CMPSC 311 Technical Report

`mdadm` Linear Device Project (Basic Functionality, Writes and Testing, Caching and Networking)

Ali Bader AlNaseeb
Aba6173@psu.edu

The Pennsylvania State University

Computer Science Technical Report
April 30, 2023

`mdadm` Linear Device Project (Basic Functionality, Writes and Testing, Caching and Networking)

Ali Bader AlNaseeb
Aba6173@psu.edu

Introduction

The marketing team of a cryptocurrency startup recently acquired 16 military-grade hard disks. These hard disks are designed to be nuclear bomb-proof and provide physical security. However, the disk company provided a JBOD storage architecture with a user manual and a device driver that lacks software investment. This technical report evaluates the feasibility of using the newly acquired hard disks in the cryptocurrency startup.

In this report, we will discuss the specifications of the hard disks, the JBOD operation format, and the device driver functionality. We will also examine the potential risks and benefits of using the military-grade hard disks in the startup's infrastructure.

The report concludes with a recommendation for further evaluation and testing before implementing the hard disks in the startup's infrastructure. It is essential to consider the risks and benefits of using the hard disks carefully to ensure that the startup's data storage and retrieval capabilities are not compromised.

Hard Disk Specifications

Each of the 16 military-grade hard disks has 256 blocks with 256 bytes, and collectively they have a combined capacity of 1 MB. The disk company provided a device driver with a single function that can be used to control the disks. The hard disks have an impressive capacity that can store a significant amount of cryptocurrency wallets. However, the device driver's limitations could potentially hinder the performance of the hard disks.

JBOD Operation Format

The JBOD operation format, as described in Table 1 of the user manual, was investigated. The format includes a command to be executed by the JBOD, an ID of the disk to perform the operation on, unused bits for now, and a block address within the disk. The JBOD operation format is crucial to understand because it determines how the disks are accessed and utilized. The investigation of the JBOD operation format is necessary to ensure that the hard disks function optimally in the startup's infrastructure.

Device Driver Functionality

The device driver provided by the disk company was evaluated. It was found that the driver only has a single function that can be used to control the hard disks. The limitations of the driver could potentially hinder the performance of the hard disks. The device driver's limitations could result in slower data storage and retrieval processes, which could be a significant obstacle to the startup's operations.

Risks and Benefits

The potential risks and benefits of using the military-grade hard disks in the startup's infrastructure were discussed. While the hard disks provide physical security, their limitations in terms of software investment and device driver functionality could pose a risk to the startup's data storage and retrieval capabilities. The risks associated with using the hard disks need to be carefully considered before implementing them in the startup's infrastructure. On the other hand, the benefits of using military-grade hard disks include the assurance of physical security, which is essential when dealing with sensitive data such as cryptocurrency wallets.

Basic Functionality(First Phase)

Bits	Width	Field	Description
26-31	6	Command	This is the command to be executed by JBOD.
22-25	4	DiskID	This is the ID of the disk to perform operation on.
8-21	14	Reserved	Unused bits (for now).
0-7	8	BlockID	Block address within the disk.

Table 1: JBOD operation format

`int jbod_operation(uint32_t op, uint8_t *block)`: This function is a crucial part of the JBOD driver and is used to execute various operations on the JBOD. It accepts an operation through the `op` parameter and a pointer to a buffer. The `op` parameter specifies the type of operation to be performed, and it follows a specific format described in Table 1. The `block` parameter is a pointer to a buffer that is used for input/output of data. The `jbod_operation` function can be called with different commands, which are defined as a C enum type in the header file. These commands include JBOD_MOUNT, JBOD_UNMOUNT, JBOD_SEEK_TO_DISK, JBOD_SEEK_TO_BLOCK, JBOD_READ_BLOCK, and JBOD_WRITE_BLOCK. Each command has specific functionality and input/output requirements. For example, the `JBOD_MOUNT` command

mounts all disks in the JBOD and makes them ready to serve commands. The function returns 0 on success and -1 on failure.

`int mdadm_mount(void);`

This function is used to mount the linear device, which combines all the disks in the JBOD into a single logical volume. The linear device allows the mdadm user to run read and operations on the linear address space using a single device file. The `mdadm_mount` function should be called before any other operations are performed on the linear device. It should return 1 on success and -1 on failure. If the function is called a second time without calling `mdadm_unmount` in between, it should fail.

`int mdadm_unmount(void);`

This function is used to unmount the linear device, which disassociates all disks in the JBOD from the single logical volume. All commands to the linear device will fail after unmounting. The `mdadm_unmount` function should be called after all operations on the linear device are completed. It should return 1 on success and -1 on failure. If the function is called a second time without calling `mdadm_mount` in between, it should fail.

`int mdadm_read(uint32_t addr, uint32_t len, uint8_t *buf);`

This function is used to read `len` bytes of data into the buffer pointed by `buf`, starting at the linear address specified by `addr`. The `len` parameter must not exceed 1024 bytes, and any read operation outside the bounds of the linear device should fail. The `mdadm_read` function is commonly used to read data from the linear device, and the data can then be processed or displayed as required.

This code is written in C language.

The first three lines include three libraries that are necessary for the code to function properly. `stdio.h` is used for input and output operations, `string.h` is used for string manipulation, and `assert.h` is used to test for logical errors in the code.

The next three lines include three header files that are specific to the project. `mdadm.h`, `jbod.h`, and `net.h` are used to define structs, functions, and constants unique to the project.

The `JBOD` struct has three fields: `currentBlockID`, `currentDiskID`, and `block_pointer`. This struct is used to keep track of the current position of the system during read and write operations.

The `jbod` variable is an instance of the `JBOD` struct.

The `mount_status` variable is used to keep track of whether the system is currently mounted or unmounted. A value of 1 indicates that the system is unmounted, while a value of 2 indicates that the system is mounted.

The `block_constructor` function takes four arguments: `BlockID`, `Reserved`, `Disk_ID`, and `Command`. It returns a value that is constructed using the bitwise OR operator.

The `mdadm_mount` function is used to mount the system. It first checks if the system is currently unmounted. If it is, it calls `jbod_client_operation` with a constructed block operation to mount the system. It then updates the `mount_status` and `JBOD` struct to reflect the mounted state. If the system is already mounted, the function returns -1 to indicate failure.

The `mdadm_unmount` function is used to unmount the system. It first checks if the system is currently mounted. If it is, it calls `jbod_client_operation` with a constructed block operation to unmount the system. It then updates the `mount_status` to reflect the unmounted state. If the system is already unmounted, the function returns -1 to indicate failure.

The `seek` function is used to move the system to a new position. It first checks if the system is mounted. If it is not, the function returns -1 to indicate failure. It then constructs two block operations: one to seek to a new block and one to seek to a new disk. It calls `jbod_client_operation` with both block operations to move the system to the new position. It updates the `JBOD` struct to reflect the new position.

The `mdadm_read` function is used to read data from the system. It first checks if the system is mounted and if the length of data to be read is valid. It then constructs a local buffer to store the data. It calculates the current disk ID, block ID, and block pointer using the given address. It then reads data from the system block by block, updating the block pointer and length with each iteration. The function returns the total length of data read.

The `mdadm_write` function is used to write data to the system. It first checks if the system is mounted and if the length of data to be written is valid. It then constructs a local buffer to store the data. It calculates the current disk ID, block ID, and block pointer using the given address. It then writes data to the system block by block, updating the block pointer and length with each iteration. The function returns the total length of data written.

The code snippet provided contains C code for a basic JBOD (Just a Bunch Of Disks) storage system. The code includes the following libraries:

- stdio.h: Standard input/output library.
- string.h: Library for string manipulation functions.
- assert.h: C library for debugging purposes.

Additionally, the code includes three header files:

- mdadm.h: Header file containing function prototypes for mdadm (Multiple Devices Admin) operations.

- jbod.h: Header file containing function prototypes for JBOD operations.
- net.h: Header file containing function prototypes for network operations.

The code also defines a structure, `JBOD`, which has three attributes: `currentBlockID`, `currentDiskID`, and `block_pointer`. The code then initializes an instance of the `JBOD` structure, `jbod`, and sets the `mount_status` variable to 1.

The `block_constructor` function takes in four parameters and returns a 32-bit integer constructed from the input parameters.

The code defines four functions:

1. `mdadm_mount`: This function mounts the JBOD system by calling `jbod_client_operation` with a constructed block operation. If the system is already mounted, the function returns -1. Otherwise, it updates the mount status and the `jbod` struct to reflect the mounted state and returns 1 to indicate success.
2. `mdadm_unmount`: This function unmounts the JBOD system by calling `jbod_client_operation` with a constructed block operation. If the system is already unmounted, the function returns -1. Otherwise, it updates the mount status to reflect the unmounted state and returns 1 to indicate success.
3. `seek`: This function takes in two parameters, `newBlockID` and `newDiskID`, and seeks to the specified block and disk ID. If the system is unmounted, the function returns -1. Otherwise, it constructs two block operations (`new_Block_op` and `new_Disk_op`) and calls `jbod_client_operation` with each operation. If either operation fails, the function returns -1. Otherwise, it updates the `jbod` struct with the new block and disk IDs and returns 1 to indicate success.
4. `mdadm_read`: This function takes in three parameters, `addr`, `len`, and `buf`, and reads data from the JBOD system starting at the specified address (`addr`) and with the specified length (`len`). If the system is unmounted, the function returns -1. If the length is greater than 1024 or less than 0, the function returns -1. If the buffer is null and the length is not 0, the function returns -1. If the specified address and length exceed the size of the JBOD system, the function returns -1. Otherwise, the function reads data from the JBOD system and copies it to the buffer. The function updates the `jbod` struct with the new block and disk IDs and returns the length.
5. `mdadm_write`: This function takes in three parameters, `addr`, `len`, and `buf`, and writes data to the JBOD system starting at the specified address (`addr`) and with the specified length (`len`). If the system is unmounted, the function returns -1. If the length is greater than 1024 or less than 0, the function returns -1. If the buffer is null and the length is not 0, the function returns -1. If the specified address and length exceed the size of the JBOD system, the function returns -1. Otherwise, the function writes data to the JBOD system and updates the `jbod` struct with the new block and disk IDs. The function returns the length.

Caching (Third Phase)

Introduction

mdadm is a software package that manages multiple devices and provides redundancy and performance across them. The software engineers are testing the storage system for building a secure crypto wallet on top of it. To improve the performance of the storage system, a block cache is required to reduce request latency. In this technical report, we will discuss the implementation of a block cache in mdadm and its integration with mdadm_read and mdadm_write functions.

Overview of Caching

Caching is a method of reducing request latency by storing frequently used data in a faster storage medium than the main storage. In mdadm, the block cache will store key-value pairs, where the key is the tuple consisting of disk number and block number that identify a specific block in JBOD, and the value is the contents of the block. When mdadm_read is called, it will first check if the block specified by the user is in the cache. If it is, the block will be copied from the cache without the need for a slow JBOD_READ_BLOCK call to JBOD. If the block is not in the cache, it will be read from JBOD and inserted into the cache for faster access in the future.

Implementation of Block Cache

The cache is implemented as a separate module and then integrated with mdadm_read and mdadm_write calls. To manage the cache entries, cache functions such as cache_create, cache_destroy, cache_lookup, cache_insert, cache_update, and cache_enabled are used. The cache_entry_t struct is defined in cache.h, which stores the cache entries. The valid field indicates whether the cache entry is valid, the disk_num and block_num fields identify the block held by the entry, and the block field holds the data of the corresponding block. The access_time field stores when the cache element was last accessed. The cache.c file contains predefined variables such as cache, cache_size, clock, num_queries, and num_hits. The cache_create function dynamically allocates space for cache entries and should store the address in the cache global variable. The cache_destroy function frees the dynamically allocated space for cache, and should set cache to NULL and cache_size to zero. The cache_lookup function looks up the block identified by disk_num and block_num in the cache. If the block is found, it is copied into buf and the num_queries and num_hits variables are incremented. The cache_insert function inserts the block identified by disk_num and block_num into the cache and copies buf to the corresponding cache entry. If the cache is full, an entry is overwritten according to the LRU policy using data from the insert operation. The cache_update function updates the block content with the new data in buf if the entry exists in cache. The cache_enabled function returns true if cache is enabled.

Implementation Strategy

The strategy for the implementation of the block cache in mdadm involves passing all the tester unit tests after implementing the functions in cache.c. Once the tests are passed, the cache is incorporated into mdadm_read and mdadm_write functions. Caching is implemented in mdadm_write as well, using write-through caching policy. The implementation is tested on the trace files to evaluate the performance. The

cost is a new metric that measures the effectiveness of the cache, which is calculated based on the number of operations executed. The tester takes a cache size when used with a workload file and prints the cost and hit rate at the end. The hit rate is computed based on num_queries and num_hits variables. The correctness of the mdadm implementation is ensured by comparing the outputs.

Adding Networking Support to mdadm(Last Phase)

Objective

The objective of this project is to add networking support to the mdadm implementation to increase the flexibility of the system. Specifically, the company wants to enable the execution of JBOD operations over the network. The goal is to allow JBOD operations to be executed on a JBOD server located anywhere on the internet.

Protocol

The JBOD protocol is composed of two messages - the JBOD request message and the JBOD response message. The JBOD request message is sent from the client program to the JBOD server and contains an opcode and a buffer when needed. The JBOD response message is sent from the JBOD server to the client program and contains an opcode and a buffer when needed. Both messages use the same format with fields indicating the size of the packet, the opcode for the JBOD operation, return code from the JBOD operation, and buffer.

Implementation

To implement networking support, the `jbod_connect` and `jbod_disconnect` functions will be implemented in the `net.c` file. These functions will allow the client to connect and disconnect from the JBOD server. In addition, all calls to `jbod_operation` will be replaced with `jbod_client_operation`, which will send JBOD commands over the network to the JBOD server.

The `jbod_connect` function will establish a connection to `JBOD_SERVER` at port `JBOD_PORT`. The `jbod_disconnect` function will close the connection to the JBOD server. This implementation will ensure that the code works as expected by running the `jbod_server` in one terminal while the tester with the workload file will be run in another terminal. The tester will issue requests to the JBOD server to verify that the networking support has been successfully implemented.

Benefits

The addition of networking support to the mdadm implementation will allow the company to increase the flexibility of their system. The system will be able to execute JBOD operations on a server located anywhere on the internet. As the company scales, they plan to add multiple JBOD

systems to their data center. Having networking support in mdadm will enable the company to avoid downtime in case a JBOD system malfunctions by switching to another JBOD system on the fly. This will reduce the load on their JBOD systems and increase the flexibility of their system.

By adding networking support to the mdadm implementation, the company will be able to reduce the load on their JBOD systems. This is because the system will be able to execute JBOD operations on a server located anywhere on the internet, which means that the load will be distributed among multiple servers. Additionally, the company will be able to avoid downtime in case a JBOD system malfunctions. If one JBOD system malfunctions, mdadm will automatically switch to another JBOD system on the fly. This will ensure that the system is always up and running, and there is no downtime.

Furthermore, the addition of networking support to the mdadm implementation will increase the flexibility of the system. The system will be able to execute JBOD operations on a server located anywhere on the internet, which means that the company can add multiple JBOD systems to their data center as they scale. This will enable the company to handle more data and provide better service to their customers.

In conclusion, the addition of networking support to the mdadm implementation will provide several benefits to the company. It will increase the flexibility of the system, reduce the load on their JBOD systems, and ensure that the system is always up and running.

```
int cli_sd = -1, lengthSize = 2, opSize = 4, retSize = 2;
```

This line declares and initializes four integer variables: `cli_sd`, `lengthSize`, `opSize`, and `retSize`.

```
static bool nread(int fd, int len, uint8_t *buf)
```

This line defines a function `nread` that takes three arguments: an integer file descriptor `fd`, an integer `len`, and a pointer to an unsigned 8-bit integer `buf`. This function returns a Boolean value.

```
for (int bytesRead = 0; bytesRead < len; bytesRead += loopResult)
{
loopResult = read(fd, buf + bytesRead, len - bytesRead);
if (loopResult < 0)
return false;
                                        }
```

This block of code is inside the `nread` function. It is a `for` loop that reads data from the file descriptor `fd` and stores it into the buffer pointed to by `buf`. The loop continues until `len` bytes

have been read. The variable `bytesRead` keeps track of how many bytes have been read so far, and `loopResult` is the number of bytes read during each iteration of the loop. If `read()` returns a value less than 0, the function returns `false`.

```
static bool nwrite(int fd, int len, uint8_t *buf)
```

This line defines a function `nwrite` that takes three arguments: an integer file descriptor `fd`, an integer `len`, and a pointer to an unsigned 8-bit integer `buf`. This function returns a Boolean value.

```
int bytesWritten = 0;
while (bytesWritten < len && (bytesWritten += write(fd, buf + bytesWritten,
len - bytesWritten)) > 0)
;
return bytesWritten == len;
```

This block of code is inside the `nwrite` function. It is a `while` loop that writes data from the buffer pointed to by `buf` to the file descriptor `fd`. The loop continues until `len` bytes have been written. The variable `bytesWritten` keeps track of how many bytes have been written so far. If `write()` returns a value less than or equal to 0, the loop terminates. The function returns a Boolean value indicating whether all `len` bytes were successfully written.

```
static bool recv_packet(int sd, uint32_t *op, uint16_t *ret, uint8_t *block)
```

This line defines a function `recv_packet` that takes four arguments: an integer socket descriptor `sd`, a pointer to an unsigned 32-bit integer `op`, a pointer to an unsigned 16-bit integer `ret`, and a pointer to an unsigned 8-bit integer `block`. This function returns a Boolean value.

```
uint8_t headbuff[HEADER_LEN];
if (!nread(sd, HEADER_LEN, headbuff))
return false;
```

This block of code is inside the `recv_packet` function. It declares an array `headbuff` of unsigned 8-bit integers with a length of `HEADER_LEN`. It then calls the `nread()` function to read `HEADER_LEN` bytes from the socket descriptor `sd` into the `headbuff` array. If `nread()` returns `false`, the function returns `false`.

```
int headOffset = 0;
uint16_t length = ntohs(*(uint16_t *)(headbuff + headOffset));
headOffset += lengthSize;
uint32_t nOp = ntohl(*(uint32_t *)(headbuff + headOffset));
headOffset += opSize;
uint16_t nReturn = ntohs(*(uint16_t *)(headbuff + headOffset));
```

This block of code is inside the `recv_packet` function. It declares three variables: `headOffset`, `length`, `nOp`, and `nReturn`. `length` is read from the `headbuff` array using `ntohs()`, which converts the network byte order to host byte order. `nOp` is read from the `headbuff` array at an offset of `opSize` bytes from the beginning of the array, and `nReturn` is read from the `headbuff` array at an offset of `retSize` bytes from the beginning of the array. `ntohl()` is used to convert the network byte order to host byte order for `nOp`.

```
*op = nOp;
*ret = nReturn;
```

This block of code is inside the `recv_packet` function. It sets the values pointed to by the `op` and `ret` pointers to `nOp` and `nReturn`, respectively.

```
if (length > HEADER_LEN && !nread(sd, JBOD_BLOCK_SIZE, block))
return false;
return true;
```

This block of code is inside the `recv_packet` function. It checks if `length` is greater than `HEADER_LEN` and, if so, calls the `nread()` function to read `JBOD_BLOCK_SIZE` bytes from the socket descriptor `sd` into the `block` buffer. If `nread()` returns `false`, the function returns `false`. Otherwise, the function returns `true`.

```
static bool send_packet(int sd, uint32_t op, uint8_t *block)
```

This line defines a function `send_packet` that takes three arguments: an integer socket descriptor `sd`, an unsigned 32-bit integer `op`, and a pointer to an unsigned 8-bit integer `block`. This function returns a Boolean value.

```
uint8_t buff[HEADER_LEN + JBOD_BLOCK_SIZE];
uint16_t length = HEADER_LEN + ((op >> 26) == JBOD_WRITE_BLOCK ?
JBOD_BLOCK_SIZE : 0);
uint16_t nLength = htons(length);
uint32_t nOp = htonl(op);

memcpy(buff, &nLength, lengthSize);
memcpy(buff + lengthSize, &nOp, opSize);

if ((op >> 26) == JBOD_WRITE_BLOCK)
{
memcpy(buff + HEADER_LEN, block, JBOD_BLOCK_SIZE);
}

return nwrite(sd, length, buff);
```

This block of code is inside the `send_packet` function. It declares an array `buff` of unsigned 8-bit integers with a length of `HEADER_LEN + JBOD_BLOCK_SIZE`. It then calculates the length of the packet based on the value of `op` and stores it in `length`. The packet length is stored in network byte order in `nLength`. The value of `op` is stored in network byte order in `nOp`. The

header of the packet is constructed by copying `nLength` and `nOp` into `buff`. If `op` indicates that a data block is included in the packet, the data block is copied into `buff` after the header. Finally, the entire packet is written to the socket descriptor `sd` using the `nwrite()` function. The function returns the result of the `nwrite()` call.

```
bool jbod_connect(const char *ip, uint16_t port)
{
struct sockaddr_in ipv4_addr = {.sin_family = AF_INET, .sin_port =
htons(port)};
return (cli_sd = socket(PF_INET, SOCK_STREAM, 0)) != -1 && connect(cli_sd,
(struct sockaddr *)&ipv4_addr, sizeof ipv4_addr) != -1;

if (inet_aton(ip, &ipv4_addr.sin_addr) == 0 || connect(cli_sd, (const struct
sockaddr *)&ipv4_addr, sizeof(ipv4_addr)) == -1)
{
return false;
}
return true;
}
```

This block of code defines a function `jbod_connect` that takes two arguments: a string `ip` and an unsigned 16-bit integer `port`. It creates a `sockaddr_in` struct with the specified IP address and port number, and then creates a socket descriptor using the `socket()` function. Then, it tries to connect to the server using the `connect()` function. If either the `inet_aton()` or `connect()` functions return an error, the function returns `false`. Otherwise, the function returns `true`.

```
void jbod_disconnect(void)
{
close(cli_sd);
cli_sd = -1;
}
```

This line defines a function `jbod_client_operation` that takes two arguments: an unsigned 32-bit integer `op`, and a pointer to an unsigned 8-bit integer `block`. This function returns an integer.

```
uint16_t ret;
if (!send_packet(cli_sd, op, block) || !recv_packet(cli_sd, &op, &ret,
block))
{
return -1;
}
return ret;
```

This block of code is inside the `jbod_client_operation` function. It declares an unsigned 16-bit integer `ret`. It then calls the `send_packet()` function to send the packet to the server and the `recv_packet()` function to receive a response from the server. If either `send_packet()` or `recv_packet()` returns false, the function returns `-1`. Otherwise, it returns the value of `ret`.

Acknowledgment

I would like to express my sincere appreciation to Professor Aghayev for their exceptional guidance and teaching during the Spring 2023 term of CMPSC 311. Their expertise and passion for computer science have been invaluable to my learning journey. I would also like to extend my gratitude to the teaching assistants and learning assistants of this course. Their tireless efforts in providing me feedback and support have been critical in my academic success. Thank you for creating such an enriching and challenging learning environment that has allowed me to grow both academically and personally.