Fall 2024 - LAB #5 - mdadm Linear Device (Networking)

CMPSC311 - Introduction to Systems Programming

Due date: December 11, 2024 (11:59 PM) EST

NO EXTENSIONS OR LATE SUBMISSIONS ACCEPTED

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
1-4	opcode	The opcode for the JBOD operation (format defined in Lab 2 README)
5	info code	Info code is 1 byte. lowest bit will represent Return code from the JBOD operation(2 states: 0,-1) + second lowest bit will represent whether Data block(payload) exists or not
6-261	Data blocks(Payload)	Where needed, a block of size JBOD_BLOCK_SIZE

Table 1: JBOD protocol 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, info code, 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 as part of info code and block (if needed) into a JBOD response message and send 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 two fields (i.e., opcode and info 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 use 2nd lowest bit of 5th byte in the protocol messages to help the client 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 functions defined in net.h in the provided net.c file. Specifically, you will implement:

1. **jbod_connect(const char *ip, uint16_t port):** Which will connect to JBOD_SERVER at port JBOD_PORT, both defined in net.h, and set *cli_sd* to track the client socket descriptor for the connection.
2. **jbod_disconnect(void):** Which will close the connection to the JBOD server. Both this and jbod_connect will be called by the tester, not by your own code.
3. **Jbod_client_operation(uint32_t op, uint8_t *block):** Your replacement for jbod_operation, this will make requests to jbod over the connection you establish in order to perform all jbod operations used in the previous assignments.

The file net.c contains some functions with empty bodies that can help with structuring your code:

1. **nread(int fd, int len, uint8_t *buf):** Will attempt to use passed file descriptor fd to read *len* number of bites from the connection, for when jbod is sending block information to your client.
2. **nwrite(int fd, int len, uint8_t *buf):** Will attempt to use passed file descriptor fd to write *len* number of bites to the connection, for when jbod is expecting to receive information from your client.
3. **recv_packet(int fd, uint32_t *op, uint8_t *ret, uint8_t *block):** Will attempt to use passed file descriptor fd to receive a formatted response packet from jbod.
4. **send_packet(int fd, uint32_t op, uint8_t *block):** Will attempt to use passed file descriptor fd to send a formatted request or response packet from jbod.

You may implement your own help 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 the detailed description on 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 .l 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.l 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 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:

- Passing trace files with cache size 1024: 30% for simple input
- 35% each for random and linear trace files.
- We will not measure caching efficiency and cost this time.

-