11/10/2023

LAB4 – CS 380P – JOHN PETERSON - JLP5729

1. PART 1 -- 2PC PROTOCOL

My implementation uses a sequential, state-based model for the coordinator and a simpler state-based solution for both client and participant. To establish initial communication, I pass an `IpcOneShotServer` to the child processes (participant and client) so that they can transmit one tx and rx channel each to the coordinator. It took me longer than I should like to admit that even though "OneShot" is in the name, I needed to use one server per process, and the object was consumed after transmitting one item on it.

Additionally, I implemented non-blocking receive in the coordinator to check each client for messages and add them to a queue for processing. This prevents the coordinator from blocking handling transactions, increases overall throughput, and guarantees proper order of messages.

My biggest challenge in this lab was understanding Rust and the programming model necessary to satisfy the compiler and type safety requirements. This helped me iterate and test progressively, leading to fewer unintended bugs.

I initially planned not to use the states in coordinator, but when trying to build a loop and complex logic, I quickly realized that a state machine would be much more helpful and reset my code to follow this pattern.

Performance was much more challenging to guarantee in the distributed/inter-process model. I spent several hours figuring out where processes were or weren't getting communication necessary or ending up in an unintended state. However, once I determined and fixed the issues, I ran several tests with flag `-s 1.0` to test performance with varying numbers of messages, participants, and clients. I verified that at all configurations, over multiple runs, all messages were successfully committed.

The following configurations were tested ten times each to verify the successful commit of all transactions with `-s 1.0` set:

- `-c 4  -p 10 -r 10 -s 1`
- `-c 4  -p 15 -r 10 -s 1`
- `-c 4  -p 10 -r 15 -s 1`
- `-c 4  -p 50 -r 25 -s 1`
- `-c 10 -p 10 -r 10 -s 1`
- `-c 10 -p 5  -r 15  -s 1`
- `-c 10 -p 15  -r 25  -s 1`
- `-c 15 -p 25  -r 100 -s 1`
- `-c 15 -p 100 -r 25  -s 1`
- `-c 25 -p 25  -r 25  -s 1`

All configurations committed all transactions; this allowed me to move forward with implementing message send failure, addressed below in Part 2.

2. PART 2: MESSAGE SEND FAILURE

As I had already implemented operation failure, and this didn't require significant infrastructure changes, this part consisted of determining a strategy and implementing a timeout requirement.

I varied the timeout specification trying the following set and variable millisecond values per transaction:

- `1 ms`
- `2 ms`
- `10 ms`
- `100 ms`
- `num_participants`
- `num_participants * 2`
- `num_participants * 10`
- `num_participants * 5`

Once again, I ran these timeout values with flag `-s 1.0` to ensure that all messages would be received with the given timeout. I varied the client, participant, and request numbers to match the list in Part 1. However, I experienced unintended message failure when using lower, set values of timeout, and while 100ms did not result in timeout, it also caused much slower execution. This caused me to realize that a variable timeout value could be more helpful. I then tried the values in the right column and found most success at `num_participants * num_clients * 5`. This balanced message load and correct reception while not adding too much delay for small values.

Most of my testing for timing and performance was done on my MacBook Pro with the Apple M1 Pro ARM64 processor; however, I verified performance on Codio before submission.

If this program were to be implemented on a more parallel/powerful machine than the Codio or my MacBook, it is entirely possible that a lower timeout value could be set, especially if inter-process communication was more efficient on a different architecture.

Additionally, I realized after my initial submission that the program should be able to handle a SIGINT prior to completion. I made modifications to all while loops, and added match OK/Err conditions for many send/receive operations that might not be performed once a SIGINT is received early; this prevented panic from calling unwrap() on an Error.

3. PART 3: INSIGHTS GAINED

The insights gained fall into two categories: type/concurrency safety provided by Rust language and compiler and considerations for distributed systems/transaction guarantees.

Some of these insights were mentioned in the above sections for the first category. However, my insights also include the limitations of the Rust compiler and static analysis. Once my algorithm was complete, I noticed that the Rust compiler was giving some warning for unused code. One by one, I cleaned up the unused code and fields that it was warning about. However, once all the warnings were gone (either due to manual changes or automatic changes suggested by the Rust compiler), my code was no longer working for values of flag `-S` less than 1.0. Rather than investigate and debug thoroughly, I reverted to a previous code version. It is possible that I made changes that caused the program to break, but if the rust compiler "thought" that the code wasn't used, I did not think deleting it would cause that problem. I would like to investigate further when using Rust in future personal or professional projects.

As for the second category of insights, sharing the state or ensuring each participant/process is in the intended state is a unique challenge. If messages aren't guaranteed to be delivered in the correct order or delivered at all, more information/coordination is required to ensure successful operation and data is handled as intended. As a thought experiment, I thought about what it would take to have a distributed coordinator. This would require even more guarantees and explicit synchronization between coordinators and processes to ensure proper delivery.

Finding a good balance for a timeout value was just one small aspect that opened my eyes to an entire category of challenges in distributed systems, and many more aspects can be important to real-world distributed systems.

I spent approximately 30 hours on this lab, broken down as follows:

- 8-10 hours trying to figure out where to start, understanding the codebase, and the IpcOneShotServer and ipc_channel packages.
- 5-8 hours setting up communication between processes and determining the info necessary to correctly transition state
- 4-6 hours writing state machine code
- 5-7 hours troubleshooting state machine and process communication
- 2-3 hours testing values for timeout and writing report.