LAB #5 – mdadm Linear Device (Networking)

CMPSC311 - Introduction to Systems Programming

Fall 2022 - Prof. Suman Saha

Due date: December 9, 2022 (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 including (discord or GroupMe, or other group messaging apps) or discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone in or outside of the class. Failure to abide by this requirement will result in a 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 its 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 issues 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
1-4	opcode	The opcode for the JBOD operation (format defined in Lab 2 README)
5	info code	The info code is 1 byte. lowest bit will represent Return code from the JBOD operation(bit:0 == JBOD operation returns ‘0’; bit:1 == JBOD operations returns ‘-1’). Second lowest bit will represent whether Data block(payload) exists or not (0 == data block is not present in the packet; 1 == data block is present) Ex. Infocode= 00000001 -> JBOD operation returns -1 and the packet contains no data block
6-261	Data block(Payload)	Where needed, block size == JBOD_BLOCK_SIZE (256)

Table 1: JBOD protocol packet format

In a nutshell, there are four steps. (Sl) 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; (S2) the server receives the request message, extracts the relevant fields (e.g., opcode, info code, block if needed), issues the jbod_operation function to its local JBOD system and receives the return code; (S3) The server wraps the fields such as opcode, info code (JBOD operation return code and data-block presence flag) and block (if needed) into a JBOD response message and send it to the client; (S4) the client (inside the function jbod_client_operation) next receives the response message, extracts the relevant fields from it, and returns the return code (from info code) and fill the parameter "block" if needed. Note that the first two fields (i.e., opcode and info code) of JBOD protocol messages can be considered as packet headers, with the size HEADER_LEN predefined in net.h. The block field can be considered as the optional payload. You can use the 2nd lowest bit of the 5th byte in the protocol messages to help the client infer whether a payload exists or not (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 functions defined in `net.h` in the provided `net.c` file. Specifically, you will implement `jbod_connect` function, which will connect to JBOD_SERVER at port JBOD_PORT, both defined in `net.h`, and `jbod_disconnect` function, which will close the connection to the JBOD server. Both of these functions will be called by the tester, not by your code. The file `net.c` contains some functions with empty bodies that can help with structuring your code, but you may implement your helper functions as long as you implement those functions in `net.h` that will be directly called by tester.c and mdadm.c. That being said, following the structure would probably be the easiest way to debug/test/finish this project. Please refer to `net.c` for a detailed description of the purpose, parameters, and return value of each function.

Testing

Once you finish implementing your code, you can test it by running the provided `jbod_server` in one terminal, which implements the server component of the protocol, and running the `tester` with the workload file in another terminal. Below is a sample session from the server and the client:

Output from the `jbod_server` terminal:

```
$ ./jbod_server
JBOD server listening on port 3333...
new client connection from 127.0.0.1 port 32402
client closed connection
```

Output from the `tester` terminal:

```
$ ./tester -w traces/random-input -s 1024 >x
Cost: 17669400
Hit rate:  24.5%
$ diff x traces/random-expected-output
$
```

You can also run the `jbod_server` in verbose mode to print out every command that it receives from the

client. Below is sample output that was trimmed to fit the space.

```
$ ./jbod_server -v
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 = 0J, result = 0
received cmd id   2 (JBOD_SEEK_TO_DISK) [disk id = 0 block id = 0J, result = 0
received cmd id   5 (JBOD_WRITE_BLOCK) [disk id = 0 block id = 0J, result = 0
block contents:
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

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.

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

- Passing trace files with cache size 1024: 25% for simple input and 30% each for random and linear trace files. **We will not measure caching efficiency and cost this time.**
- Adding meaningful descriptive comments: 5%
- Successful "make" and execution without error and **warnings:** 5%
- Submission of commit id: 5%