CS425, Distributed Systems: Fall 2023
Machine Programming 1 – Distributed Log Querier

Group No: 58 **Names (netId):** Risheek Rakshit S K(rrs7), Siddharth Lal (sl203)
Repository link: *https://gitlab.engr.illinois.edu/sl203/mp1-cs425-group58.git*

The distributed log querier is built using Golang and comprises of both client and server implementations run in the same program. Our design handles the execution of grep commands on the server side, and subsequently, it sends the results back to the client. On executing the main.go program on any virtual machine (VM), the server behavior is initiated in the main thread of the program. Concurrently, the client function is invoked in a separate Goroutine. The server remains in a listening state, ready to handle incoming client requests, while the client routine waits for user grep command. The client resolves the IP addresses of the machines and initiates a connection request to all the servers (including its own VM counterpart) in parallel when user prompts.

Testing:

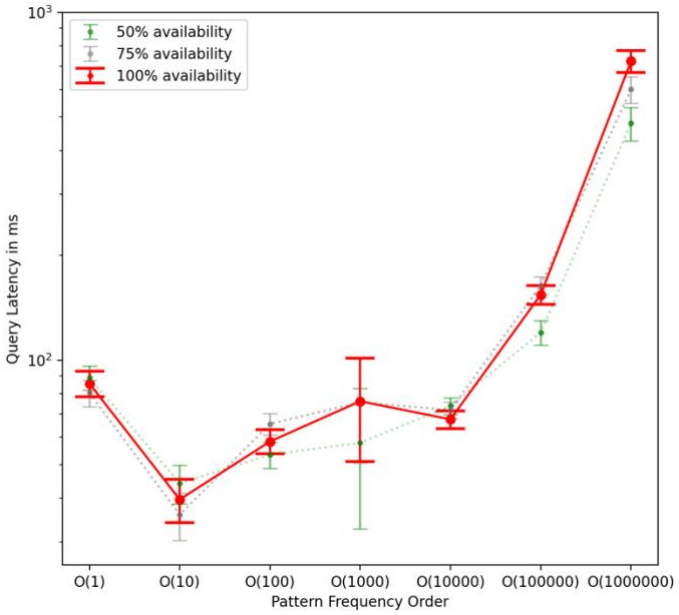
For simple testing, we have used the "testing" package of Go to perform standard unit tests on bash usage of "exec" package and a simple grep on a small test log via the exec package.

For distributed testing enabling sanity checks, we have designed tests for the following 7 grep command conditions, on the logs which were given for demo. We assume that all 10 machines are up initially:

- grep on a rare pattern
- grep on a frequent pattern
- grep on a somewhat frequent pattern
- grep on a pattern found on only 1 machine
- grep using -c option: counts
- grep using -E option: on a regex
- grep testing fault tolerance, when machine 10 is down/killed

Experiments:

To analyze the performance of our program, we performed experiments using 4 VMs (100% availability) containing logs of ~60MB each. We identified the patterns occurring across the log files from the least frequent to most frequent in exponential orders of 10. Using two VMs as clients we queried the identified patterns with 5 runs on each VM (10 queries in total per pattern). We ran the same experiment by crashing the program on one VM (75% availability) and two VMs (50% availability) respectively. The graph shows the average query latency along with SD error-bars on Y-axis on a log scale(excluding printing time on client) vs pattern frequency on X axis for the before mentioned three cases. O(1) means pattern occurs 0-9 times, O(100) means pattern occurs 100-999 times, and so on. (Note: We use *grep <pattern>* without any options)



Observations:

Average query time in ms: O(1): 85.6 O(10): 39.7 O(100): 58.3 O(1000): 76.1 O(10000):67.5 O(100000):154.1 O(1000000): 723.4. With all the results returned under 1 second for the above test scenarios.

For 100% availability (program runs on all 4 VMs; represented by solid red in the graph): Generally, as the frequency of the pattern increases, the query latency increases. This can be explained as the load on the connection increases since each machine will be sending large quantities of data to the querying machine.

For 75% availability (program runs on 3 VMs ; represented by dotted grey in the graph) and 50% availability(program runs on 2 VMs ; represented by dotted grey in the graph) we observe that the trend for query latency vs frequency of patterns remains the same as in 100% availability. For higher order frequency patterns, we observed that the query latency decreases with decrease in availability. This can be explained, since the client has to handle comparatively less load. We suspect that the deviations from the general trend seen between O(1) & O(10) and O(1000) & O(10000) might be due to our program taking higher times for smaller patterns. The pattern lengths we have used are as: O(1) – 13, O(10) - 49, O(1000) – 11 and O(10000) – 16.