A1: Distributed Log Querying **Lucas Palazzi, Jiahui Tang**
CPEN 533

1 System Design Overview

To realize distributed log querying, we used a client/server architecture. The system has two modules: Server program and client program. the two modules communicate by message passing over TCP sockets. All the machines from which log files have to be queried must have the Server program running. The Server program always listens to the requests from Client. The Client program receives the grep query from the users and displays back the matching lines on the terminal.

The work-flow of our system is as follows: The user executes the client program on any virtual machine by using a command like ***./client \"PATTERN\"***. The client then takes the grep query and spawns multiple threads. Each thread transmits the grep query to a server. In the server, the grep query is executed locally and the resulting lines are transmitted back to the client. Eventually, the client collects and combines all the results, then prints to the user. The results contain line numbers and log file names. Figure 1 shows the overall structure of our system.

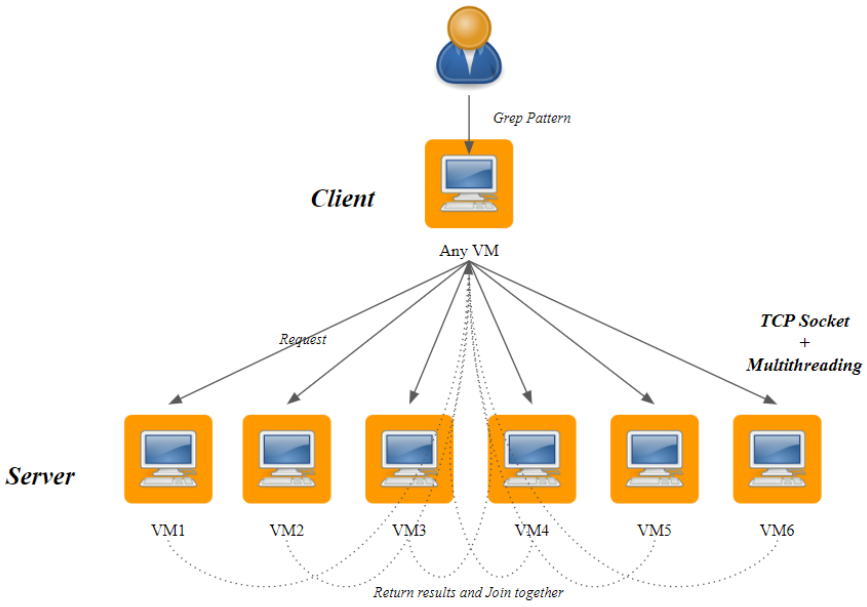

Figure 1: System Structure of Distributed Log Querying.

Here are some details about our implementation: The Server program is initialized on each virtual machine, which establishes TCP sockets (port number 5330) and waits for connection from the client. The client program has hard-coded IP address strings for all six virtual machines (AWS EC2 instances with Ubuntu 18.04 LTS). If any of these connections fail, we record the failure and then continue the execution of the code. The grep binary is executed with the -Hn options to give the line number and filenames. Once all of the threads and responses have completed, client program will combine the returned strings using pthread_join. We implemented our system in C++ and used g++ to help us build the executable binaries.

Since we have designed a multithreaded C/S mode, our system has a good performance, fault-tolerance, and scalability. All the grep can be executed at the same time on each virtual machine. If any of virtual machines failed to be connected, or the virtual machine is not running, our system still can return the results and print the failure status of the specific server.

2 Testing

To test our system, we implement a Python script that executes various different log querying requests for a variety of situations. The test cases we invoke include strings that occur:

- once on one machine,
- once on all machines,
- multiple times on one machine, and
- multiple times across all/some machines.

The test strings placed in the logs were formed using a variety of different characters. We chose not to focus on testing all of the different possible regular expression situations, as we assume that the locally executed grep command is working properly. We instead focused on testing the different cases of strings distributed across the log files to test the functionality of our system. The testing script also has support for testing when Servers are not online or have failed. We can then use it to test how the system responds if we manually take a Server offline.

3 Average Query Latency

The query latency is related to the query output on the Server program side. Generally, querying rare patterns will generate lower query results, then it will have lower query latency. As shown in the table below, we tried four different cases in order to get the average query latency. Each virtual machine (we have six VMs in total) has a 100 MB log file. For each pattern, we executed our system 5 times to calculate the average time.

#	***Pattern***	***Frequency***	***Result (Total Lines Matched)***	***Average Time***
1	AAAAAAAAAA	Rare	0 lines matched from 6 VMs	0.0564s
2	java	Rare	134 lines matched from 6 VMs	0.1294s
3	htm	Frequent	59447 lines matched from 6 VMs	1.4634s
4	com	Frequent	681563 lines matched from 6 VMs	30.0598s