LAB #5 – `mdadm` Linear Device (Networking)
CMPSC311 - Introduction to Systems Programming
Spring 2025 - Prof. Suman Saha **Due date: May 2, 2025 (11:59 PM) EST**
NO EXTENSIONS OR LATE SUBMISSIONS ACCEPTED

Like all lab assignments in this class, you are prohibited from copying any content from the Internet or discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone (AI or human) in or outside of the class. Failure to abide by this requirement will result in penalty as described in our course syllabus.

Adding a cache to your `mdadm` system has significantly improved its latency and reduced the load on the JBOD. Before you finish your internship, however, the company wants you to add networking support to your `mdadm` implementation to increase the flexibility of their system. The JBOD systems purchased by the company can accept JBOD operations over a network using a proprietary networking protocol. Specifically, a JBOD system has an embedded server component that can be configured to have an IP address and listen for JBOD operations on a specific port. In this final step of your assignment, you are going to implement a client component of this protocol that will connect to the JBOD server and execute JBOD operations over the network. As the company scales, they plan to add multiple JBOD systems to their data center. Having networking support in `mdadm` will allow the company to avoid downtime in case a JBOD system malfunctions, by switching to another JBOD system on the fly.

Currently, your `mdadm` code has multiple calls to `jbod_operation`, which issue JBOD commands to a locally attached JBOD system. In your new implementation, you will replace all calls to `jbod_operation` with `jbod_client_operation`, which will send JBOD commands over a network to a JBOD server that can be anywhere on the Internet (but will most probably be in the data center of the company). You will also implement several support functions that will take care of connecting/disconnecting to/from the JBOD server.

Protocol

The protocol defined by the JBOD vendor has two messages. The JBOD *request message* is sent from your client program to the JBOD server and contains an opcode and a buffer when needed (e.g., when your client needs to write a block of data to the server side jbod system). The JBOD *response message* is sent from the JBOD server to your client program and contains an opcode and a buffer when needed (e.g., when your client needs to read a block of data from the server side jbod system). Both messages use the same format:

Bytes	Field	Description
0-1	length	The size of the packet in bytes
2-5	opcode	The opcode for the JBOD operation (format defined in Lab 2 README)
6-7	return code	Return code from the JBOD operation (i.e., returns 0 on success and -1 on failure)
8-263	block	Where needed, a block of size JBOD_BLOCK_SIZE

Table 1: JBOD procotol packet format

In a nutshell, there are four steps:

1. The client side (inside the function `jbod_client_operation`) wraps all the parameters of a jbod operation into a JBOD request message and sends it as a packet to the server side
2. The server receives the request message, extract the relevant fields (e.g., opcode, block if needed), issues the `jbod_operation` function to its local jbod system and receives the return code
3. The server wraps the fields such as opcode, return code and block (if needed) into a JBOD response message and sends it to the client

4. The client (inside the function `jbod_client_operation)` next receives the response message, extracts the relevant fields from it, and returns the return code and fill the parameter "block" if needed.

Note that the first three fields (i.e., length, opcode and return code) of JBOD protocol messages can be considered as packet header, with the size HEADER_LEN predefined in net.h. The block field can be considered as the optional payload. You can set the length field accordingly in the protocol messages to help the server infer whether a payload exists (the server side implementation follows the same logic).

Implementation

In addition to replacing all `jbod_operation` calls in `mdadm.c` with `jbod_client_operation`, you will implement several functions defined in `net.h`, within the provided `net.c` file.

Specifically, you must implement the `jbod_connect` function, which establishes a TCP connection to the JBOD server using the `JBOD_SERVER` and `JBOD_PORT` constants (both defined in `net.h`). You must also implement the `jbod_disconnect` function to properly close this connection.

Both of these functions are invoked automatically by the `tester` so you will not call them directly in your code.

The `net.c` file includes skeletons for other helper functions (e.g., `send_packet`, `recv_packet`) that will help in organizing your implementation. While you are free to write additional helper functions, you must ensure that all functions declared in `net.h` are implemented and behave as expected when called by `tester.c` and `mdadm.c`.

For more information about the function responsibilities, parameters, and expected return values, refer to the comments in `net.c` and the function declarations in `net.h`.

Testing

Once you finish implementing your code, follow the steps below to test your work using the provided `jbod_server` and `tester` programs.

Step 1: Prepare the Server Binary

Depending on your machine architecture, **you must first rename** the appropriate JBOD server and object files:

- **If you're on an ARM64 (e.g., M1/M2 Mac):**

```
mv jbod_server_arm64 jbod_server
mv jbod_arm64.o jbod.o
```

- **If you're on an x86 machine (Intel or most Linux systems):**

```
mv jbod_server_x86 jbod_server
mv jbod_x86.o jbod.o
```

After renaming, make the server binary executable:

```
chmod u+x jbod_server
```

Step 2: Run the JBOD Server

In one terminal, start the server:

```
$ ./jbod_server
```

You should see:

```
JBOD server listening on port 3333...
```

Note: Nothing else will appear until you run the client (`tester`) from another terminal.

Step 3: Run the Tester in a New Terminal

In a second terminal, run the tester with a trace file:

```
$ ./tester -w traces/random-input -s 1024 > x
```

Expected output:

```
Cost: 17669400
Hit rate:  24.5%
```

To verify correctness:

```
$ diff x traces/random-expected-output
```

If the `diff` command produces no output, your implementation is correct.

Step 4 (Optional but Recommended): Use Verbose Mode for Debugging

Verbose mode prints every command the server receives. This is **very helpful for debugging**, especially to check if your command encoding is correct.

To enable verbose mode:

```
$ ./jbod_server -v
```

Example output (trimmed for clarity):

```
JBOD server listening on port 3333...
new client connection from 127.0.0.1 port 38546
received cmd id = 0 (JBOD_MOUNT) [disk id = 0 block id = 0], result = 0
received cmd id = 2 (JBOD_SEEK_TO_DISK) [disk id = 0 block id = 0], result = 0
received cmd id = 5 (JBOD_WRITE_BLOCK) [disk id = 0 block id = 0], result = 0
block contents:
0x00 0x00 0x00 0x00 ...
```

If you see unexpected commands or zeros where other values should be, there might be an issue with your encoding logic.

If your implementation is correct, your output x will be the same as the expected output from each trace workload. You will use the diff command to measure the difference, as you did for lab 3 and lab 4. Since your lab 5 code does not change the caching policy, it should produce the same result as that from your lab 4. **We will only consider your net.h, net.c, cache.h, cache.c, mdadm.h and mdadm.c from your submission in our test.**

Deliverables: Push your code to GitHub. Submit the commit ID of your latest code to Canvas for grading. Ensure that there is no additional text, words, or comments around the commit ID. Check canvas assignment page for more details.

Grading rubric The grading would be done according to the following rubric:

- Passing trace files with cache size 1024: 25% for simple input, 35% for random and 35% for linear trace files. We will not measure caching efficiency (hit rate) and cost (even if cost is 0). We do require that cache be used correctly.
- Adding meaningful descriptive comments: 5%
- If you have **any make errors or warnings or program is stuck in a loop** then you will **receive a straight 0**. Make sure your code does not have any make error before submitting
- **If you do not submit your commit ID** on canvas you will receive a **straight 0**

Penalties: No late submission allowed. **We will not grade any submissions after 05/02/2025.**

Common Problems and Solutions

1. **Received "invalid packet length" error:**
   This typically means the length field in your packet is incorrect. Ensure that:
   - If no block is being transmitted, length should equal `HEADER_LEN`.
   - If a block is included, length should equal `HEADER_LEN + JBOD_BLOCK_SIZE`.

   Also, make sure to use the proper byte order conversion functions:
   - `htons()` and `ntohs()` for 16-bit values (e.g., length).
   - `htonl()` and `ntohl()` for 32-bit values (e.g., opcode).
2. **Confused about reading with `recv_packet`:**
   Split the reading into two parts:
   - First, read the header (`HEADER_LEN` bytes).
   - Then, based on the length field, read the rest (payload), if applicable.
3. **Error: `jbod_server bind failed: Address already in use`**
   This means the server is already running in the background. You need to stop it before restarting:

   ```
   $ ps aux | grep jbod_server
   $ kill <PID>
   ```

4. **Trace file takes forever to run:**
   If your traces (especially `random-input`) take longer than 1-2 minutes:
   - You may have redundant calls to `jbod_client_operation`.
   - Optimize your `send_packet`: send header and block in one `nwrite` call using a single buffer, instead of two separate calls.
5. **How to debug using `gdb`:**
   GDB is useful for tracing where your code is stuck. Example:

   ```
   $ gdb tester
   (gdb) break send_packet
   (gdb) run -w traces/simple-linear
   (gdb) next
   ```

You can step through with `next` or `n`, inspect variables, and explore the call stack. You may also try breaking at other functions like `recv_packet` or `jbod_client_operation`.