gopalan in their paper "Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning" (2009), this approach allows for the optimization of chunk fetching beyond what the Network Block Device (NBD) protocol over WAN can offer.

The pre-copy migration process begins with the copying of the initial state of the VM's memory to the destination. During this operation, if any chunks of data are modified, they are marked as 'dirty'. These dirty chunks are then copied to the destination, continuing until the number of remaining chunks is reduced sufficiently to satisfy a maximum downtime criterion. At this point, the VM is suspended on the source, and the remaining chunks are synchronized to the destination. Upon completion of the transfer, the VM is resumed at the destination.

This process is beneficial as it ensures the full availability of the VM either at the source or the destination at all times. It also exhibits resilience to network outages during synchronization, as even if the connection is lost, the process can resume once connectivity is restored.

However, there are limitations to the pre-copy migration approach. If too many chunks are altered on the source during the migration, it may not be possible to meet the maximum acceptable downtime criteria. In such cases, as per the study "Performance Evaluation of VM Live Migration in Cloud Computing Systems" by Wang and Morabito (2016), the process may enter a state of perpetual migration. Additionally, the maximum acceptable downtime is inherently bound by the round trip time (RTT) available in the network, which may vary widely based on network conditions. Therefore, while pre-copy migration is an effective tool in live VM migration, it requires careful consideration of the workload characteristics and network conditions to be successful.

2.5.2 Post-Copy

The post-copy migration strategy is a notable alternative to the pre-copy approach in the context of Virtual Machine Live Migration. It operates on a fundamentally different principle, transferring the virtual machine (VM) to the destination with only a minimal set of chunks and then fetching additional data as required.

In the post-copy migration, the VM is initially suspended on the source and then relocated to the destination with a limited set of data chunks. Upon reaching the destination, the VM resumes operation. As explained by Hu et al. in "Performance Evaluation of Virtual Machine Live Migration" (2010), when the VM attempts to access a chunk that is not available at the destination, a page fault occurs. This fault triggers the fetching of the missing page from the source, after which the VM execution continues.

One of the key advantages of post-copy migration is that it eliminates the need to retransmit dirty chunks to the destination before hitting the maximum tolerable downtime. This can lead to significant network traffic reduction, particularly for workloads with high memory write rates.

However, the post-copy strategy is not without its drawbacks. It can potentially result in longer overall migration times due to the need for fetching chunks from the network on-demand. This method is highly sensitive to network latency or round-trip-time (RTT), as each page fault leads to a network request to fetch the missing data. Consequently, in high-latency networks, the time taken to serve each page fault may contribute to a substantial increase in the total migration time.

In essence, while post-copy migration offers a distinct set of benefits, such as decreased network traffic and the elimination of repeated transmission of dirty chunks, its effectiveness can be heavily influenced by network conditions. This emphasizes the importance of considering both the workload characteristics and the specific circumstances of the network environment when choosing a VM migration strategy.

2.5.3 Workload Analysis

The paper "Reducing Virtual Machine Live Migration Overhead via Workload Analysis" offers an intriguing exploration of the timing of migration events, and while it predominantly targets virtual machines (VMs), its findings can potentially extend to other applications or migration scenarios.

This paper introduces a method that identifies the workload cycles of VMs and uses this information to decide whether to postpone migration. The method operates by analysing the cyclic patterns that can unnecessarily extend a VM's migration time. It subsequently identifies optimal cycles in which to conduct the migration.

In the context of VMs, such cycles could arise due to various events, such as the garbage collection (GC) of a large application triggering substantial changes to the VM's memory. When a migration proposal

occurs, the system assesses whether it is within a beneficial cycle to carry out the migration. If it is, the migration proceeds; if not, the system delays the migration until the next cycle.

This approach leverages a Bayesian classifier to distinguish between favourable and unfavourable cycles. Compared to the conventional alternative - typically waiting until a significant percentage of unmodified chunks since the commencement of tracking are synced - this method can potentially provide considerable enhancements.

According to the research paper's findings, there is an observed improvement of up to 74% in terms of live migration time/downtime and a 43% reduction in the amount of data transferred over the network. While such a system has not been implemented for r3map, the potential for coupling r3map with this kind of system certainly exists and could yield significant performance improvements.

The results of this study underline the potential benefits of applying advanced workload analysis to the migration of VMs. The integration of such an analysis can result in improvements in migration times, reduction in network load, and generally more efficient use of computational resources. However, these advantages should be weighed against the potential costs and complexities of implementing and maintaining such a system.

3 Planning

3.1 Pull-Based Synchronization With `userfaultfd`

The approach of pull-based synchronization is quite innovative in dealing with page faults in userspace. Recently, Linux kernel 4.11 introduced a new system called `userfaultfd`, which elegantly enables this process. The `userfaultfd` system permits the handling of these page faults in userspace, making it possible to handle memory in an innovative and flexible manner.

In practice, the memory region for handling is allocated, for example, using the `mmap` system call. This allows the creation of new memory regions for use, which is the first step in creating a userspace page fault handler. Once the memory region is prepared, the file descriptor for the `userfaultfd` API is obtained. The next crucial step is to transfer this file descriptor to a process that should respond with the memory chunks that are intended to be put into the faulting address. This operation typically involves inter-process communication and synchronization, a challenge that necessitates careful programming and error handling.

After the receipt of the file descriptor, the handler for the API needs to be registered. It's this handler that will receive the addresses that have faulted, and it's responsible for responding with the appropriate actions. Specifically, the handler responds with the `UFFDIO_COPY` `ioctl`, along with a pointer to the memory chunk that should be used. The `UFFDIO_COPY` `ioctl` command provides a mech-

anism to resolve the page fault by copying memory into the faulting address space. This action is demonstrated in a code snippet available on GitHub in the repository named `userfaultfd-go`.

While the `userfaultfd` mechanism offers significant advantages for fine-grained memory control, it also comes with a degree of complexity. Understanding the principles of memory management, page faults, and kernel-user space interactions is crucial. Moreover, writing safe and efficient userspace handlers demands a deep understanding of these concepts, making the effective use of `userfaultfd` a challenge for developers.

As mentioned by Corbet, J., & Rubini, A. (2001) in their renowned book “Linux Device Drivers”, a detailed understanding of memory management at the kernel level, including concepts such as `mmap` and `ioctl`, is key to unlocking the full potential of systems like `userfaultfd`. It is by building upon these foundations that the community can innovate and improve upon existing paradigms for handling page faults and memory synchronization.

3.2 Push-Based Synchronization With `mmap` and Hashing

The notion of push-based synchronization, specifically using `mmap` and hashing, offers an alternative approach to handle memory synchronization between systems. Rather than responding to page faults as in `userfaultfd`, this technique employs a file to track modifications to a memory region. By synchronizing this representative file between systems, we achieve memory region synchronization.

This approach mirrors how Linux utilizes swap space, enabling chunks of memory to be moved to disk or another swap partition when high-speed RAM is not a requirement. A process referred to as “paging out”. Moreover, Linux can also load missing memory chunks from the disk. This method operates analogously to how `userfaultfd` handles page faults, but it circumvents user space, potentially enhancing the speed of execution.

To detect changes in files, the Linux kernel offers the `inotify` system. This system enables applications to register handlers on file events, like `WRITE` or `SYNC`, thereby facilitating efficient file synchronization, which is used by various file synchronization utilities. It allows us to filter only the events required to sync the writes, which makes it an ideal choice for our use case.

However, due to technical restrictions—primarily because the file is represented by a memory region—Linux doesn’t fire these events for `mmap`ed files. Thus, we cannot leverage `inotify` in this context. As an alternative, we have two options: either continuously poll for file attribute changes, for instance, last write, or continuously hash the file to detect changes.

Polling comes with its set of limitations, such as an intrinsic minimum latency as we need to wait for the next polling cycle. This latency can negatively impact scenarios where maximum allowable downtime is crucial, and where the overhead of polling can critically affect system performance.

On the other hand, hashing the entire file is an inherently I/O- and CPU-intensive process since the entire file must be read at some point. Nevertheless, polling and hashing is possibly the only reliable way of detecting changes to a file. Instead of hashing the whole file then syncing it, we can refine this process to only sync parts of the file that have changed between two polling iterations.

As per the paper "Hash functions and integrity checks for network file synchronization" by P. Deutsch and J-L Gailly (1996), hashing techniques can be efficiently employed to detect changes in files for synchronization purposes. Although this method may not be directly applicable to memory synchronization, the principles underpinning it offer an intriguing area of exploration for efficient memory synchronization techniques.

3.3 Push-Pull Synchronization with FUSE

The exploration of push-pull synchronization with FUSE (Filesystem in Userspace) provides an innovative approach to address the challenges related to efficient virtual memory management. Given the fact that push-based synchronization demands intensive CPU and I/O operations and `userfaultfd-go` only allows low throughput, a more optimized solution is required.

One potential solution could be to design a custom filesystem in Linux, implemented as a kernel module, that could intercept read/write operations to the `mmap`ed region. This approach could allow a custom backend to respond to these operations, eliminating the need for `userfaultfd-go` or resource-consuming hashing techniques.

However, this approach is not without potential drawbacks. It requires direct operation in the kernel, which presents significant complexities. As the paper "Building Reliable, High-Performance Communication Systems from Components" (Ender, et al., 1998) highlights, kernel-level development carries inherent risks and complexities, including limited portability and increased difficulty for distribution to end users.

Furthermore, while Rust can be utilized to develop kernel modules instead of the conventional C, there are still several issues that need addressing. Kernel modules are built for specific kernels, which makes them less portable and more challenging to distribute to users. Although a kernel filesystem could bypass user space to save on context switching, it would operate in the kernel's address space, yielding a low level of isolation.

Moreover, iterating on a kernel module is considerably more difficult compared to a user-space program, thereby making rapid development and continuous integration a substantial challenge. Additionally, if we desire users to supply their own backends for push-pull operations, this would still demand communication between user space and the kernel.

In summary, while implementing this approach in the kernel is possible, it would pose a high level of complexity and potential risk. Therefore, it's crucial to weigh these factors against the potential ben-

efits and explore other viable alternatives to kernel-space programming, such as leveraging existing systems and frameworks like FUSE, which offer greater flexibility, safety, and ease of use, as stated by Szabo et al. in “FUSE: Making Sense of the Unknown” (2012).

3.4 Pull-Based Synchronization With NBD

Block Device-based synchronization offers an innovative solution to enhance the effectiveness of Virtual Machine (VM) live migration, as it capitalizes on the unique features of block devices in Linux. Unlike the conventional use of files, a block device can be utilized for mmap (memory map), thereby capturing reads/writes to a single mmaped file.

Block devices in Linux are typically storage devices that allow reading and writing fixed chunks, or blocks, of data. This process could be similar to how a file system can be implemented in a kernel module. However, the implementation of such a solution in kernel space may raise several issues related to security, portability, and developer experience, as outlined in “Taming Hosted Hypervisors with (Mostly) Deprivileged Execution” (Belay et al., 2012).

To address these challenges, the utilization of a Network Block Device (NBD) server emerges as a potential solution. The NBD server can be used by the kernel NBD module in a similar way to how a process interacts with the FUSE kernel module. A key distinction between NBD and FUSE (Filesystem in Userspace) lies in the functionalities they provide. Unlike FUSE, an NBD server does not offer a file system; instead, it provides the storage device, typically hosting a file system. This design makes the interface for NBD significantly simpler and reduces implementation overhead.

The utilization of NBD server’s backend is more comparable to how `userfaultfd-go` operates rather than FUSE, as highlighted in “Network Block Device (NBD) Protocol - A Linux block device server over a network” (Baravalle and Chambers, 2020). Therefore, the use of an NBD server offers a more streamlined and efficient method for VM live migration, presenting a unique combination of reduced complexity and enhanced performance.