CIS*3050 A2 Report

David Pearson

1050197

2022-11-14

Note: After completing this report I recognize its length may be longer than expected, I apologize for that. The report covers the entire program in detail and how it was implemented. All sections have been clearly labeled to make it easier to skim to the parts that are necessary for marking. To make things easier a table of contents have been provided so key information is easily accessible.

Table of Contents

Run Instructions

All scripts must be given the appropriate permissions before they can be run.

In one terminal window run ***'./server.sh'*** then in another terminal window we can submit jobs to be distributed to the workers using ***'./submitJob.sh [CMD]'*** where CMD is the task to be sent to the server. Other options for CMD include **'-s'** to output server status and **'-x'** to gracefully terminate the server and all workers.

server.sh Overview

How does the server track worker status? On line 11 we declare an associative array with the worker ID being the key and the stored value can be 'ready' or 'wait' depending on its status.

How are workers generated? Workers are spawned on start-up with the *spawn_workers()* function defined on line 13. For each of the cores detected in the system, we execute '*./worker.sh $i &*' which passes the workers ID as a parameter and runs the processes in the background. In this function we also initialize the workers status entry in the array to 'ready'

How does the main loop work? The main loop starts on line 91 and will continue indefinitely until the server receives a message to exit. On line 97 we read from the server FIFO. From the line we read we extract the communication identifier called the 'token' here. Since the server can be receiving messages from both the worker and submitJob.sh, we must find a way to differentiate how communication is handled. This will be discussed further in the Communications sections of this report. If the server receives a message from a worker it will be a worker status update. In this section of the loop, we update the status array with whatever is received and alter the status variables to reflect the changes. If the 'token' we received started with 'MSG' then the server executes the special cases for shutdown and status. The actual command distribution takes place starting at line 136. When the message read from the FIFO has no communication token we interpret it as a command. We add the command to the queue, and then if the worker currently pointed to is ready, we can deliver the command at the top of the queue.

How does the server handle exit? Cleaning up the server on exit is handled by two functions that work together. First is *'killReady()'* defined on line 41. This function will send the quit message to all workers with a 'ready' status starting at the worker pointed to by *cur_worker*. In the case where all workers are ready (not executing) we will continue to kill all the workers until the number of exited equals the number of cores. Once this state is reached we remove the servers fifo and exit. In cases where not all were ready to exit when this function is called, we continue in the main cleanup function '*newClean()*' defined on line 66. This function initially calls the killReady function to exit workers that can be. If not all were ready, we enter a loop waiting until the remaining workers have the 'ready' status. The loop, which starts on line 72, checks the conditional that 'wait' is still found in the status array. If some workers are waiting we can start reading from the FIFO. At this point, since the user initiated the shutdown sequence we can ignore any other submissions to the server that are not messages from the workers. This loop will continue reading from the FIFO and updating worker statuses until all workers are in the ready state. At this point we call '*killReady()*' a final time to finish the cleanup.

worker.sh Overview

The worker script is really just a stripped-down version of the server. It continuously reads from its FIFO in a loop until shutdown is requested. The worker though must handle the log file linked to its ID. This ID is passed to the worker on start-up in the server. On line 15 the log file is cleared from the previous execution. In the main loop which starts on line 17, we read from the FIFO until the exit message is received. All communication that the worker receives will have a token identifier, we must parse this out. If the worker receives 'MSG quit' the server notifies the worker to shutdown. In this segment, the worker removes its FIFO and exits. If the worker receives 'CMD {task}' it notifies the server that it is busy and runs the task, appending the output to the worker log. Once the task is complete the worker will notify the server it is ready once again.

Communication Protocol

The easiest way to discuss how the communication protocol is implemented is to step through the scripts as a message is passed along. This process starts in *submitJob.sh* where we identify if a special task has been issued. These tasks could be *-x* or *-s* for shutdown and status respectively. If either of these tasks are received by the submission script, we issue a special message to the server with 'MSG' appended to the front to differentiate between a 'shutdown' executable and the server shutdown message. Generally, this is how all communications are handled, special messages have a token appended to the start so whatever is reading it knows how to handle it. If neither of these special cases are issued, it is assumed to be an executable of some kind. This allows the user to submit a job for a worker to run '*shutdown'* or '*status'*.

Within the server, many different forms of communication may be received since this acts as a hub for all interactions. We differentiate between worker messages and the special commands noted above with the token. For workers, the messages they send to the server will always be related to the status. To note this, workers will send their update with 'WS' appended to the front. This is meant to indicate 'worker status' where 'WS 3 ready' is an example of worker 3 alerting the server it has finished executing. These messages are sent in the format 'WS [worker_id] [status]'. If the server reads a line from the FIFO that starts with 'MSG' this indicates one of the special commands discussed above. These messages will always be in the form 'MSG [command]' where command is either shutdown or status. If there is no identifier token appended, we interpret this as an executable to be sent to a worker.

If the server identifies the message received from submitJob.sh is an executable (no token) we can send this command to the workers. This command is sent to the worker with the token identifier 'CMD' at the start. In the worker, there are two forms of communication we expect to receive, commands and shutdown notification. Shutdown is handled if the message read is in the form 'MSG quit'. For commands, these will be received in the format 'CMD [command]'. For these commands the worker will simply identify the token, and if it's a command, remove the token component and run it, appending the output to the worker log.

Test Cases – runtests.sh Overview

How to run the test script: Simply ensure that all scripts have the proper permissions, then run ***./runtests.sh*** Everything is handled cleanly within the script itself so only this single command must be executed.

How the test script works: To make testing easier, for each test case, a fresh instance of the server will be created. This is to ensure that testing is kept separate and also reduces the burden on the user. For each test, we call the *runTest()* function defined on line 134 this function takes in the test case as a parameter, starts up a server instance in the background, and runs the test case. For each such test case, it also runs *testExit()* which is a function that validates that the shutdown sequence of the server was successful. To make analysing server status easier, the output of the server will be placed in a file *server.log* which is created in the same directory the script was executed in. Note: These test cases are designed in such away they should work with a varying number of cores but relies on the fact that there are at least 2 workers active. All development and testing was done on a machine with 16 active cores. For each test we expect a similar format of the output to make visual validation easier. Though test cases will automatically check for pass/fail, some additional information may be useful to the tester. For each test the script will output the name of the test case being executed, test status (pass/fail) and output on why the script validated it as such. It will also output the number of jobs the server has processed at the time of the status command being processed server side. While this stat would be convenient for test validation, some test cases where we bombard, the server slows to the point the status return can be delayed at the time of processing the test results. As such, testing is done based on the number of lines found in the log files for the workers which from lecture was indicated to be a simple and effective comparison.

Limitations: It is worth mentioning that in order to facilitate this form of automatic test validation, a number of sleep commands are issued in the script. This is done to slow processing of the tests down to the point that the server can catchup – command sent to worker, worker executes, worker ready status is read. As such there may be some instances where the output of the test indicates a failure when the log files contains the expected outputs. While I do not believe this to be an issue, since there were so many headaches setting up the tests to wait just long enough that the worker finishes the command but not so long that it takes forever to run, I figured it prudent to mention this fact. By nature of static value sleeps there is a chance the test script isn't waiting long enough to validate correctly. Screenshots will be included below of the outputs of the tests that I received and on the off chance this is an issue, tests can be run individually by commenting out the calls to *'runTest'* and manually validating through the worker logs. I stress this is a limitation of the test script potentially not waiting long enough and not necessarily an issue in the server.

The Test Cases

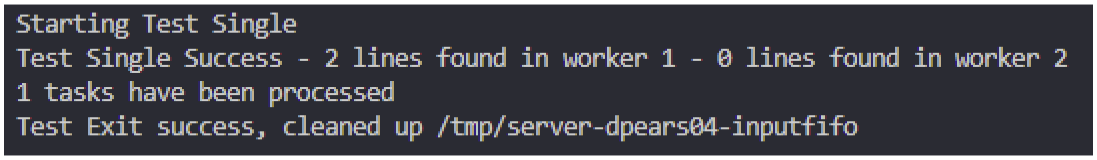
testSingle() – line 8

Object of test: This test is designed to validate that a single task submitted to the server is sent to the first worker, as would be expected in the round robin format. We check that the first worker receives and successfully processes that job and that the next worker in line did not receive any job. This test case will also validate that the server exited successfully be checking that the server FIFO was closed – the final step of the shutdown process.

Expected Output: The expected output of this test is that the log file for worker 1 has the output of the job *'./timedCountdown.sh 2'* and that worker 2 has an empty log file indicating no task has been executed on this worker. Another way to validate this, though it is not handled automatically is looking at the number of tasks processed which is visible in the testing output.

How to determine if test failed or succeeded: The test script will check that the log file for worker 1 has 2 lines and the log file for worker 2 has no lines. The timedCountdown script we used to test is ran for 2 seconds, meaning there should be 2 lines in worker 1. If the difference between the line count in worker 1 log and the worker 2 log is not 2 we know at some stage either worker 1 executed something not as expected or worker 2 received a job it should not have.

Did the code fail/succeed? Success – we can see from the screen shot that 2 lines were found in worker 1 and 0 found in worker 2. Further, the server status output indicates that 1 task was processed. The exit status also shows that the server cleaned up as expected



```
Starting Test Single
Test Single Success - 2 lines found in worker 1 - 0 lines found in worker 2
1 tasks have been processed
Test Exit success, cleaned up /tmp/server-dpears04-inputfifo
```

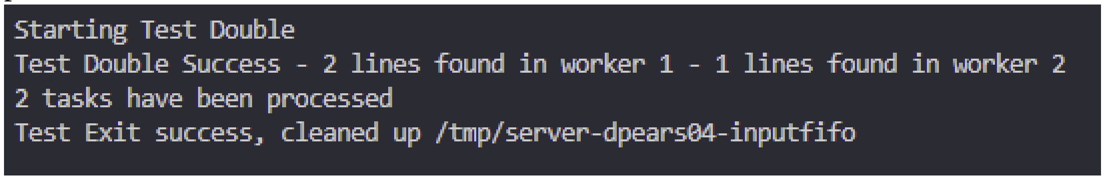
testDouble() – line 25

Object of test: For this test the goal was to verify on a small scale that round robin is working as intended. We submit two jobs, which if all goes according to plan, on a system with at least two cores, it should give a task to worker 1 and a task to worker 2. Each of these tasks is different so we can tell if by some chance an error gave the same task to both. This is a test that job distribution is working as intended without relying on the queue or round robin being circular.

Expected Output: The expected output is that the worker 1 log contains 2 lines, and the worker 2 log contains 1. This is what is expected given that each worker should be provided a different task.

How to determine if test failed or succeeded: The test case will compare the line count in each log file to see if the difference between them is 1. This is a simple way to validate that the workers ran the tasks successfully and that the correct task was given to each. Another way to determine this is manually by looking at the tasks that have been processed. You could even manually check each log file to ensure that the correct outputs are found.

Did the code fail/succeed? This test case succeeded. As seen by the screenshot below, this is the output the testcase provided. 2 lines were found in worker 1 and 1 line was found in worker 2 log files. AS this was expected the automatic validation indicated the test passed. Further manual validation shows that 2 jobs were processed by the server, which is inline with expectations. The exit status of the server was also successful, reflecting no issues in graceful shutdown after the processes ran.



```
Starting Test Double
Test Double Success - 2 lines found in worker 1 - 1 lines found in worker 2
2 tasks have been processed
Test Exit success, cleaned up /tmp/server-dpears04-inputfifo
```

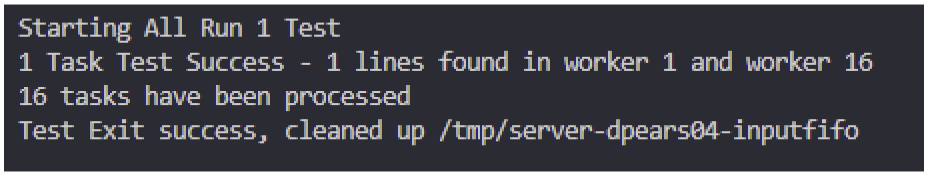
testAllRun1() – line 45

Object of test: This tests that the server can handle exactly as many tasks as there are workers available. This is a more rigours version of the previous test in that it validates round robin task distribution up to the number of cores. Given that the worker with an ID equal to the number of cores is considered an edge, it is important to validate up to that edge. This test demonstrates that all workers a proceeding as expected, not just 2 like the previous case covered.

Expected Output: This test case should result in every worker available being given a task and sending the output of that task to the log file. In this case since we are testing with '*timedCountdown.sh 1*' each file should have one line that says, "1 second remaining".

How to determine if test failed or succeeded: The automatic validation in the script will check that the number of lines in the worker 1 log is equal to the number of lines in the worker ${CORES} log. The idea is that if issues arise in distribution, these will not be equivalent. Other ways of validation include manually checking each of the worker logs. Also, we can see from the test output the number of tasks that have been processed. If this number equals the number of cores in the system, this test has passed.

Did the code fail/succeed? The code passed this test – the number of lines were equivalent; we processed the expected number of tasks (16 core system). Finally, the exit test showed that everything cleaned up as expected.



```
Starting All Run 1 Test
1 Task Test Success - 1 lines found in worker 1 and worker 16
16 tasks have been processed
Test Exit success, cleaned up /tmp/server-dpears04-inputfifo
```

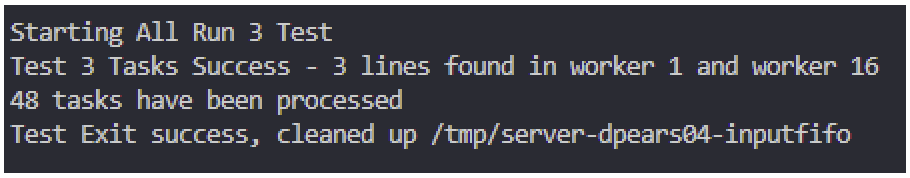
testAllRun3() – line 68

Object of test: The object of this test is to take the previous test and start to consider beyond the edge of the number of workers operational. This test is designed to submit 3 jobs for every worker running. In this case, it should submit 48 jobs. We are hoping to validate that round robin is working as intended when requests are being received but the workers are busy. Is the queue functioning as intended, are the correct tasks distributed? This test will also stress the server somewhat, 48 tests are being submitted all at once. We hope to see that the server can keep up and read the jobs into the queue and that as workers become available the appropriate task from the queue is handed off.

Expected Output: the expected output of this test case is that every worker contains three lines consisting of "1 second remaining". We also expect the server to process 48 tasks by the end of the full cycle.

How to determine if test failed or succeeded: This test case is automatically validated by the script by comparing the length of worker 1 and the length of the worker ${CORES} log files. If these file lengths are equivalent we know that no issues came up in the distribution algorithm. Further, we can do manual validations by inspecting all the individual log files and ensuring each has the 3 lines we expect. Lastly, we can consider the number of tasks that have been processed, if this number equals 3*${CORES} the test was a success

Did the code fail/succeed? The code passed this test. As we can see from the screenshot below, the same number of lines were found in both worker log files, indicating that tasks were given correctly. There are also 48 tasks completed as reported by the server's status message. This reflects the expected value and is further validation on the success of this case. Lastly, the exit testing was successful, everything closed cleanly as anticipated.



```
Starting All Run 3 Test
Test 3 Tasks Success - 3 lines found in worker 1 and worker 16
48 tasks have been processed
Test Exit success, cleaned up /tmp/server-dpears04-inputfifo
```

testBombard() – line 90

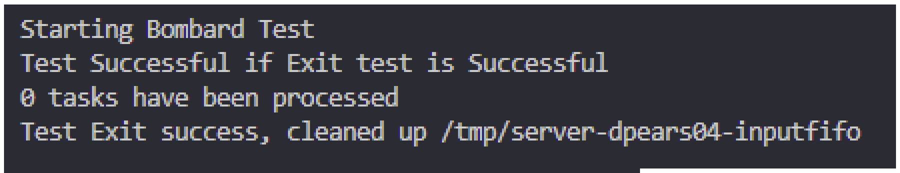
Object of test: The purpose of this test is to try and overload the server with requests. This test will submit 10 jobs for every worker operational. Since this test does not wait for the workers to complete their tasks, we also use this test as an opportunity to validate shutdown while workers are processing. When the server receives the shutdown request, it ignores the remainder of the command queue and waits for all the workers to enter the ready state before initiating shutdown. This test will validate that workers do not continue executing the remainder of the queue and that they successfully complete their task. As a by product, this test case also checks that the server can still receive and process a status request while all workers are busy executing.

Expected Output: The expected output of this test case is that the appropriate number of jobs enter the command queue. Along with this, we expect that the workers should complete one task, begin the second and receive a shutdown after completing the second task. In total, each worker should process 2, 5 second timedCountdown tasks. Although, this test is reliant on the sleeps found in the test script for how these requests are timed. As such contents of the logs were only manually validated. As a result of the sleeps in runtests, the status request sent to the server takes place before the workers finish their first task, resulting in an output of 0 for tasks processed. This is expected and reflects the limitations of the testing suite.

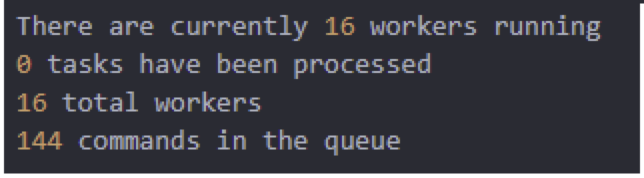
How to determine if test failed or succeeded: The major thing we are looking to test for this case is that graceful shutdown proceeds as expected despite the attempt to overload the workers. We expect that from the time the server receives the shutdown request that it finishes processing its tasks and no more.

Did the code fail/succeed? The code passed this test. I have included to screenshots below, one includes the testing output, and the other is the server status output found in the server.log file. As we can see from the status output, 16 workers are currently processing. This is expected and validates that all workers received a task and are executing. At the time the status request was submitted, the workers are still processing the first task and have not completed any. This is expected and is a result of a necessary sleep on line 146. The log file does reflect that there are

144 tasks in the queue waiting to be processed, coupled with the 16 being processed, we have 160 tasks total, 10 for each of the 16 workers. Meaning that all tasks were added to the queue successfully. Along with this, we can see that the server successfully terminates before all 10 of those tasks for each worker are processed. After manually reviewing the log file for worker 1, the server is waiting for the worker to be 'ready' before terminating. This test put the most stress on the system and encompassed a significant number of potential failure points, the code handled it like a champ.



```
Starting Bombard Test
Test Successful if Exit test is Successful
0 tasks have been processed
Test Exit success, cleaned up /tmp/server-dpears04-inputfifo
```



```
There are currently 16 workers running
0 tasks have been processed
16 total workers
144 commands in the queue
```

testSIGINT() – line 101

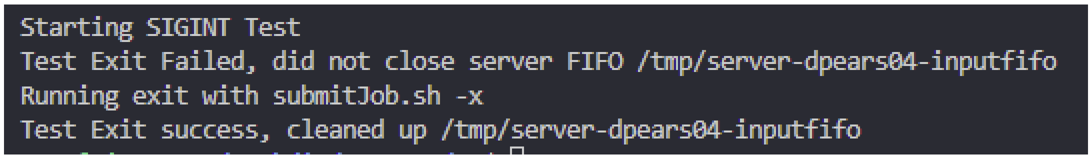
Object of test: The object of this test was meant to simulate a CTRL-C exit sequence of the server and validate that the signal catching is working as expected.

Expected Output: The expected output is that the exit test succeeded, and FIFO's have been removed. In other words, a graceful shutdown proceeded as planned.

How to determine if test failed or succeeded: The server would automatically determine this by checking if the FIFO files have been closed. Manually, we can run 'ps' to validate that there are no processes left running

Did the code fail/succeed? In this case the code failed, although I do not believe it is an error within my server code. Based on further research it appears that starting a new process in the background from a shell script can set the default SIGINT handler of that new background process to SIGIGN to prevent SIGINT propagation to those background processes. I did feel it valuable to leave this test case in however as a means of demonstrating test coverage.

Notably, we can manually execute this task and it **succeeds.** I found that if the same commands executed by this script are executed manually (copy and paste) in the bash terminal, the test passes. Further, if the server is started in the foreground and CTRL-C is issued, shutdown proceeds as expected. It was only when these commands were executed within the script does it fail, the server process actually becomes entirely unresponsive to SIGINT regardless of how it is delivered.



```
Starting SIGINT Test
Test Exit Failed, did not close server FIFO /tmp/server-dpears04-inputfifo
Running exit with submitJob.sh -x
Test Exit success, cleaned up /tmp/server-dpears04-inputfifo
```

testExit() – line 121

Object of test: The object of this test was to validate that the server shutdown process cleans up files and exits worker processes as expected. This was a key component of all the tests mentioned above as it provides even stronger code coverage. Alone the test cases above have great coverage, but by executing this test case for each we can test that despite the stress we force the server through, it still exits cleanly and gracefully.

Expected Output: The expected output for this test case is the output that the server fifo has been removed. This is the final element of the cleanup and is a great way to see if other elements failed to exit.

How to determine if test failed or succeeded: The test case will automatically check if the server fifo has been removed, but manually we can navigate to the /tmp/ dir and check that the fifo's have been closed.

Did the code fail/succeed? In all cases above the test case passes. Key to this test case however is that enough time is allowed to pass for the current processes to complete. As the limitations section mentions above, by using static sleeps which have been set to give just enough time that it works on my machine, we run the risk of validating too early. In the case this test is run on a slower machine and processing takes more time, the value of the sleeps may not be enough, and this test might check for the server FIFO before the server has completed its cleanup. In this case it would return a failure despite the fact that manual validation shows the cleanup was a success.

Final Notes on Testing

I believe that the test cases listed above are thorough and provide sufficient code coverage. Although each of these test cases are executed using the same timedCountdown process, it is worth considering that this script executes its own set of commands. By varying the commands used to test we would only be testing that the submission is delivering the correct command, which has already been tested. By using timedCountdown we test not only that scripts containing multiple commands execute on the worker and are handled by the server as expected, but that commands with arguments are passed correctly since timedCountdown must pass a time value. Each test case executed is really two tests in one because we test job submission and execution in its various forms but also the exit sequence for each. There are really ~12 tests in total as it is important to test that the server can shutdown correctly despite the stress the job test was designed to inflict. Finally, this test script and as a result, the server as a whole, was also executed successfully on another machine with only 4 worker processes.