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

> 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.

Memory management in Linux plays a crucial role in ensuring efficient utilization of system resources. The Linux kernel employs various mechanisms, such as virtual memory, to provide each process with its own address space. This enables processes to access memory regions specific to their needs, while maintaining isolation and security.

In recent years, there has been a growing interest in exploring memory as a universal storage API. The concept revolves around using memory as a means to access not only traditional data stored in RAM but also other resources, such as files, network resources, and even remote objects. By treating memory as a universal interface, applications can seamlessly access and manipulate diverse resources using familiar memory-based operations.

Considering the potential of memory as a universal access mechanism raises the question: What would be possible if memory were the universal way to access resources? This paradigm shift could unlock numerous benefits, including simplified programming models, improved performance, and enhanced interoperability. Applications could treat different resources uniformly, utilizing a consistent set of memory operations to read, write, and manipulate data regardless of its origin or type.

Efficient memory synchronization emerges as a crucial component in realizing the vision of memory as a universal storage API. Synchronization mechanisms ensure that data in memory remains consistent across different processes or systems that access and modify it concurrently. Efficient synchronization enables seamless sharing and coordination of data, facilitating collaboration and enabling real-time data processing and analysis.

In various industries today, high-level use cases for memory synchronization are becoming increasingly relevant. In the financial sector, real-time trading systems require efficient synchronization of market data across multiple trading applications and strategies. In scientific research, large-scale simulations demand synchronized access to shared memory spaces to enable parallel processing and analysis. Similarly, in the gaming industry, multiplayer games rely on efficient memory synchronization to maintain consistent game states across players.

Moreover, memory synchronization can enable distributed computing frameworks to efficiently exchange data among distributed nodes, facilitating scalable and fault-tolerant processing. Industries such as artificial intelligence and machine learning can leverage memory synchronization to enable distributed training and inference across multiple compute nodes.

Overall, efficient memory synchronization holds the potential to revolutionize various industries by enabling seamless access to diverse resources and facilitating real-time collaboration and data processing. As advancements continue in this field, it is expected that more innovative and high-impact use cases will emerge, further highlighting the importance of efficient memory synchronization in modern computing systems.

2 Pull-Based Memory Synchronization with userfaultfd

Page faults occur when a process attempts to access a memory region that has not been mapped into its address space. By monitoring these page faults, we can identify when a process intends to access a specific memory segment. Leveraging this information, we can implement a pull-based approach where we retrieve the required memory chunk from a remote location and map it to the address where the page fault occurred. This strategy ensures that data is fetched only when it is needed, optimizing memory utilization.

Typically, the handling of page faults is a responsibility of the kernel. However, in our case, we aim to handle page faults in userspace. Previously, this was achievable by handling the SIGSEGV signal within the process. However, we can now utilize a more elegant solution called userfaultfd, which became available since kernel version 4.11. The userfaultfd system allows for the handling of these page faults in userspace, offering improved control and flexibility.

Implementing this functionality in Go presented challenges, requiring the use of the unsafe package. To interact with system calls like ioctl, we can utilize the syscall and unix packages. These packages provide the necessary tools to communicate with the kernel and perform operations such as registering handlers for userfaultfd.

By utilizing these mechanisms, we can effectively handle page faults in userspace and leverage the capabilities of userfaultfd to facilitate efficient memory synchronization in our applications.

To utilize the userfaultfd API, we can employ the ioctl syscall to obtain a file descriptor that corresponds to the API. This file descriptor is then used to register the API, enabling it to handle any page faults that occur within the designated memory region. The allocation of the region to be handled can be performed using mmap or similar techniques.

Once we have acquired the file descriptor for the userfaultfd API, we need to transfer it to the process responsible for providing the memory chunks to be placed into the faulting addresses. File descriptor passing between processes can be achieved using a UNIX socket, which facilitates the communication and exchange of file descriptors.

Upon receiving the socket, we must register the appropriate handler for the userfaultfd API to handle the requested memory chunks. When the handler receives a faulting address, indicating a page fault, it responds by invoking the UFFDIO_COPY ioctl and provides a pointer to the corresponding memory chunk to be used with the file descriptor.

One significant advantage of utilizing userfaultfd in conjunction with the pull-based approach is the ability to simplify the backend of the entire system to an io.ReaderAt interface. This means that nearly any implementation of io.ReaderAt can serve as a backend for a userfaultfd-go registered object. This flexibility allows us to leverage various data sources and treat them uniformly, enhancing the versatility and usability of the system.

By utilizing the userfaultfd API, transferring file descriptors, and simplifying the backend to an io.ReaderAt interface, we can effectively handle page faults in userspace, facilitate efficient memory synchronization, and seamlessly integrate diverse data sources into our applications.

In the context of memory synchronization using userfaultfd, we can assume that access to memory will always be aligned to 4 KB chunks or the system's page size. This knowledge allows us to determine an appropriate chunk size on the server side based on this alignment.

In one example scenario, we can return a random pattern as the backend for the memory synchronization. This demonstrates a powerful capability of exposing arbitrary information into a byte slice without the need for pre-computation or modifying the application's logic.

Furthermore, since a file is a valid implementation of io.ReaderAt, we can directly use a file as the backend for memory synchronization. This effectively allows us to mount a (remote) file into memory, enabling seamless access and manipulation of the file's contents.

Similarly, we can utilize memory synchronization to map a remote object from S3 into memory. By accessing only the required chunks of the object, which can be achieved using HTTP range requests, we can efficiently utilize memory while working with remote resources stored in S3.

Using userfaultfd, we have the ability to map almost any object into memory, providing a flexible and versatile approach to memory synchronization. This method offers a clean implementation with relatively low overhead.

However, it is important to note that this approach has significant architecture-related limitations that restrict its use cases. One prominent limitation is the inability to catch subsequent page faults. Once a chunk of memory has been accessed and retrieved, all future requests for the same chunk will directly return the data from the destination host's RAM, bypassing the memory synchronization mechanism. This constraint hinders the synchronization of dynamically changing or frequently updated data and restricts the usefulness of the approach.

Despite these limitations, userfaultfd-based memory synchronization remains a valuable technique for specific use cases that require efficient access to memory and the ability to map diverse resources into memory for manipulation and processing.

The limitations of the userfaultfd-based memory synchronization approach become apparent when dealing with remote resources that undergo updates. Since the system can only respond to the first access to a memory chunk, subsequent updates to the remote resource will not be reflected in the synchronized memory. This restriction also prevents the handling of concurrent writers or shared resources, as there is no mechanism to update conflicting sections of memory.

As a result, the userfaultfd approach is primarily suitable for read-only scenarios, where remote resources can be mounted into memory for efficient access but lack synchronization capabilities.

Furthermore, the current design hinders the ability to pull memory chunks before they are accessed, requiring additional layers of indirection to overcome this limitation. This can introduce complexity and reduce efficiency in certain use cases.

The synchronous nature of the userfaultfd API socket poses another challenge. Each memory chunk needs to be sent sequentially, making the approach vulnerable to long round-trip time (RTT) values. Consequently, the initial latency of each request is at least equal to the RTT to the remote source. Without caching mechanisms, subsequent requests also suffer from similar delays.

One of the significant drawbacks is the limited maximum throughput achievable in real-life scenarios. Even with a local process handling page faults on a modern computer, the maximum attainable throughput is approximately 50MB/s. This performance limitation restricts the usability of the approach, particularly when dealing with larger datasets or operating in high-latency scenarios or wide-area networks (WANs).

A benchmark analysis is crucial to understand the sensitivity of userfaultfd to network latency and throughput. This evaluation can provide insights into the performance characteristics of the approach and help identify its limitations in practical use cases.

In summary, while the userfaultfd-based memory synchronization approach is interesting and idiomatic to Go, it falls short in addressing the requirements of most data-intensive applications, especially those involving larger datasets or operating in high-latency scenarios or WAN environments. Therefore, there is a need for a more robust and efficient solution to overcome these limitations and enable effective memory synchronization in a wider range of use cases.

3 Push-Based Memory Synchronization with `mmap` and Hashing

To address the limitations of the userfaultfd-based memory synchronization approach, an alternative push-based synchronization method can be employed. This approach aims to improve upon userfaultfd by utilizing a file to track changes to a memory region and synchronizing this file between two systems to effectively synchronize the corresponding memory region.

In Linux, the concept of swap space allows the operating system to move chunks of memory to disk or other swap partitions when the fast access of RAM is not required. This process, known as “paging out,” is similar to how the proposed synchronization method works. Linux can load missing chunks from a disk, eliminating the need to go through userspace as required by userfaultfd. Consequently, this approach can potentially offer faster synchronization.

The mmap function in Linux enables the mapping of a file into memory, providing a means to synchronize the file representing the memory region. By default, mmap does not automatically write changes from the file back into memory, regardless of the file descriptor's permissions. However, the MAP_SHARED flag can be added when using mmap. This flag instructs the kernel to write back changes made to the memory region to the corresponding regions of the backing file.

Similar to userfaultfd, Linux caches reads from a backing file, meaning that only the first page fault would require fetching from the disk. Subsequent reads can be satisfied from the cache, improving performance. The same principle applies to writes; just as files need to be synced to ensure the changes are written to disks, mmaped regions need to be msynced to flush the changes to the backing file.

By utilizing mmap, tracking changes in a file, and employing synchronization mechanisms such as msync, it becomes possible to establish a push-based memory synchronization method that eliminates the limitations of userfaultfd. This approach offers faster synchronization by leveraging the native capabilities of the Linux kernel, improving efficiency and addressing the architectural challenges of the previous approach.

In order to effectively synchronize changes to the memory region between hosts by syncing the underlying file, it is crucial to ensure that the changes are accurately represented in the file. This is where the msync function becomes critical. By invoking msync, the changes made to the memory region are synchronized and reflected in the underlying file.

When dealing with regular files, the O_DIRECT flag can be used to bypass kernel caching if the process already performs its own caching. However, this flag is ignored when working with mmaped regions, necessitating alternative approaches.

Traditionally, inotify is utilized to monitor changes to a file. This feature enables applications to register event handlers for specific events, such as WRITE or SYNC. Inotify provides efficient file synchronization and is widely used by file synchronization tools. It offers the ability to filter events, making it a suitable choice for this use case. However, Linux does not fire these events for mmaped files due to technical reasons, rendering inotify ineffective for monitoring changes in this context.

The next viable options for detecting changes in the file are polling for file attribute changes, such as the last write timestamp, or continuously hashing the file to check for changes. Polling has its downsides, including the guaranteed minimum latency introduced by waiting for the next polling cycle. This latency can impact scenarios with strict maximum allowed downtime, where the polling overhead becomes a critical factor.

On the other hand, hashing the entire file is an IO- and CPU-intensive process since the entire file needs to be read at some point. However, polling and hashing combined offer a reliable approach for detecting changes in a file.

When selecting algorithms for the hashing process, the key metric to consider is the throughput at which the hashes can be computed, along with the likelihood of collisions. This ensures that the hashing process is efficient and accurate in detecting changes.

In summary, while inotify is the preferred method for file synchronization, it is not suitable for monitoring changes in mmaped files. As a result, alternatives such as polling and hashing become the reliable options for detecting changes in the file. These methods come with their own trade-offs, and the choice of hashing algorithm should prioritize throughput and collision prevention.

To optimize the synchronization process and minimize the amount of data transferred, we can adopt an approach where only the parts of the file that have changed between polling iterations are synced, rather than hashing and syncing the entire file. This can be achieved by opening the file multiple times, hashing individual offsets, and aggregating the changed chunks.

By hashing and syncing individual chunks, we can better utilize CPU resources for concurrent processing, especially if the underlying hashing algorithm is CPU-bound. This approach provides better scalability and performance in scenarios where concurrent processing is beneficial.

However, opening multiple file descriptors for each chunk increases the initial latency and overhead. To assess the impact of this approach, benchmarking can be conducted to compare the performance of hashing individual chunks versus hashing the entire file.

One significant advantage of hashing and syncing individual chunks is the reduction in data transfer. By synchronizing only the delta, the amount of data that needs to be transferred between systems is drastically decreased. This optimization not only speeds up each individual polling tick but also minimizes network usage and improves overall efficiency.

The practice of hashing and/or syncing individual chunks that have changed is commonly employed in file synchronization tools. One prominent example is rsync, a popular tool used for file synchronization. When the delta-transfer algorithm is active in rsync, it computes the difference between the local and remote file and synchronizes only the changes. This approach further emphasizes the effectiveness of syncing only the modified chunks instead of the entire file.

In summary, hashing and syncing individual chunks that have changed offers a more efficient and optimized approach to file synchronization. It reduces the amount of data transferred, improves performance by leveraging concurrent processing, and aligns with common practices seen in widely-used file synchronization tools like rsync.

The delta synchronization algorithm, commonly employed in tools like rsync, follows a specific process to efficiently synchronize file changes. First, the file on the destination side is divided into fixed-size blocks. A weak and fast checksum is calculated for each block, and these checksums are sent to the source. On the source side, the same checksum calculation process is performed, and the checksums are compared to identify the changed blocks.

Once the changed blocks are determined, the source sends the offset and data of each changed block to the destination. Upon receiving a block, the destination writes the data to the specified offset, effectively reconstructing the file. This process repeats for each polling interval, ensuring continuous synchronization of changes.

To facilitate this delta synchronization, a TCP-based protocol can be implemented. Similar to rsync's delta synchronization algorithm, a simple multiplexing system can be employed by sending a file ID with each message. This protocol allows for efficient transmission of the changed data between the source and destination systems.

However, this delta synchronization system also has its limitations. Unlike userfaultfd, which only catches reads, this system is limited to capturing writes to the file. As a result, it is essentially a write-only system, and adding hosts to the network at a later stage becomes inefficient.

To address this limitation, a star-based architecture with a central forwarding hub can be used when there are multiple possible destinations for migrating state. This static topology allows for hashing to be performed only on one destination and the source, rather than on all destinations. Consequently, changes need to be pushed only to the hub, which then forwards the messages to all the other destinations, reducing the overall overhead.

Benchmarking can be conducted to compare the throughput of this custom synchronization protocol, which utilizes the delta synchronization algorithm, against rsync, which hashes entire files. This evaluation provides insights into the performance characteristics and efficiency of the custom synchronization protocol in comparison to the widely-used rsync approach.

In summary, the delta synchronization algorithm, implemented through a custom TCP-based protocol, enables efficient synchronization of file changes. While it is limited to capturing writes and requires careful consideration in scaling the network, it offers a practical approach to continuously synchronize file modifications. Benchmarking against rsync can shed light on the relative performance and effectiveness of the custom synchronization protocol.

4 Push-Pull Memory Synchronization with a FUSE

To overcome the limitations of the push-based method and the low throughput of userfaultfd-go, an alternative solution is needed. What if it were possible to capture read and write events without relying on userfaultfd-go or performing hashing?

One approach could involve creating a custom file system in Linux and loading it as a kernel module. This file system would intercept reads and writes to and from the mmaped region, allowing custom backends to respond to these events. However, implementing such a system directly in the kernel introduces several potential drawbacks.

Writing kernel modules, traditionally done in C, has its challenges. Although it is now possible to write kernel modules in Rust, many issues persist. Kernel modules lack portability as they are built for specific kernels, making distribution to users difficult. Furthermore, running a file system in the kernel eliminates the need for user space involvement, reducing context switching. However, it also means running in the kernel address space, which compromises isolation.

Iterating on a kernel module is much more challenging compared to iterating on a program running in user space. Making changes to a kernel module often involves more complex processes and can be slower, hindering development efficiency.

Additionally, if the goal is to allow users to provide their own backends for pulling and pushing, communication between user space and the kernel would still be required. This further adds to the complexity of the implementation.

While it is technically possible to add this custom file system implementation in the kernel, it would introduce significant complexity and potential drawbacks. The trade-offs in terms of development efficiency, distribution, isolation, and overall system complexity must be carefully considered when evaluating this approach.

To implement file systems in user space, the FUSE (Filesystem in Userspace) API can be utilized. With FUSE, a user space program registers itself with the FUSE kernel module and provides callbacks for various file system operations, such as open, read, and write. When a user performs a file system operation on a mounted FUSE file system, the kernel module sends a request for the operation to the user space program. The program then replies with a response, which the FUSE kernel module returns to the user.

Using FUSE significantly simplifies the process of creating a file system compared to writing it directly in the kernel. The file system can run in user space, which is safer as errors in the FUSE module or the backend do not risk crashing the entire kernel. This layer of indirection also makes the file system portable, as it only needs to communicate with the FUSE module rather than relying on a custom kernel module.

One of the advantages of using FUSE is the ability to use complex and seemingly incompatible backends as the file system's backend. By leveraging a file system abstraction API like afero.Fs, the FUSE implementation can be separated from the actual file system structure. This separation enables unit testing and the possibility of adding caching mechanisms in user space. It also allows for flexibility in switching between different file system backends without requiring FUSE-specific modifications.

With the ability to map any afero.Fs to a FUSE backend, the implementation can support various file system backends without directly writing FUSE-specific code. This flexibility makes it easier to switch between different file system implementations without major code modifications.

In summary, utilizing the FUSE API for implementing file systems in user space provides numerous benefits. It simplifies the development process, enhances safety by avoiding custom kernel modules, enables portability, supports complex backends, and allows for the separation of the FUSE implementation from the file system structure.

While FUSE provides flexibility for implementing file systems in user space, it also has its limitations. Operating in user space necessitates context switching, which can introduce overhead and impact performance. Additionally, some advanced features may not be available when using FUSE.

Furthermore, the overhead of using FUSE, as well as implementing a completely custom file system, remains significant for the purpose of synchronizing memory. In scenarios where memory synchronization is the primary focus, exposing a full file system to track changes might not be the optimal solution.

Alternatively, an approach that exposes a single file for tracking changes could be more efficient and practical. However, implementing such a file system comes with significant implementation overhead. As a result, the drawbacks and complexities associated with this approach might outweigh its benefits, leading to its exclusion as the chosen solution.

Considering the specific requirements and constraints of synchronizing memory, it is essential to carefully evaluate the trade-offs of different approaches and select the most suitable solution that balances performance, complexity, and implementation effort.

5 Pull-Based Memory Synchronization with NBD

To address the need for a more efficient API for catching reads and writes to a single mmaped file, an alternative approach can be considered that does not require implementing a complete file system. Instead, a block device can be utilized.

In Linux, block devices are typically storage devices capable of reading and writing fixed chunks of data known as blocks. Similar to mmaping a file, it is possible to mmap a block device. However, block devices are typically implemented as kernel modules or reside in kernel space, introducing similar challenges regarding security, portability, and developer experience as with file system implementations.

Instead of using FUSE, an alternative solution involves creating a Network Block Device (NBD) server. The NBD server can be used by the kernel NBD module, similar to how the process connected to the FUSE kernel module. The key distinction is that the NBD server does not provide a file system; it serves as the storage device hosting the file system.

Compared to FUSE, the interface for an NBD server is much simpler since it focuses solely on providing the storage device. Consequently, the implementation overhead of an NBD server's backend is more akin to how userfaultfd-go operates, rather than the complexities associated with a full FUSE file system implementation.

By leveraging a NBD server, it becomes possible to achieve efficient synchronization of memory without the need for a complete file system. This approach simplifies the implementation, improves performance, and reduces the overhead associated with complex file system operations.

When considering the most appropriate solution, the trade-offs between performance, implementation effort, security, and portability should be carefully evaluated. The selection of the optimal approach will depend on the specific requirements and constraints of the memory synchronization use case.

The NBD (Network Block Device) protocol serves as the communication mechanism between the NBD server in user space and the NBD client provided by the NBD kernel module. While the NBD protocol can operate over wide area networks (WAN), it is primarily designed for usage within local area networks (LAN) or on the localhost.

The NBD protocol consists of two main phases: the handshake phase and the transmission phase. The protocol involves multiple actors, including one or multiple clients, the server, and the concept of an export, which represents a device or storage resource provided by the server.

During the handshake phase, when a client connects to the server, the server sends a greeting message that includes its flags. The client responds with its own flags and provides the name of the export it wishes to use. The server then sends metadata about the export, including its size. The client acknowledges the received data, and the handshake phase concludes.

Following the handshake, the client and server engage in exchanging commands and replies. A command in the NBD protocol can represent various basic operations needed to access a block device, such as read, write, or flush. Depending on the command, it may also include additional data, such as the chunk of data to be written, offsets, lengths, and other relevant parameters.

The NBD protocol facilitates efficient communication and coordination between the NBD server and client, enabling the transfer of block-level data and performing operations on the virtual block device provided by the server.

It's important to note that due to the design and intended usage of the NBD protocol, it is typically more suitable for LAN or localhost scenarios rather than for wide area networks.

The NBD (Network Block Device) protocol supports different types of replies, which can include an error, success value, or data, depending on the specific type of reply being sent. However, the NBD protocol has certain limitations. For instance, the maximum message size allowed is 32 MB, while the maximum block or chunk size supported by the kernel is only 4096 KB. This limitation makes the NBD protocol suboptimal for running over wide area networks (WAN), particularly in high latency scenarios.

The NBD protocol also provides functionality for listing exports, allowing clients to discover and choose from multiple memory regions provided by a single server. It is worth noting that NBD is an older protocol with various handshake versions and legacy features. In the context of this use case, it is recommended to implement the latest versions and the baseline feature set, given that both the server and client are typically controlled.

The baseline feature set includes features such as the ability to list and select an export, perform read and write commands, and handle disconnections. Implementing this subset of features simplifies the protocol implementation, but it also means that NBD is less suitable for use cases where the underlying device behaves significantly differently from a random-access storage device, such as a tape drive. However, for the specific use case of narrow memory synchronization, this limitation can be considered more of a feature than a drawback.

Due to the lack of pre-existing libraries, a new pure Go NBD library was developed. This library was implemented without relying on CGo or pre-existing C libraries. By avoiding context switching between Go and C, the pure Go NBD library can offer improved performance and efficiency for NBD protocol implementations in Go.

In summary, the NBD protocol provides a simple yet limited approach for communication between NBD servers and clients. While it may have some drawbacks and limitations, it can serve as a suitable solution for narrow memory synchronization use cases, especially when implemented with the latest recommended versions and the baseline feature set.

The go-nbd library provides a simple and straightforward backend interface consisting of four methods: ReadAt, WriteAt, Size, and Sync. An example backend that aligns well with a block device is the file backend, which enables reading and writing data to a file.

Unlike userfaultfd-go, go-nbd backends are designed to handle both reads and writes. The library also includes a Handle function, which facilitates multiple users without relying on a specific transport layer. This allows systems like WebRTC, which do not provide a TCP-style accept syscall, to be easily integrated.

Furthermore, go-nbd supports hosting NBD and other services on the same TCP socket, enabling efficient resource utilization. The server component of go-nbd leverages the binary package for encoding and decoding messages, ensuring proper communication between the server and clients. To facilitate ease of parsing, structured messages such as headers are represented as Go structs.

During the handshake phase, a simple for loop is employed to handle the negotiation process. If an error occurs, the loop returns, or else it breaks to proceed with the transmission phase. In the transmission phase, messages are read and processed based on their type, and relevant data or replies are exchanged accordingly.

Overall, go-nbd provides a user-friendly and efficient implementation of the NBD protocol, with support for multiple backends and the flexibility to integrate with various transport layers.

The go-nbd library enables the implementation of the NBD server entirely in user space without any kernel components involved. However, the NBD client utilizes the kernel NBD client. To utilize the client, it is necessary to find an available NBD device.

NBD devices are pre-created by the NBD kernel module, and additional devices can be specified using the `nbds_max` parameter. To find a free NBD device, one can either directly specify it or check for a NBD device with a zero size in the sysfs.

The negotiation for the NBD client is performed in user space by the Go program using a simple for loop, similar to the server implementation. After fetching the metadata for the export during the handshake, the kernel NBD client is configured using ioctls.

The `DO_IT` syscall used by the kernel NBD client does not return, so an external system must be used to detect when the device is ready. There are two approaches to determine readiness: polling sysfs for the size parameter or using udev.

Udev is a device management framework in Linux. When a device becomes available, the kernel sends a udev event, which can be subscribed to and used as a reliable way to wait for the ready state. However, polling sysfs directly can sometimes be faster than subscribing to the udev event. Therefore, go-nbd provides the option to switch between both methods, allowing users to choose the one that best suits their needs.

When opening the block device that the NBD client has connected to, the kernel typically provides a caching mechanism and requires a sync operation to flush changes. However, by using the `O_DIRECT` flag, it is possible to bypass the kernel caching layer and write changes directly to the NBD client or server. This approach is especially beneficial when both the client and server are on the local system, and minimizing the time spent on syncing is crucial.

Using `O_DIRECT` requires ensuring that reads and writes on the device node are aligned to the system's page size. This alignment can be achieved with a client-side chunking system, although it does require application-specific code.

NBD is a battle-tested solution for this scenario and offers good performance. However, in the future, a more lightweight implementation called ublk could also be considered. ublk utilizes io_uring, which has the potential to enable faster concurrent access. It follows a similar architecture to NBD, with a user space server providing the block device backend and a kernel ublk driver that creates `/dev/ublkb*` devices.

Unlike the NBD kernel module, which relies on slower UNIX or TCP sockets for communication, ublk leverages io_uring pass-through commands. The io_uring architecture promises lower latency and better throughput, which can be advantageous for high-performance scenarios.

While BUSE (block devices in user space) and CUSE (char device in user space) are alternative options for implementing block devices, they have their own limitations and complexities.

BUSE, similar to FUSE, consists of a kernel component and a user space server component. However, it is still in the experimental stage, and client libraries in Go are also experimental, which limits its usability compared to more mature solutions like NBD.

CUSE, on the other hand, offers a flexible way of defining char and block devices, and it allows for interesting features such as custom ioctls. However, implementing CUSE without incurring significant overhead typically requires using CGo, which comes with its own set of complexities. Calling Go closures from C code can be challenging and requires careful handling of userdata parameters.

Given the experimental nature and limitations of BUSE and CUSE, NBD was chosen as the preferred option for implementing the block device due to its stability, wide adoption, and better documentation.

Considering the complexities, limitations, and overhead associated with implementing BUSE, CUSE, and custom kernel extensions for virtual file mmaping, it was decided to proceed with NBD as the chosen solution for now.

NBD provides a battle-tested protocol and a well-established interface for implementing block devices. It offers good performance and has widespread adoption, making it a reliable choice for synchronizing memory regions.

While alternatives like ublk and potential future advancements may provide more performant options, NBD currently fulfills the requirements and provides a stable solution for the memory synchronization use case.

6 Push-Pull Memory Synchronization with Mounts

To overcome the challenges of latency when using NBD over a public network, a layer of indirection called a Mount is introduced. The Mount consists of both a client and a server, both running on the local system. This approach allows for smarter pull/push strategies and provides better performance in WAN scenarios.

The server and client are connected using a connected UNIX socket pair, providing a reliable and efficient communication channel between them. By building on this basic direct mount, additional functionalities like file and slice mounts can be implemented, enabling easy usage and integration with sync and msync operations.

These managed mounts provide a higher level of abstraction and flexibility, allowing for more sophisticated synchronization strategies and optimizing data transfer between the client and the remote server.

The use of the mmap/slice approach brings several benefits. Firstly, it allows the byte slice to be directly used as if it were allocated by `make`, while transparently mapping it to the remote backend. The mmap/slice implementation also replaces the syscall-based file interface with a random access one, enabling faster concurrent reads from the underlying backend.

An alternative approach would be to format the server's backend or block device using standard file system tools. Once the device is ready, it can be mounted to a directory on the system. This allows for mmaping one or multiple files on the mounted file system instead of directly mmaping the block device. This approach enables handling multiple remote regions using a single server, reducing initialization time and overhead.

However, using a full-fledged file system introduces storage overhead and complexity, which is why alternatives like the FUSE approach were not chosen for this use case.

The simplest form of the mount API is the direct mount API. It replaces NBD with a transport-independent RPC framework, but does not include additional optimizations. In this API, there are two actors: a client and a server, with the server providing methods to be called.

To support proper chunking in the protocol, an additional layer needs to be implemented. While the NBD client allows specifying a block size for chunking, it is limited to a maximum of 4 KB chunks. In scenarios where the round-trip time (RTT) between the backend and server is large, it may be beneficial to use larger chunk sizes for network transfers.

Many backends have constraints that require a specific chunk size or aligned offsets. For example, tape drives work best with larger chunk sizes (multiple MBs) instead of smaller ones. Even if there are no constraints on the backend side, limiting the maximum supported message size between the client and server can help prevent potential denial-of-service (DoS) attacks by forcing the backend to allocate large amounts of memory to satisfy requests.

To implement the chunking system, an abstraction layer can be used to create a pipeline of readers/writers. This can be achieved by combining the `io.ReaderAt` and `io.WriterAt` interfaces into a `ReadWriterAt` interface. This allows forwarding the `Size` and `Sync` system calls directly to the underlying backend, while wrapping the backend's `ReadAt` and `WriteAt` methods in a pipeline of other `ReadWriterAt` instances.

The `ArbitraryReadWriterAt` implementation provides a way to break down a larger data stream into smaller chunks. In the `ReadAt` method, it calculates the index of the chunk that the offset falls into and the position within the chunk. It reads the entire chunk from the backend into a buffer, copies the necessary portion of the buffer into the input slice, and repeats this process until all requested data is read. Similarly, in the `WriteAt` method, it calculates the chunk's index and offset. If an entire chunk is being written, it bypasses the chunking system and writes it directly to avoid unnecessary data copying. If only parts of a chunk need to be written, it reads the complete chunk into a buffer, modifies the buffer with the changed data, and writes the entire chunk back until all data has been written.

Additionally, there is a `ChunkedReadWriterAt` implementation that ensures the maximum size supported by the backend and the actual chunks are respected. It checks if a read or write operation is valid by verifying that the read/write offset is a multiple of the chunk size and that the length of the data slice being read/written is equal to the chunk size.

The chunking can be done either on the mount API's side or on the backend's side. However, performing the chunking on the backend's side is usually faster, as writes with lengths smaller than the chunk size would require fetching the remote chunk, resulting in increased latency, especially in high round-trip time (RTT) scenarios.

A benchmark can be conducted to compare the performance of local chunking (done on the mount API's side) versus remote chunking (done on the backend's side). This would help evaluate the impact of chunking on latency and throughput.

Pre-copy migration works by periodically transferring the memory state of the VM from the source host to the destination host while the VM continues to run. This allows the destination host to have an up-to-date copy of the VM's memory. The process typically involves iteratively copying memory pages from the source to the destination, with each iteration updating the changed pages since the previous iteration.

Post-copy migration, on the other hand, takes a different approach. Initially, only the minimal set of memory pages required to start the VM are transferred to the destination host. The VM then starts running on the destination host with a potentially incomplete memory state. As the VM accesses additional memory pages that are not yet present on the destination host, page faults occur. The page faults trigger the transfer of the missing pages from the source to the destination host on-demand. This process continues until all the required memory pages have been transferred, allowing the VM to fully run on the destination host.

The managed mount API aims to optimize the live migration of VMs by implementing a smart pull/-push strategy. This strategy allows for the efficient transfer of memory regions between hosts, minimizing downtime. By intelligently synchronizing the memory regions between the source and destination hosts, the managed mount API can ensure that the necessary memory pages are available on the destination host before they are accessed by the VM during the migration process. This eliminates the need for on-demand transfers and reduces the downtime associated with fetching missing pages.

The managed mount API provides a higher-level abstraction for handling the migration process, including the synchronization and optimization of memory regions. It helps facilitate the efficient transfer of VM state and connected devices from the source to the destination host, ultimately minimizing downtime and improving the live migration experience.

Post-copy migration, as described, offers the benefit of not requiring the re-transmission of dirty chunks to the destination before reaching the maximum tolerable downtime. This approach allows for quicker initial VM migration since only a minimal set of chunks needs to be transferred. However, it introduces potential delays when accessing missing chunks on the destination, as fetching them from the network is latency-sensitive.

During post-copy migration, when the VM accesses a chunk that is not yet available on the destination, a page fault occurs. The missing page is then fetched from the source and delivered to the destination. The VM can continue execution once the missing page is available. This on-demand transfer of chunks introduces additional latency due to network round-trips, potentially resulting in longer migration times compared to pre-copy migration.

The trade-off with post-copy migration is that it prioritizes minimizing the initial downtime required for VM migration, at the cost of potential latency delays when accessing missing chunks. The effectiveness of post-copy migration depends on factors such as the network latency between the source and destination hosts and the frequency of VM memory access patterns.

It’s worth noting that both pre-copy and post-copy migration approaches have their advantages and considerations. The choice between them depends on factors such as the desired maximum tolerable downtime, network conditions, and the workload characteristics of the VM being migrated.

The managed mount API provides two paradigms for data synchronization: pre-copy and post-copy. While the API is primarily designed for reading from a remote resource and syncing changes back to it, the migration API is a more specialized version optimized for migrating resources between hosts.

In the pre-copy paradigm, the managed mount API includes a Puller component that performs asynchronous pulls of chunks in the background. It operates based on a heuristic function provided by the user to determine the order in which chunks should be pulled. This approach enables preemptive fetching of chunks to minimize the downtime during synchronization. The Puller component pulls chunks from a remote source asynchronously, allowing for efficient background synchronization.

The specific implementation details of the Puller component, including the chunk pulling mechanism and the pull heuristic function, are not provided in the given information. However, the concept of the Puller component is to facilitate proactive chunk fetching based on a user-defined heuristic, enhancing the performance of the pre-copy synchronization process.

It’s important to note that the migration API, which is mentioned briefly, likely offers a more optimized approach specifically tailored for migrating resources between hosts.

The pull heuristic plays a crucial role in optimizing the data synchronization process. By leveraging the correct pull heuristic, applications can ensure that the most important or frequently accessed data is fetched first, resulting in faster availability of the resource locally.

For example, if a resource consists of a header followed by the main data, using a pull heuristic that prioritizes fetching the header chunks first can significantly speed up the process. Similarly, in the case of synchronizing a file system where superblocks are stored in a known pattern, a pull heuristic can be designed to fetch these superblocks first.

In formats like MP4, which have an index, the pull heuristic can be utilized to fetch the index first. This allows the index to be accessed and processed while the remaining data is pulled in the background, reducing the overall synchronization time.

Once the puller has determined the order in which chunks should be pulled based on the heuristic, it starts multiple worker threads to fetch the chunks in parallel. The puller itself is responsible for requesting the chunks to be pulled but does not handle the copying to/from a destination. The actual copying is managed by a separate component.

The SyncedReadWriterAt component is responsible for the copy logic. It takes both a remote reader (the ReaderAt) and a local ReadWriterAt, which is responsible for handling the actual copying between the remote and local resources.

By combining the pull heuristic, puller, and SyncedReadWriterAt components, the managed mount API can optimize data synchronization by fetching chunks in an order that prioritizes important or frequently accessed data.

The process of pulling and pushing chunks in the managed mount API involves tracking the availability of chunks and managing their synchronization between the remote reader and the local ReadWriterAt.

When a chunk is read, such as when the puller component calls ReadAt, it is marked as remote and added to a local map. The chunk is then fetched from the remote reader and written to the local ReadWriterAt. Once the chunk is successfully copied, it is marked as locally available. This means that on subsequent reads, the chunk can be directly fetched locally without the need for remote access.

A callback function is invoked to monitor the progress of the pull process. If the managed mount API is used in conjunction with the Puller component, any chunks that haven't been fetched asynchronously yet will be scheduled for immediate pulling.

When writing chunks, the process begins by tracking the chunk but immediately marks it as available locally, regardless of whether it has been pulled before. This allows for efficient local writes without requiring remote access.

The combination of the SyncedReadWriterAt and Puller components implements a modular and testable system for pull-based synchronization. It enables both pre-emptive fetching of chunks and the ability to write back changes to the remote resource.

By utilizing the Puller interface, it is possible to implement a read-only managed mount, similar to the rr+ prefetching mechanism described in the “Remote Regions” paper by Aguilera et al. However, to support write operations, a push system is also started in parallel with the pull system. This ensures that changes made locally can be pushed back to the remote resource.

In the managed mount API, the push system works alongside the pull system to synchronize changes between the local and remote resources.

The push system takes both a local and a remote ReadWriterAt as inputs. Chunks that have been modified and are ready to be pushed are marked with the MarkOffsetPushable method. This ensures that only the modified chunks are synchronized back to the remote resource, avoiding unnecessary data transfer.

Once opened, the pusher component starts a background goroutine that periodically calls the Sync method. This background worker system launches additional workers to handle individual chunks. Each worker waits for a chunk to be assigned and then reads it from the local ReadWriterAt and copies it to the remote resource. To ensure safe access, each chunk is individually locked during the process.

The pusher component also serves as a step in the ReadWriterAt pipeline. It exposes ReadAt and WriteAt methods. ReadAt acts as a simple proxy, while WriteAt additionally marks a chunk as pushable (since it has been modified) before writing to the local ReadWriterAt.

In comparison to the direct mount setup where the NBD server is directly connected to the remote resource, the managed mount API utilizes a pipeline consisting of pullers, pushers, a syncer, and an ArbitraryReadWriter. This pipeline coordinates the pulling and pushing of chunks between the local and remote resources.

The graphic representation of the four systems (pullers, pushers, syncer, and ArbitraryReadWriter) shows their connections and interactions in the managed mount API, highlighting the difference from the direct mount approach.

In a read-only scenario, the Pusher step is skipped, as there is no need to synchronize changes back to the remote resource.

If background pulls are not enabled, the creation of the Puller component is also skipped. This means that pulling from the remote ReadWriterAt can be initiated even before the NBD device is open. It allows the background pulling process to start as the NBD client and server are still initializing.

By parallelizing the startup process and allowing pre-emptive pulling of data, the initial latency can be significantly reduced. This approach is especially effective when the startup time of the NBD client and server is comparable to or longer than one round-trip time (RTT). Benchmarking can be performed to evaluate the impact of parallel startups and pulling a certain amount of data as the device starts.

The use of a simple interface and benchmarking allows for easy testing of the entire system, ensuring its reliability and performance.

The managed mounts API provides a fast and efficient option for accessing remote resources in memory. It builds upon the basic path mount interface, which allows for the construction of file and mmap interfaces.

However, implementing the lifecycle correctly in the managed mounts API can be more complicated. For example, ensuring that the msync operation on the mmaped file happens before calling Sync() on the syncer requires the use of a hooks system.

One challenge with implementing the managed mounts API in Go is related to potential deadlocking issues. When the garbage collector (GC) tries to release memory, it needs to stop the world, which can cause problems if the GC attempts to manage the underlying slice used by the mmap API or release memory while data is being copied from the mount.

These issues need to be carefully addressed to ensure the correct behavior and reliability of the managed mounts API.

The deadlock issue caused by the NBD server running in the same process as the mount's backend can be resolved by locking the mmaped region into memory. However, this workaround leads to all chunks being fetched, resulting in high latency during the Open() operation.

Such issues indicate that Go may not be the ideal language for this particular part of the system. In the future, using a language like Rust, which doesn't have a garbage collector, could provide a better alternative.

Although the current API is specific to Go, it could be exposed through a different interface to make it usable in Go. A similar approach was taken in RegionFS, where the regions file system is mounted to a path, exposing regions as virtual files. This allows permissions to be set on the virtual files, similar to using chmod, to control access to memory regions.

The approach proposed in Remote Regions, which uses standard utilities like open and chmod, allows the API to be easily used from different programming languages. It provides a way to control access to memory regions through permissions set on virtual files.

However, it's important to note that Remote Regions is primarily designed for private use cases with a limited number of hosts in a LAN environment, where low-latency connections are expected. It doesn't specifically address migration scenarios, which the modular approach of r3map supports.

Remote Regions doesn't specify how authentication would work, focusing more on authorization based on permissions. Additionally, its wire protocol seems to target LAN environments with protocols like RDMA comm modules, while r3map is designed to work over WAN with a pluggable transport protocol interface.

7 Pull-Based Memory Synchronization with Migrations

The managed mounts API we have implemented provides efficient access to a remote resource through memory. However, it is not well suited for migration scenarios where minimizing the maximum acceptable downtime is crucial.

To optimize migration, we need to split the process into two distinct phases. We can still leverage preemptive background pulls and parallelized device/syncer startup, but the push process is dropped. This approach allows us to pull the majority of the data first and then finalize the move later with the remaining data.

This approach is inspired by the pre-copy approach to VM live migration, but also incorporates some benefits of the post-copy approach. One of the challenges we face is concurrent access to the resource by multiple readers or writers, which the mount-based API does not allow safely. This constraint poses a problem for migration because suspending the VM or app writing to the source device before the transfer begins introduces significant latency.

Furthermore, the mount API was not designed to easily support resource sharing in this manner. Therefore, additional optimizations and considerations are needed to address the specific requirements of migration scenarios.

To address the challenges in migration scenarios, we have introduced a migration API that includes two new actors: the seeder and the leecher. The seeder represents the host that exposes a migratable resource, while the leecher represents the client that wants to migrate the resource to itself.

The migration process starts by running the application with its state on the seeder's mount. When a leecher connects to the seeder, the seeder begins tracking any writes to its mount. The leecher starts pulling chunks from the seeder to its local backend. Once the leecher has received a satisfactory level of locally available chunks, it requests the seeder to finalize the migration.

During finalization, the seeder stops the remote application, performs necessary sync/flush operations on the drive, and returns the chunks that were changed between the start of tracking and finalization. The leecher marks these chunks as remote, immediately resumes the application, and queues them to be pulled immediately.

By splitting the migration into two phases, we can skip the overhead of starting the device on the leecher and perform additional initialization tasks that don't depend on the application's state before suspending it. This approach helps minimize the downtime and latency associated with the migration process.

The migration API combines both the pre-copy and post-copy algorithms into a unified protocol, significantly reducing the maximum tolerable downtime and minimizing the need for re-transmitting dirty chunks. This protocol allows for a maximum guaranteed downtime that includes the time it takes to synchronize the seeder's application state, the round-trip time (RTT), and the time to fetch the chunks written between tracking and finalization.

To achieve this, the seeder defines a read-only API with familiar methods like ReadAt, as well as additional methods such as returning dirty chunks from Sync and adding a Track method. The seeder also exposes a mount through a path, file, or byte slice, enabling continued access to the underlying data by the application during the migration process.

The tracking functionality is implemented using a TrackingReadWriterAt, which is connected to the seeder's ReadWriterAt pipeline. This modular and composable approach allows for efficient tracking of changes and synchronization of the migrated resource.

Overall, the migration API provides a comprehensive solution that combines the benefits of pre-copy and post-copy algorithms, resulting in reduced downtime and efficient migration of resources.

Once the tracking is activated by calling Track, the tracker intercepts all WriteAt calls and adds them to a local de-duplicated store. When Sync is called, the changed chunks are returned, and the deduplicated store is cleared. This mechanism ensures that only the client initiates the RPC calls, providing flexibility in using different transport layers and RPC systems. By returning an abstract service utility struct from Open, the backend can be implemented using various RPC frameworks such as gRPC.

When the leecher is opened, it calls Track on the seeder and concurrently starts the device. The leecher introduces a new component called LockableReadWriterAt into its internal pipeline. This component blocks all read and write operations to and from the NBD device until Finalize is called. This is necessary to prevent stale data from poisoning the cache on the mmaped device before the changes are marked as finalized.

Overall, this approach allows for flexible RPC implementation and ensures proper synchronization and blocking of operations during the migration process to maintain data integrity.

Once the leecher has started the device, it sets up a syncer to synchronize data with the remote. A callback function can be used to monitor the progress of the pull process. After achieving a satisfactory level of local data availability, the leecher can call the Finalize method.

During the Finalize process, the leecher calls Sync() on the remote to ensure data consistency. It then marks the changed chunks as remote and schedules them to be pulled in the background. This ensures that any modifications made to the data during the migration process are properly synchronized.

To prevent accessing the path/file/slice too early and potentially causing deadlocks, only the Finalize method returns the managed object. This design decision ensures that the migration process follows a controlled flow and minimizes the risk of concurrency issues.

Once a leecher has successfully reached 100% local availability and completed the migration, it calls the Close method on the seeder and disconnects from it, causing both the leecher and seeder to shut down. At this point, the migration process is considered complete.

After the leecher has exited, a new seeder can be started to allow for migrating from the destination to another destination if needed.

When to start the finalization step in the two-step migration API is an interesting question. The finalization stage is critical and can be challenging to recover from depending on the implementation. Calling Finalize multiple times to restart the memory sync is not straightforward since it requires suspending the VM or app on the source device before Finalize can return. This suspension operation is not necessarily idempotent and may involve shutting down dependencies and other complex tasks. Therefore, careful consideration should be given to the timing and conditions for initiating the finalization step in order to ensure a successful migration.

The paper “Reducing Virtual Machine Live Migration Overhead via Workload Analysis” presents an interesting analysis of options for determining the optimal timing of migrations. Although primarily focused on virtual machine migration, the concepts and techniques discussed in the paper could serve as a basis for other application or migration scenarios as well.

The proposed method involves identifying workload cycles of virtual machines and leveraging this information to determine when to postpone or proceed with a migration. By analyzing cyclic patterns within the workload, the system can identify optimal cycles for migrating virtual machines. For example, a favorable cycle could be identified when the VM is experiencing low activity or when certain tasks with minimal impact on migration are being performed.

Using a Bayesian classifier, the system can classify the current workload cycle as favorable or unfavorable for migration. If the cycle is deemed unfavorable, the migration is postponed until the next cycle. This approach can yield significant improvements compared to traditional methods, such as waiting for a certain percentage of unchanged chunks before initiating the migration.

According to the paper’s findings, this approach resulted in improvements of up to 74% in terms of live migration time/downtime and a reduction of 43% in the amount of data transferred over the network.

While the specific implementation of this system was not integrated into r3map, it is certainly possible to use r3map in conjunction with such a workload analysis system to further optimize migration processes and reduce overhead.

Comparing the performance of the migration API and the managed mount API in terms of maximum acceptable downtime for a migration scenario would provide valuable insights into their respective capabilities.

To conduct this benchmark, you would need to set up a migration scenario using both APIs and measure the time it takes to complete the migration process while varying the maximum acceptable downtime parameter. The benchmark would involve simulating a migration from a source to a destination, tracking the time taken to pull the necessary chunks, finalize the migration, and resume the application or VM on the destination.

By running the benchmark with different maximum acceptable downtime values and comparing the results, you can assess the effectiveness of each API in minimizing downtime during the migration process. This would help determine which API performs better in terms of reducing the downtime and meeting the specified requirements.

It's important to note that the benchmark results would depend on various factors, such as the network latency, the size and complexity of the migrated resource, and the specific implementation details of the APIs being tested. Therefore, conducting the benchmark in your specific environment would provide the most accurate insights into the performance comparison between the migration API and the managed mount API.

8 Optimizing Mounts and Migrations

Indeed, r3map offers unique advantages compared to existing remote mount and migration solutions. Its transport-agnostic design allows it to be flexible and adaptable to different deployment scenarios.

In LAN environments, where assumptions can be made about security and the absence of bad actors within the subnet, r3map can leverage fast and latency-sensitive protocols like SCSI RDMA (SRP) or custom protocols. This enables efficient and high-performance migration or mount scenarios within a local network.

On the other hand, for WAN deployments where security is a greater concern, r3map can utilize standard internet protocols such as TLS over TCP or QUIC. These protocols provide encryption and secure communication, allowing for migration over the public internet with appropriate security measures in place.

Additionally, r3map's transport-agnostic approach opens up possibilities for unique migration or mount scenarios in NATed environments using technologies like WebRTC data channels. This allows for seamless communication and migration even in complex network setups where traditional approaches may face challenges.

By supporting a wide range of transport protocols, r3map ensures compatibility with diverse networking environments while providing the flexibility to incorporate encryption and security measures based on specific deployment requirements.

Authentication and authorization can be implemented in various ways within r3map, depending on the deployment scenario. In LAN environments, where trust can be assumed within the local subnet, simple approaches like subnet-based trust can be utilized. However, for public deployments, more secure methods such as mutual TLS (mTLS) certificates or higher-level protocols like OpenID Connect (OIDC) can be employed, depending on the chosen transport layer.

For WAN deployments, new protocols like QUIC provide tight integration with Transport Layer Security (TLS), offering authentication and encryption capabilities. This ensures secure communication over the public internet, enhancing the security of the migration or mount process.

In the context of the mount use case, the initial latency of remote ReadAt requests is an important metric as it strongly influences the overall latency. To optimize this aspect, protocols like QUIC provide features such as 0-RTT (zero round-trip time) TLS, which can save one or more round-trip times and significantly reduce the overhead. This optimization not only enhances latency but also enables authentication to be handled efficiently in the same step.

Additionally, the pull-only architecture of r3map has been designed to cater to WAN deployment scenarios. By focusing on pulling data as needed, rather than pushing it proactively, the system minimizes the amount of data transferred and reduces latency, making it well-suited for remote resource access over wide-area networks.

In a pull-only system, the client retains control of tracking and pulling the necessary chunks, which offers advantages in scenarios with NATs or network outages during migration. If there is a network outage, the client can resume pulling the chunks it needs once the connection is restored. This would be more challenging to implement in a push system, as the server would need to track the state for multiple clients and manage their lifecycle.

Unlike the push system used in the hash-based synchronization solution, the pull-only system does not require a central forwarding hub. The push-based system relied on a static topology, where all destinations had to receive the changes made to the remote app's memory since it started. To avoid unnecessary duplicate data transmissions, a central forwarding hub was introduced. However, this hub added additional latency. With the migration protocol's P2P, pull-only algorithm, the need for a central forwarding hub is eliminated, resulting in reduced latency.

Concurrent background pulls play a crucial role in WAN environments with high latency. Without concurrent background pulls, latency accumulates rapidly, as each memory request would incur at least the round-trip time (RTT) as latency. Enabling concurrent fetching of chunks helps mitigate the impact of latency in WAN scenarios.

To support concurrent access and mitigate the bottleneck caused by global file locks, the directory backend is introduced. Unlike the file backend, which locks the entire file, the directory backend uses a chunked approach. Each chunk is represented by a separate file within the directory. This allows for individual locking and enables concurrent access to different chunks.

The directory backend maintains an internal map of locks to keep track of the chunks. When a chunk is first accessed, a new file is created specifically for that chunk. This ensures that each chunk can be independently locked and accessed concurrently. Writes to different chunks can also be performed simultaneously since each chunk is backed by its own separate file.

By using the directory backend, the bottleneck caused by global file locks in high-latency scenarios is alleviated, enabling efficient concurrent access to multiple regions.

In the directory backend, when a chunk is being read, the corresponding file is truncated to the length of one chunk. However, to prevent exhausting the maximum allowed file descriptors for a process, a check is implemented. If the maximum number of open files is exceeded, the first file in the map is closed and removed, allowing it to be reopened on a subsequent read.

Although these optimizations add some initial overhead to operations, they can significantly improve the pull speed in scenarios where the backing disk is slow or the latency is high. A benchmark comparing the performance of the file backend and the directory backend would provide insights into the efficiency of these optimizations.

In real-life deployments, the choice of RPC framework and transport protocol also plays a crucial role in performance. The mount and migration APIs in r3map are designed to be transport-independent. One option is to use the dudirekta RPC framework, which is reflection-based and allows for quick iteration on the protocol.

To use dudirekta, a wrapper struct with the necessary RPC methods is created. This struct serves as a simple interface for calling the corresponding backend (or seeder) functions, making it easy to integrate with the RPC framework.

By carefully selecting the backend implementation, optimizing file access, and leveraging a suitable RPC framework, the overall performance of r3map can be significantly improved.

The wrapper struct, which contains the necessary RPC methods, is passed as the local function struct into a registry. The registry then creates the RPC server. When a client connects using the TCP transport protocol, it is linked to the registry.

The protocol used by the RPC framework, such as TCP, is simple and straightforward. When an RPC method, such as ReadAt, is called, it is looked up in the registry using reflection and validated. The arguments, supplied as JSON, are then unmarshalled into their native types, and the method of the local wrapper struct is invoked in a new goroutine.

One important feature of this RPC framework is its support for concurrent RPC calls. Many simple RPC frameworks only allow one RPC call at a time due to the lack of demultiplexing. However, by implementing concurrent RPC support, this framework avoids the performance issues observed in frameworks like dRPC, which do not support concurrent RPCs.

To use an RPC backend on the destination side, the remote representation of the wrapper struct is used. This allows for seamless communication between the source and destination during migration or mount operations.

By utilizing the dudirekta RPC framework and its concurrent RPC support, efficient communication between the client and the server is achieved, contributing to improved performance and scalability in r3map.

On the destination site, the remote representation's fields are iterated over, and functions are created to marshal and unmarshal the function calls into the dudirekta JSON protocol. The arguments are marshalled into JSON, and a unique call ID is generated. The remote then sends a response message containing the unique call ID, and the arguments are unmarshalled and returned.

This approach makes both calling and defining RPCs transparent to the user. However, due to some limitations in dudirekta, such as passing slices as copies instead of references and the requirement for context to be provided, the resulting remote struct cannot be used directly.

To work around these limitations, the standard go-nbd backend interface is implemented for the remote representation. This creates a universally reusable and generic RPC backend wrapper, enabling seamless communication between the source and destination in the migration or mount process.

Similarly, the same backend implementation is done for the seeder protocol, allowing for consistent RPC communication between the seeder and the leecher during the migration process.

By leveraging the dudirekta RPC framework and implementing the necessary backend interfaces, the r3map system achieves interoperability and enables efficient communication between different components of the system.

While dudirekta serves as a good reference implementation of a basic RPC protocol, it may not scale particularly well due to certain design aspects. One aspect is the use of JSON(L) as the wire format, which, although simple and easy to analyze, can be slow when it comes to marshaling and unmarshaling data.

Another aspect is the support for defining functions on both the client and the server. While this feature is useful for implementing protocols like pre-copy, where the source pushes chunks to the destination, it deviates from the traditional RPC model where only the server exposes functions and the client calls them. This introduces complexities, especially in WAN scenarios where authentication and dialability between client and server are involved.

Dudirekta addresses this by making the protocol itself bi-directional, allowing both the client and server to define and call functions. This enables the implementation of protocols like pre-copy without the need for the destination to be directly dialable from the source. However, this design choice also comes with trade-offs, such as the inability to use connection pooling, as each client dialing the server would be treated as a separate connection.

Overall, while dudirekta provides a useful starting point for implementing RPC communication, it may not be the most scalable solution for large-scale deployments. Alternative RPC frameworks or protocols that optimize for performance and scalability may be more suitable in such cases.

Switching from dudirekta to gRPC can offer several benefits in terms of performance, protocol definition, and language support. gRPC is a high-performance RPC framework based on Protocol Buffers (protobuf), which provides a well-defined protocol that can be specified using the proto3 DSL.

With gRPC, we can define the wire protocol using the protobuf DSL, which offers advantages such as specifying the order of each field in the resulting wire protocol. This allows for evolving the protocol over time without breaking backward compatibility, which is crucial for the long-term maintainability of r3map. It also enables easier porting to other languages with less overhead, such as Rust, by reusing the existing wire protocol definition.

One significant advantage of gRPC is its extensive language support. While dudirekta currently only supports Go and JavaScript, gRPC supports a wide range of programming languages, making it easier to integrate r3map with different language ecosystems.

Additionally, gRPC provides efficient and optimized communication using Protocol Buffers, which can enhance the overall performance of the RPC communication.

Overall, adopting gRPC as the RPC framework for r3map can bring performance improvements, protocol evolution capabilities, and broader language support, paving the way for future enhancements and portability.

Using gRPC as the RPC framework for r3map brings several advantages, including support for streaming RPCs, authentication, authorization, and encryption. Streaming RPCs, although not currently used in the r3map protocol, can be beneficial for implementing a pre-copy migration API with pushes, similar to how dudirekta exposes RPCs from the client.

gRPC's support for authentication, authorization, and encryption makes it suitable for secure WAN migration or mount scenarios. The use of Protocol Buffers as the wire format eliminates the need for encoding chunks, as they can be sent as raw bytes. This improves efficiency compared to using JSON, which requires base64 encoding.

Being based on HTTP/2, gRPC can leverage existing load balancing tooling and infrastructure commonly used in the web context. This adds further benefits for WAN deployments.

To integrate gRPC into r3map, the protocol is defined using the protobuf DSL, and the generated bindings are used to implement the backend interface. The dudirekta wrapper struct can serve as the abstraction layer, allowing for easy migration from the dudirekta-based implementation. Unlike dudirekta, gRPC provides support for concurrent RPCs and connection pooling, enabling efficient handling of concurrent requests.

Overall, adopting gRPC offers features and capabilities that align well with the requirements of secure WAN migration or mount scenarios, while also providing additional benefits such as streaming RPCs, improved wire format efficiency, and compatibility with existing tooling.

Connection pooling is an important optimization technique that can improve the efficiency of RPC frameworks. With gRPC, it is possible to reuse existing connections for RPCs or create new ones as needed, enabling parallel requests and reducing overhead. This enhances the performance of concurrent operations.

While gRPC offers numerous benefits, including support for authentication, authorization, and streaming RPCs, it may not always be the fastest serialization framework, especially when dealing with large data chunks. In scenarios where high throughput is possible, this can become a bottleneck. To address this, fRPC can be considered as an alternative RPC library.

fRPC is known to be significantly faster than gRPC, offering 2-4x performance improvements, particularly in terms of throughput. It provides similar functionality to gRPC, supporting multiplexing and connection pooling. The use of the same proto3 DSL makes it easy to replace gRPC with fRPC in r3map.

When choosing the RPC protocol for r3map, factors such as throughput and latency are crucial, as they directly impact the maximum acceptable downtime for migrations or the initial latency for mounts. By using fRPC as a drop-in replacement for gRPC, r3map can leverage the performance benefits offered by fRPC while maintaining compatibility with the existing codebase.

Overall, fRPC provides a compelling option for achieving high-performance RPC communication in r3map, offering significant performance gains compared to gRPC, especially in scenarios where high throughput is critical.

When considering the scalability and performance of RPC frameworks, it's important to assess their ability to handle concurrent requests, particularly in scenarios involving high-latency networks. By scaling the number of pull workers, the impact on latency and throughput can be evaluated.

A benchmark can be conducted to measure the effect of tuning the number of push/pull workers on latency and throughput for the three backends: dudirekta, gRPC, and fRPC. The goal is to observe how each framework responds to increasing the number of workers, specifically in terms of reducing the initial latency for retrieving the first set of chunks and improving overall throughput.

While backends like dudirekta, gRPC, and fRPC provide a way to access a remote backend and can offer authentication and authorization mechanisms, there are cases where direct access to a remote backend without additional indirection is beneficial. This is particularly relevant for the mount API, where a remote backend acts as a remote random-access storage device.

Redis, an in-memory key-value store with network access, is an example of an existing system that provides remote access to data. Leveraging protocols like Redis can be advantageous when implementing the mount API, as they offer well-established methods for accessing remote resources over a network. This allows for seamless integration with the mount functionality and enables efficient data retrieval and storage operations.

When utilizing Redis as a backend for the mount API, chunk offsets can be mapped to keys within the key-value store. Since Redis supports bytes as valid key types, the chunks themselves can be stored directly in the Redis KV store. This simplifies the storage process and allows for efficient retrieval of chunks using their corresponding offsets.

Redis is particularly advantageous due to its efficient server-side locking mechanism, making it well-suited for high-throughput scenarios. Authentication and multi-tenancy can also be handled using the Redis protocol, either by utilizing multiple databases or using a prefix to differentiate between different users or resources.

The Redis backend offers fast read/write speeds, thanks to its optimized protocol and serialization. However, when deploying the system to the public internet, it may not be the most ideal choice due to potential security and scalability considerations.

Alternatively, the S3 backend is well-suited for mapping public information, such as media assets, binaries, or large read-only filesystems, into memory. Originally provided as an object storage service by AWS, S3 has become a widely adopted standard for accessing blobs, with open-source implementations like Minio available.

Similar to how individual files were used in the previous approach, the S3 backend utilizes one S3 object per chunk for storage. Since S3 is based on HTTP, chunking is required since it does not support updates to part of a file. This approach allows for seamless integration with S3 and efficient retrieval of chunks when needed.

When using a NoSQL server like Cassandra as a backend option, it offers a proof-of-concept scenario to demonstrate the flexibility and versatility of mapping a database to a memory region. This approach can be useful for accessing the content of a remote database without relying on a specific client.

The ReadAt and WriteAt operations in the Cassandra backend are implemented using Cassandra's query language. Locking individual keys can be handled by the database server itself. As Cassandra stores data on disk, it can be used for persistent data storage, unlike Redis, which primarily operates in-memory.

To use Cassandra as a backend, migrations need to be applied to create the required tables and structures. These migrations are responsible for setting up the necessary database schema and configurations.

Benchmarking these three backends, Redis, S3, and Cassandra, on both localhost and remote hosts can provide insights into their performance and suitability for different use cases.

Additionally, it is important to consider the overhead of managed mounts. Managed mounts, as well as migration scenarios, may introduce additional performance costs compared to direct mounts. Evaluating the performance differences between these approaches can help identify any potential optimizations needed.

Benchmarking the latency and throughput of all the backends, including Redis, S3, and Cassandra, in both localhost and realistic latency and throughput scenarios is crucial. This will provide insights into the performance characteristics of these backends and help evaluate the effectiveness of managed mounts compared to direct mounts.

In the case of localhosts, where the latency between the local and remote backends is very low, using managed mounts may result in potential duplicate I/O operations. This means that the benefits of background pulls and parallel chunk fetching may not offset the cost of duplicate I/O when compared to directly reading from the remote backend. Therefore, direct mounts might outperform managed mounts in such scenarios.

However, in realistic latency and throughput scenarios, where latency becomes higher or slow local backends are utilized (e.g., slow disks or memory), the ability to perform background pulls and parallel chunk fetching can help offset the cost of duplicate I/O operations. In these cases, managed mounts may prove to be more efficient and provide better overall performance.

Conducting comprehensive benchmarks that capture latency, throughput, and various scenarios will provide a clearer understanding of the trade-offs between managed mounts and direct mounts in different deployment environments.

9 Case Studies

ram-dl is a tech demo and experiment built on top of the r3map framework. It utilizes the fRPC backend to expand local system memory by allowing the mounting of a remote system's RAM locally. The primary purpose of ram-dl is to enable the inspection of a remote system's memory contents.

The implementation of ram-dl is based on the direct mount API provided by r3map. It leverages utilities like mkswap, swapon, and swapoff to enable paging out to the block device exposed by the direct mount API.

ram-ul, on the other hand, is a companion tool that facilitates "uploading" RAM by exposing memory, file, or directory-backed files over fRPC. Together, ram-ul and ram-dl demonstrate how r3map can be used for unique use cases, such as accessing real, remote RAM or Swap.

Although ram-dl and ram-ul are not intended for real-world use cases, they serve as demonstrations of the capabilities of r3map. The projects are designed to be lightweight, with the entire codebase comprising only around 300 lines of code, including backend configuration, flags, and other necessary components.

Another interesting use case of r3map is tapisk, which is similar to STFS (Storage Transactional File System). tapisk exhibits high read/write backend latency, often in the range of seconds to even up to 90 seconds, due to seeking. However, since the access pattern is linear with no random reads, tapisk can greatly benefit from asynchronous writes provided by managed mounts. Additionally, fast storage can be employed as a caching layer to further optimize performance.

Overall, ram-dl, ram-ul, and tapisk showcase the versatility and potential of the r3map framework for various specialized use cases.

In tapisk, the backend is linear, meaning that only one read or write operation can be performed at a time. However, when using a local backend, writes are automatically de-duplicated, and both reads and writes can be asynchronous and concurrent.

For tapisk's tape drive backend, chunking is performed on tape drive records and blocks. Due to the linear nature of tape drives, only one concurrent reader or writer makes sense. Syncing intervals to and from the tape can be set to minutes or even longer to improve efficiency. This helps reduce the need for frequent seeking to different positions on the tape, as long, connected write intervals are preferred.

The tape backend also enables access to large amounts of linear data, such as terabytes of data stored on tape, as if it were in memory. This allows comfortable access to large datasets or backups. Additionally, if a file system like EXT4 is written to the tape, the tape can be mounted as a random-access device, further enhancing its usability.

The tape-specific backend in tapisk is implemented as the Backend interface from the go-nbd library. This backend can be reused outside of tapisk, for example, as a backend for ram-dl.

To maintain the index of the tape contents, tapisk uses bbolt DB. This index maps simulated offsets to the real tape records. When reading, the tape backend first looks up the real tape record for the requested offset.

Overall, tapisk demonstrates how tape drives can be efficiently utilized to access large datasets or backups, providing a convenient and comfortable way to work with linear data.

In tapisk, seeking within the tape drive is achieved using the accelerated MTSEEK ioctl, which allows fast-forwarding to a specific record on the tape. After seeking to the desired block, the chunk is read from the tape into memory.

When writing to the tape, the drive seeks to the end of the tape unless the last operation was a write, in which case it remains at the current position. The current real tape block position is then requested, stored in the index, and the offset is written to the tape. This effectively transforms the tape into a chunked and reusable ReadWriterAt, similar to the directory backend.

By using a remote procedure call (RPC) backend, tapisk enables accessing a remote tape in the same way as a local tape, allowing for scenarios where a remote library's tape robot can be mapped to the local memory.

Tapisk showcases the versatility of the chosen approach and demonstrates its flexibility. For example, the chunking system does not need to be reimplemented, as the managed mount API can be used directly without modifications. This highlights the adaptability and reusability of the underlying concepts and APIs.

Tapisk can indeed serve as a real use case to replace LTFS (Linear Tape File System). While LTFS is a kernel module filesystem specifically designed for tape drives, it creates its own filesystem structure. In contrast, tapisk allows you to use any existing and tested filesystem on top of the generic block device, providing more flexibility.

One limitation of LTFS is its lack of support for caching, making it challenging to use for memory mapping. Tapisk overcomes this limitation by enabling caching and efficient memory mapping.

Tapisk also demonstrates its minimalistic nature. While LTFS consists of tens of thousands of lines of code, tapisk accomplishes similar functionality and more with just under 350 lines of code. This showcases the simplicity and effectiveness of the design.

Another potential use case for r3map is to create a unique mountable remote filesystem. Currently, there are two main approaches to implementing this:

1. File synchronization platforms like Google Drive and Nextcloud monitor file changes in a folder and synchronize files when they are modified. However, this approach requires storing all files locally, and if a large amount of data is involved (e.g., terabytes), selecting and managing the locally available files can become cumbersome.
2. The other approach is using r3map, where files are dynamically downloaded as they are accessed. This approach allows for efficient usage of storage space since files are only downloaded when needed. It also provides transparency to the user, as the filesystem behaves as if all files are locally available (with asynchronous syncing of writes back to the remote storage).

Both approaches have their advantages and considerations, and the choice depends on the specific requirements and constraints of the use case.

Indeed, when considering the choice between offline usage and FUSE-based solutions like s3-fuse, both options have their limitations. FUSE-based solutions often come with a performance penalty, especially for writes, as they rely on kernel communication. Offline usage can also be challenging, and some FUSE implementations may lack certain features or permissions support.

Using r3map, however, provides a solution that combines the advantages of both approaches. With r3map, you can avoid downloading all files in advance, allowing for on-demand fetching of files. Additionally, r3map offers asynchronous write-back capabilities, enabling efficient updates to the remote storage. It also functions as a fully-featured filesystem, addressing the limitations found in some FUSE implementations.

Furthermore, with r3map's managed mount API, files can be downloaded preemptively for offline access, similar to the approach used by platforms like Google Drive. By adjusting the number of pull workers, you can control the background downloading of files.

Overall, r3map offers a versatile and efficient solution that combines the benefits of on-demand file fetching, asynchronous write-back, and offline access, providing a more comprehensive filesystem solution compared to the individual downsides of alternative approaches.

Absolutely! With r3map, the combination of read caching, local backend storage, and periodic syncing of writes allows for efficient access to remote databases. This approach is particularly useful for in-memory or on-disk databases like SQLite.

By mounting the database through r3map, you can access it as if it were a local database. The advantage is that you don't have to download the entire database before using it. Instead, the database is accessed on-demand, significantly reducing the network overhead. Additionally, as reads are cached using the managed mount API, subsequent reads can be as fast as native disk reads, further enhancing the performance.

This use case showcases how r3map bridges the gap between accessing remote databases and providing a seamless local experience. It highlights the flexibility of the memory synchronization tooling and how it can be applied to synchronize various forms of state, including databases.

That's correct! The concept of using r3map to access remote databases can be extended to other file formats that typically have limitations due to technical constraints. A notable example is the MP4 format.

In the traditional approach, when someone downloads an MP4 file, they have to wait until the entire file is downloaded before they can start playing it. This is because MP4 files store metadata at the end, and generating the metadata requires encoding the video first.

By using r3map, the MP4 file can be mounted, and access to the file can be provided while it is being downloaded. The metadata can be fetched on-demand, and as chunks of the file become available, they can be streamed and played without the need to wait for the entire file to download.

This approach demonstrates the versatility of r3map and how it can be applied to various file formats that typically have limitations in terms of streamability. By leveraging the memory synchronization capabilities of r3map, it becomes possible to provide efficient access to such file formats, making them streamable and accessible while being downloaded.

In the traditional approach to downloading files, such as the MP4 format, accessing metadata before the file is fully downloaded is challenging. The metadata is typically stored at the end of the file, and generating it requires encoding the video first. Inverting the download order or modifying the file structure may not be feasible for existing files.

However, with the utilization of r3map, this limitation can be overcome. By leveraging the pull heuristic function in r3map, the metadata of an MP4 file can be pre-fetched immediately. The individual chunks of the file can then be fetched in the background or as they are accessed. This enables the streaming of an MP4 file before it has been completely downloaded, making it possible to access the metadata and other parts of the file while it is being downloaded. This approach can be applied not only to MP4 files but also to various file formats, without requiring any modifications to the video/audio player or the format itself.

In addition to multimedia applications, the concept of in-place streaming facilitated by r3map has potential use cases in the gaming industry. Currently, when downloading a game, all assets are typically downloaded before the game becomes playable, which can result in large download sizes, often exceeding hundreds of gigabytes for new AAA games. However, not all assets are required to start the game. By employing r3map, only essential assets such as the launcher, UI libraries, or the first level can be fetched initially, allowing the game to be playable while other assets are fetched in the background or on-demand. This significantly reduces the initial download size and enables a more efficient and interactive gaming experience.

The concept of in-place streaming, made possible by r3map, showcases the versatility of this approach across various domains. It enables the streaming of files before they are fully downloaded, provides on-demand access to specific parts of the file, and eliminates the need for substantial modifications to existing formats or applications.

In the realm of gaming, it is possible to design a game engine that fetches game assets from a remote source as they are required. However, this approach would necessitate substantial changes to the way new games are developed and would not be applicable to existing games. Additionally, playing games directly from a network drive or a FUSE-based system is unrealistic due to performance limitations.

By leveraging the managed mount API offered by r3map, it becomes feasible to mitigate the performance limitations associated with fetching game assets on-demand. The background pull system in r3map allows for efficient retrieval of already pulled chunks, enabling reads from these chunks to be nearly as fast as native disk reads. Through careful analysis of the data access pattern of an existing game, a pull heuristic function can be created. This function can preemptively fetch the necessary game assets, prioritizing those required for immediate gameplay, and minimizing latency spikes.

To further optimize the gaming experience, the chunk availability hooks provided by r3map can be utilized. For instance, a game could be paused on a loading screen until a certain percentage of the required assets have been locally pulled and are readily available. This approach ensures that latency spikes due to missing assets are minimized while still enabling faster game startup times.

The same concept of in-place streaming and on-demand retrieval can be applied beyond gaming, extending to the launch of any application. In many programming languages, the complete loading of a binary or script into memory is not a prerequisite for execution. However, without significant modifications to language interpreters or virtual machines, accessing files from a file system remains the primary means of loading programs. By leveraging r3map's capabilities, it becomes possible to streamline the launch process by fetching the required portions of the application on-demand, leading to improved application startup times and enhanced user experiences.

With the managed mount API, it becomes possible to add streaming execution support to any interpreter or virtual machine. This can be achieved by utilizing the managed mount as a filesystem to stream multiple binaries, scripts, or libraries, or by directly accessing the block device as a file or memory-mapped resource. By leveraging these capabilities, the synchronization of application state becomes more accessible.

Traditionally, synchronizing application state across multiple systems involves developing custom protocols and relying on complex sync mechanisms. In some cases, database solutions like Firebase may be used for synchronization, but they often have limitations in terms of supported data structures and require the use of specific APIs for syncing.

However, if data structures can be represented as byte arrays ([]byte), it becomes possible to allocate them from a block device. These data structures can then be sent, received, or mounted using the managed mount, direct mount, or migration APIs. This opens up interesting possibilities for various use cases.

For instance, consider a TODO app where the TODO list is mounted as a byte slice from a remote server by default. By leveraging the managed mount API, the TODO app can seamlessly access and synchronize the TODO list across multiple systems. This eliminates the need for complex sync protocols and allows for efficient data transfer and synchronization without relying on third-party solutions like databases.

In essence, the ability to mount and migrate any resource using the r3map framework enables streamlined and efficient synchronization of data structures, empowering developers to build robust and scalable applications.

The migration use case presents an interesting opportunity within the r3map framework. With the pluggable authentication and the ability to leverage different backends, scaling can be achieved effectively.

In the context of migrating a TODO app, the migration process can involve streaming the byte slice from the source server to the destination server. By utilizing the background pull mechanism, the byte slice can be fetched in real-time as the app accesses it, while also pre-fetching the remaining data.

Modifications made to the TODO list in memory are automatically written to the local backend. The asynchronous push algorithm then takes these changes and synchronizes them with the remote backend as necessary.

By using a persistent local backend, this setup can even survive network outages, ensuring data integrity. However, it's important to note that the current implementation only supports a single user, and locking mechanisms are not provided.

To make the in-memory representation universal across different CPU architectures, compiling the application to WebAssembly (Wasm) and pointing the Wasm VM's linear memory to the NBD device can be a viable solution. This approach allows for storing the entire app state in the remote Wasm environment without requiring modifications to the app itself.

While the smart mount feature, where the remote struct remains transparent to the user, is interesting, the migration use case holds significant potential. It enables seamless transfer of application state between different systems, ensuring continuity and flexibility in distributed environments.

One compelling use case for r3map is enabling seamless migration of app state between different devices. For example, consider a scenario where a user has the TODO app running on their phone but wishes to continue editing the TODO list on their desktop. Using the migration API provided by r3map, the app's state can be migrated from one device to another.

In this particular use case, the migration process could be automatically triggered when the phone comes physically close to the desktop. Since this would typically occur within a local area network (LAN) environment, the migration would be fast and enable interesting hand-off scenarios.

The migration API in r3map offers hooks and a protocol that ensures the app is closed on the source device (phone) before being resumed on the destination device (desktop). This helps minimize the risk of state corruption during the migration process, as the remote state is synchronized and up-to-date when the app is resumed on the desktop.

Migrating the TODO list or any internal data structure of the app is just one possibility. By incorporating a toolkit like Apache Arrow, a universal format for representing memory, r3map can facilitate the migration of various types of data structures and enable flexible migration of app state across different devices.

By adding a layer of indirection through r3map, developers can explore numerous possibilities for seamless app state migration and leverage the benefits of remote memory access in a variety of use cases.

r3map offers a wide range of possibilities for state migration, including the ability to migrate and synchronize the memory of a Wasm virtual machine (VM). By leveraging r3map, any Wasm application, along with its binary, mounted WASI filesystem, and other associated resources, can be seamlessly transferred between hosts.

The flexibility of r3map extends beyond Wasm VMs and enables the migration of state for various systems. For example, it can be used for live migration of running VMs between hosts, regardless of the hypervisor used, as long as the hypervisor supports mapping a VM's memory to a block device. This concept can also be applied to other VMs, such as the JVM, a JavaScript VM in a browser, or even an entire process's memory space, provided the processor architectures align.

Moreover, by utilizing the mount API, r3map enables the concept of VM snapshots for virtually any application. With the preemptive pull mechanism in place, restoring a Wasm VM, for instance, outside of a migration becomes a swift operation. Essentially, r3map facilitates VM memory migration and snapshots for diverse types of state, without imposing specific requirements on the data structures other than their ultimate representation as a []byte.

This capability is particularly intriguing as it opens doors to seamless state migration and snapshotting across a broad spectrum of applications, encompassing processes, VMs, and other systems, with the underlying principle of representing state as []byte, which is inherent to any form of state at a process or VM level.

10 Conclusion

In conclusion, we have explored various synchronization options and compared their ease of implementation, CPU load, and network traffic. We have discussed the limitations of traditional approaches, such as file synchronization, database synchronization, and custom sync protocols. These methods often require complex sync protocols, suffer from high network traffic, and lack universality.

We then introduced r3map, a novel solution that leverages memory as a universal access format for synchronization. By utilizing the managed mount API, r3map allows for efficient and seamless synchronization of state across different systems. The ability to mount remote memory regions, migrate state between hosts, and stream data on-demand opens up new possibilities for efficient synchronization and access to resources.

This approach offers several advantages, including reduced network traffic, lower CPU load, and the ability to handle various types of state with a simple []byte representation. By combining the benefits of memory synchronization, direct mount APIs, and migration capabilities, r3map provides a versatile and efficient solution for state synchronization.

In terms of further research, one potential recommendation is to explore the concept of ublk, which could extend the capabilities of r3map to support the synchronization of block devices. This would enable the synchronization of entire file systems, disks, or other block-based storage systems, further expanding the scope of r3map's applicability.

Overall, r3map presents a promising approach to state synchronization, leveraging memory as a universal access format. Its flexibility, efficiency, and ease of use make it a compelling option for various synchronization use cases, paving the way for future advancements in this field.