HW4

October 24, 2023

1 HW#4

Running the project:

Server: `python3 ./blockserver.py -nb 256 -bs 128 -port 8000`

client: `python3 ./fsmain.py -port 8000 -cid 0`

1.1 `block.py`

1.1.1 Implement Locking

1. `RSM (self, block_number)`
    1. Primitive to invoke `blockserver.py` and implement locks
    2. There is going to be a single lock for the enire block array

    ```
    ...
    block = block_server.Get(fsconfig.Total_Num_Blocks - 1)
    ...
    ```

2. `Acquire (block_number)`
    1. before performing the operation and acquiring the lock - call `CheckAndInvalidateCache ()`.
    2. Book's implementation is referred.
    3. `RSM` on block.
    4. if block is already locked - wait until it's unlocked and grab the lock.
3. `Release ()`
    1. set the block to unlocked - `RSM` block value to `0`

1.2 Implement *at-least-once RPC* semantic

- Client resends request until receiving a response from the service, ensuring the request is processed at least once and design recover from timeouts
- setup retry mechanism for all `block.py` functions.

Implementation:

1. Modify `Put, Get, and RSM` definitions to implement exception handling with `try-except` block.
2. Catch `TimeoutError`

3. log `SERVER_TIMED_OUT`.
4. Setup retry - add recursive call into the `Except TimeoutError` block until call succeeds.

1.3 Client Side *write-through* Cache

1.3.1 initialize cache into `DiskBlocks ()` Constructor

```
def __init__(self):
    ...
    self.cache: {}
    ...
    pass
```

1.3.2 Write `CheckAndInvalidateCache ()`

1. Pull the latest CID blocks from the server
2. Compare it with the `fsconfig.CID`
3. Incase both are different - invalidate cache `cache = {}`
4. Print `CACHE_INVALIDATED`

1.3.3 Refactor `Get (), Put (), RSM ()`

1. Refactor current function definition and implement server calls
2. Incorporate at-least-once semantics.

1.3.4 Additionally -

Refactor `PUT ()`

1. Call server to update datablock
2. Update the client cache with data - `cache[block_number] = block_data`
3. Log `CACHE_WRITE_THROUGH`
4. Update server `CID` with `fsconfig.CID` to indicate the last update

`GET ()`

1. If the block_number is available in the cache, return `cache[block_number]`
2. Log `CACHE_HIT {BLOCK_NUMBER}`
3. If block is not in the cache OR if request is for RSM or CID block fetch it from the server.
4. Log `CACHE_MISS {BLOCK_NUMBER}`
5. Read through the unavailable data `cache[unavailable_block_number]=fetched_block_data`

1.4 `shell.py`

To Implement the *before-or-after* atomicity:

- Ensures that the result of every read or write operation appears as if it occurred entirely before or entirely after any other read or write operation.
- Any operation that is leading a write on the server will be eligible to update the content available on the server. This also applied to operations where we are reaching out to the server to fetch data blocks.

- Wrap the operations with locking with `Acquire()` and `Release()`

1.5 Testing

- `Acquire(), Release()` were tested by manually calling them and reading the `RSM` block right after that making sure that the value is `1` in case of locked `0` in case of unlocked data block.
- `RSM()` primitive was tested individually by putting a log over `blockserver.py` and making sure that it's being called.
- Once the cache invalidation and the cache policy was implmented, the strategy was to be able to run the program on smaller tests. i.e. first try running the program and looking into the specifics for one client and then with multiple clients and then gradually adding complex commands.
- This exposed a minor redundancy while setting up the RSM and CID blocks.
- This also exposed the issue with cache invalidation method call. In the initial implementation the invalidation was called along side the `Put` but after spending significant time I realised that if we acquire the lock and no other client is able to write on the server, we could just invalidate the cache at the time of acquiring the lock. Which fixed multiple failing test cases (looking at the diff file.)
- Next step was to be able to be able to understand the server delays, which after some time I realised is the delay after the number of steps and not the time at which delay should happen. Which helped in fixing issues related to expected print statements.

While the design for the method was not changed, it was a iterative process in terms of deciphering the expectations from the test cases and aligning the implementation, mostly the print statements.

1.6 Assignment Questions

1. **If you implement Acquire() and Release() correctly, multi-step operations run exclusively in one client at a time to enforce before-or-after atomicity. Suppose you didn't implement Acquire()/Release(). What is one example of a race condition that can happen without the lock? Simulate a race condition in the code (comment out the lock Acquire()/Release() in the cat and append functions, and place sleep statement(s) strategically) to verify, and describe how you did it.**

   A race condition happens when two or more threads or processes access shared data simultaneously, leading to unpredictable and undesirable behavior.

   Stale reads are a possibility, meaning that `cat` could potentially read stale file data, while the append operation is not done executing.

   To simulate this -

   - removed the Acquire/Release from both cat and append
   - Add timeout at the append function to delay the write on server
   - Call `cat` from another client to read the data while append is writing on the server
   - `cat` returns old file data.

2. **Describe, in your own words, what are the changes that were made to the Get() and Put() methods in the client, compared to the HW#3 version of the code?**

   - Homework 4 divided the FS into two layers - 1. server and clients. The data now resides on the server which exposes RPCs to provide an interface for the clients to connect.

- This required a change in `Get and Put` method to call the RPCs to fetch or update the data.
- Caching, allowed us to reduce the number of RPCs happening to re-fetch a block of data.
- `Put` method, after it fetches the latest data block from the server, update the cache.
- `Get` method, before calling the RPC, checks the cache for the data block. If the block is not present it fetches the data from server and again store it in the cache.

3. **At-least-once semantics may at some point give up and return (e.g. perhaps the server is down forever). How would you implement this in the code (you don't need to actually implement; just describe in words)**

   - To be able to give-up or not run retry forever and get out of the retry "loop", we need a break-out condition.
   - Condition could be setup by setting up a *retry limit*.
   - This will require us keep track of retry count and use it to return eventually.
   - To implement this -
     - We would need to set up variables called `retry_limit` and `retry_count`
     - Each recursive call will increase `retry_count` by `1`
     - Update the `Except` block to include a `if` condition - `if retry_count < retry_limit` before making the call.
     - This set up will stop retrying once the `retry_limit` is reached.

4. **Discuss in what respects this implementation is similar to NFS, and in what respects it is fundamentally different from NFS**

   Similarities:

   - Like NFS, this implementation is a networked file system, where multiple clients could connect to a file system server.
   - This implmentation too supports and uses RPC protocols to invoke file operations on the server.

   Differences:

   - The main difference is the statelessness of NFS, which means that NFS does not store any information about the connected clients including - client-id. But this implementation stores it. (later version do track this but I think that's not the reference for this implementation).
   - Server Client ID here, enables us to implementation caching policies, which reduces the number of server hit rates, also reduces conflicts when server has multiple active clients.
   - This implementation locks the entire block while the NFS could support locking a file.

5. **Evaluate the performance of the cache you implemented. Create three test benchmarks, describe the reasoning behind your tests, and evaluate: 1) the cache hit rate, and 2) the expected improvement in average latency due to the cache, assuming that the hit time is 1ms and the miss time is 100ms**

1.6.1 Cache Performance Evaluation

- **Test Description**: Test:ID:CLIENT:CID:OPERATION

- **Cache Hit Ratio**: hits / (hits + misses)
- **Latency equivance (Given)**:

  ```
  Hit  == 1ms
  Miss == 100ms
  ```

  With all of the scenarios presented below, the objective is to analyze:

  1. Whether or not caching improves the operation performance by skipping the Get server hits for data by measuring the total latency with hits and without hits and measuring the overall improvement in the system calculated in %.
  2. How does this get affected when multiple clients are involved in the operations over the server.

  Testing Conditions and Assumptions:

  1. Tests metrics do not take operation while establishing the connection with the server into account for any of the client. these operations are considered to be affecting all of the clients the same.
  2. For test 2, we only consider the performance of the `slice` operation in this test and not the set of operations - creating the file and appending it with data. Since, these operations are common for both tests - 2A, 2B.
  3. For the ease of measurements of HIT/MISS - before each test the server is restarted and the required clients are re-connected.

Test 1: Basic Write operations   Tests:

```
Test:1A:CID:0:create f1 > CID:0:append f1 withdata
Test:1B:CID:0:create f1 > CID:1:append f1 withdata
```

Test 2: Operations that involve both Reads and Writes   Tests:

```
Test:2A:CID:0:create f1 > CID:0:append f1 withmoredata > CID:0:slice f1 0 2
Test:2B:CID:0:create f1 > CID:0:append f1 withmoredata > CID:1:slice f1 0 2
```

Test 3: Lookup and Read

```
Test:3A:CID:0:create f1 > CID:0:append f1 withdata > CID:0:cat f1
Test:3B:CID:0:create f1 > CID:0:append f1 withdata > CID:1:cat f1
```

Results

Test Name	Cache Hit Ratio	Latency	Improvement In Latency
1A	9/11	209 ms	81%
1B	9/14	509 ms	63%
2A	11/12	111 ms	90.75%
2B	8/12	408 ms	66%
3A	8/9	108 ms	88%
3B	4/8	404 ms	49.5%

Conclusion As we could see the caching has performed well and have reduced the latency overall. However, it's interesting to note that the latency of tests which involved two clients (Bs) have higher latency. Which make sense because of the additional reads required to fetch new information from the server.