Raft Performance Evaluation

Suman Nakshatri, Harsimran Pabla
David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
sdnaksha@uwaterloo.ca, hspabla@uwaterloo.ca

ABSTRACT

Raft is a consensus building algorithm for distributed systems designed for better understandability than Lamport's Paxos. While there are multiple implementations of the algorithm, to our knowledge, the original paper and any subsequent papers do not provide any evaluations with respect to Raft performance under different workloads. With this paper, we use the original open source implementation for Raft, LogCabin, and perform a series of experiments on different workloads and varying contention on the system. We attempt to justify the performance observed to the design decision made by Raft protocol.

1. INTRODUCTION

While designing a fault tolerant distributed system, consensus management becomes a key and fundamental problem. It is to agree on a single value between multiple servers under any failure model. Once agreed upon, the decision is intact and fixed. In this paper, our reference to failure model points to non-byzantine failures only. This need for consensus arises when we replicate state machines across servers to support fault-tolerance. While the client looks at this system as one server which responds to their requests, this consensus module ensures each server executes the same commands in the same order so as to produce the same result. Typically, this agreement process involves getting a majority of the servers to agree upon a value, which allows for continued operation even when a minority of the servers fail. One of the original solutions to consensus problem was Paxos [cite paxos] by Leslie Lamport. While it is highly popular and used in quite a few systems, it is considered to be a highly convoluted algorithm, which is difficult to understand, realized even further while implementing it as a real system. Furthermore, classical Paxos has many performance flaws, which were fixed in follow up papers such as Multi Paxos, E Paxos etc. While there are research papers proposing easier solutions for Paxos, many lacked mathematically or experimentally proven good and complete implementable solutions.

As a result, Raft was introduced to provide a simpler to understand, easy to implement solution for consensus building. The consensus problems, such as leader election, log replication, safety were decomposed into independent sub-problems for improving understandability, while making the algorithm more intuitive by reducing the degree of non-determinism and inconsistency amongst the servers. The original Raft research paper provided evaluations in three aspects: understandability , wherein students were made to take a Paxos and Raft quiz and their marks were compared; correctness, which provided a formal specification and a proof of safety; and performance, to measure its leader election operation. However, it did not evaluate Raft in terms of its execution time or failure tolerance under contention and under different workloads.

With this paper, we answer the below research questions:

1. How does RAFT system perform under contention and without contention?

2. How does RAFT perform on the following workloads: read-only, write-only, equal read-write workload.

The remainder of this paper is structured as follows: In Section 2, we gives a brief overview of the Raft consensus module. Section 3 covers the experimental setup and methodology used to measure performance. Section 4 details the results obtained from our experiments, their corresponding graphs and our insights into the design decisions made in Raft that could have caused the results. We provide potential future work in Section 5. Section 6 offers concluding remarks followed by acknowledgements.

2. BACKGROUND

Raft introduces the following significant features amongst others: strong leader; leader election and cluster membership changes. It implements consensus by first electing a leader. The protocol involves strong form of leadership in that the log entries only flow from the leader to other servers in the cluster, also known as the followers. Also, the leader never overwrites or deletes entries

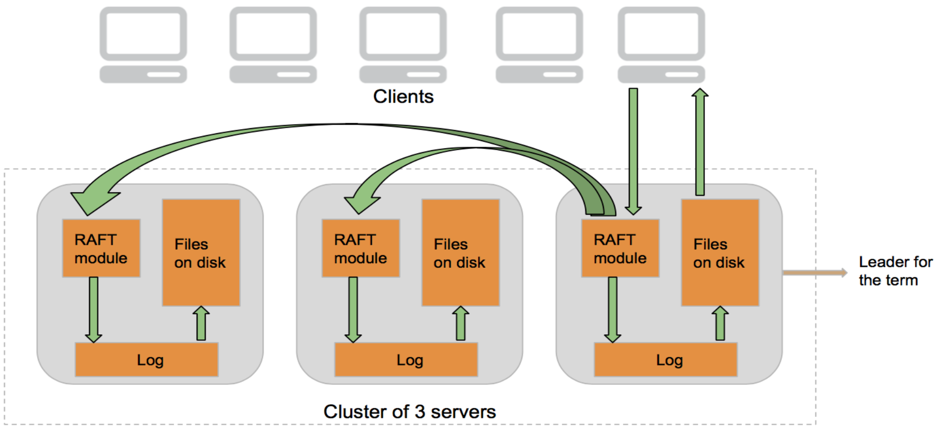

Figure 1: Experimental setup on DigitalOcean.

in its own log. This simplifies the management of replicated log and presumably makes it easier to understand the protocol. Raft divides times into terms and each term begins with leader election. It uses randomized timers to elect leader so as to resolve any conflicts. In practice, it works without having extra overhead of consensus agreement on leader. The servers in the cluster communicate using remote procedure calls(RPCs).

Write requests under normal operation works as follows : Once a leader has been elected, it begins servicing client requests. Each client request contains a command to be executed by the replicated state machines. The leader appends the command to its log as a new entry, then issues RPCs in parallel to each of the other servers to replicate the entry. When the entry has been safely replicated on a majority of servers, the leader applies the entry to its state machine and returns the result of that execution to the client.

Read only requests can be handled without writing anything to the log. However, before responding to read request, leader ensures that it is still the leader of the cluster. Raft handles this by having the leader exchange heartbeat messages with a majority of the cluster before responding to read-only requests.

Our performance evaluation is measured in terms of read and write request time taken by the client, under various workload scenarios. It is also important to note that all requests are serviced by the leader alone. The leader taken in requests from the client and replicates log entries across the cluster. Once an entry is applied to the state machine, no other server can apply a different command for the same log index. This ensures necessary safety in the protocol. Also, clients do not maintain any active information on who the current leader of the cluster is. Thus, there is a need for client communication protocol to resolve leader id and then direct the requests to this id.

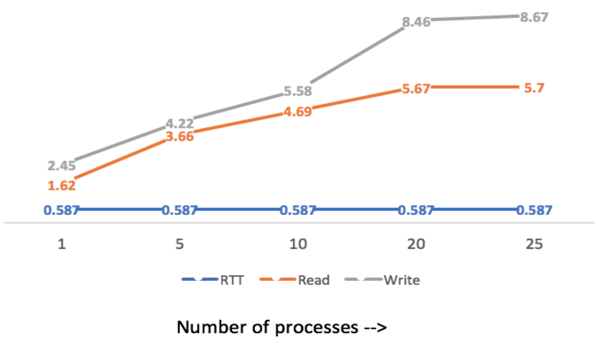

Figure 2: Execution time (in ms) of 1000 READ and WRITE requests generated from each process running in a single client. These requests perform action on the same file.

3. METHODOLOGY

Eight servers were rented from the cloud service platform, DigitalOcean, to perform experiments. <Write about server configuration>. We installed Stanford open source implementation of raft, LogCabin, in three of the servers and configured them to run as a Raft cluster. The remaining five servers were used as clients to issue read and write requests to Raft. Fig.1 gives the resultant setup of the model.

The state machine behind the cluster is a file system stored on disk. Each file in the file system stores one single value. These file structures are stored and accessed using tree operations, where the file name is accessed by traversing through the directory structure in the tree. This becomes similar to a key-value store, where the filename and structure is the key which stores a single value. Read requests read the value stored in the file, while write writes or overwrites the value in the

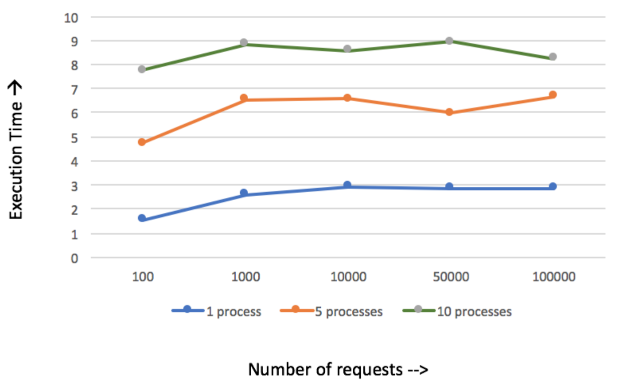

Figure 3: Execution time (in ms) of increasing number of READ requests generated from each process running in 5 clients. These requests perform action on the same file.

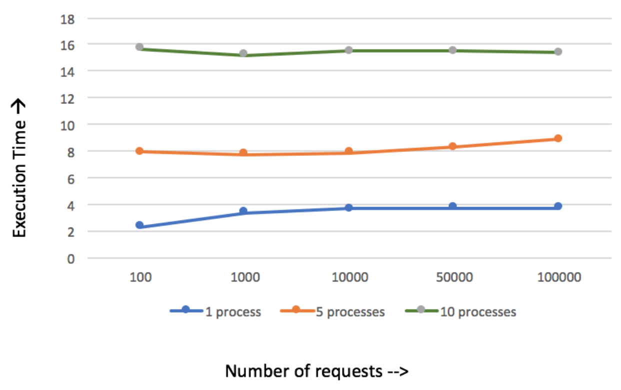

Figure 4: Execution time (in ms) of increasing number of WRITE requests generated from each process running in 5 clients. These requests perform action on the same file.

file.

We created read and write files which performs a single read/write or a sequence of reads/writes. Concurrent requests were issued by creating one or multiple processes of these executable read and write files. Hence, if we ran 5 read processes in 5 clients at the same time, each issuing 100 sequential reads, we throw in 25 requests at one time until 100 of such concurrent requests are issued sequentially.

4. EVALUATION

In this section, we present our results for the different experiments performed by varying contention and workload types.

4.1 Single client

In the first set of experiments, we wanted to measure the time taken for one single read and write request and benchmark it against the round trip time(RTT) for a simple ICMP message. The first request issued from any client process is directed to a random server chosen from the list of servers in the Raft cluster. Hence, it may either directly reach the leader or reach a follower and get re-directed to the leader. We observed that a single read request took 3-5ms if the request reached the leader directly, and 4-11ms on a miss, while a single write request took 2-4ms on hitting the leader and 3-10ms otherwise. Because there was a huge range in the results obtained for a single request, it was only natural to evaluate the total time taken for a thousand requests to obtain the average for a single request. Fig.2 gives the average read and write time while running increasing number of processes in one client and 1000 requests in each of these processes. While a single read took more time than a single write, we observe that the average read time is consistently lesser than the write time. While, read calls a majority of other servers to ensure it is still the leader, a write call does extra work of appending the log entry on a majority of other servers. Hence, a write call taking more time than read is as expected considering the algorithm design. Also, with the increase in number of processes, the average read and write time increase almost linearly. The important point to note here is that the ICMP RTT is 0.587 and hence, the difference of 1.1ms for a read and 1.9ms for a write is the overhead Raft takes for its protocol execution.

4.2 Multiple clients

In the second set of experiments, we measured the affect of issuing a large set of sequential requests on Raft. For this, we issued sequential requests ranging from 100-100000 using five clients. Six experiments were performed by changing the number of concurrent requests to 5, 25 and 50(1, 5 and 10 processes in each client), for a workload of 100% read and 100% write.

Fig. 3 shows the result for running increasing number of sequential results on Raft using 5, 25 and 50 concurrent requests for a workload of 100% reads, while Fig. 4 shows the same for 100% write requests. Both the graphs show a very similar pattern, except that average write time at every point is greater than the average read time for the same number of requests and processes. The graphs also clearly show more the number of concurrent requests, higher is the average read and write time taken. However, increase in the number of sequential requests has no effect on the execution time for both reads and writes. Hence, Raft is robust under low concurrency and high request bombardment. This naturally brought us to the next set of experiments which dealt with increase in concurrent requests to Raft, while keeping the sequential requests consistent.

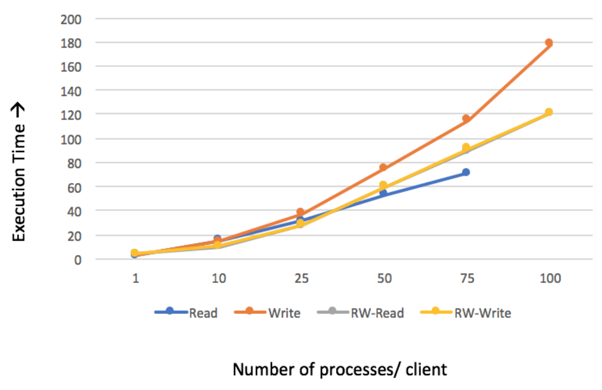

Figure 5: Execution time (in ms) of 1000 READ and WRITE requests generated from each process running in 4 clients. These requests perform action on the same file.

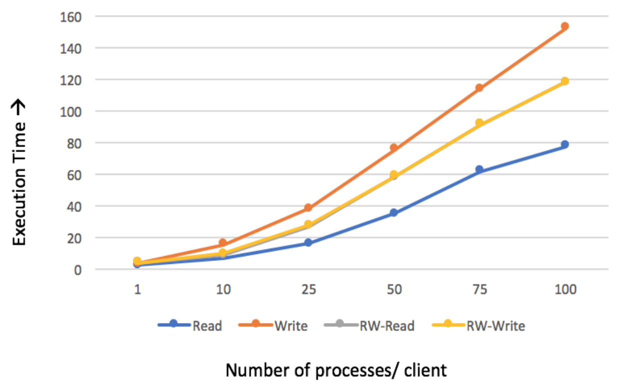

Figure 6: Execution time (in ms) of 1000 READ and WRITE requests generated from each process running in 4 clients. These requests perform action on different files.

In the next phase, we used 4 clients and increased the number of processes running in each client so as to load Raft with higher number of concurrent requests. Fig.5 displays the result for running 1000 requests in each process and increasing the number of processes from 1 to 100 in each client. In this and all previous experiments, all actions of read and write were done on the same file. However, we also wanted to judge the performance when the actions were performed on different files, to eliminate extra concurrency overhead associated with potential file locks.

5. CONCLUSIONS

6. ACKNOWLEDGMENTS

We would like to thank Prof. Samer Al-Kiswany for his guidance during our brainstorming and initial project discussions.

7. REFERENCES