Testing Reliability of Quorum Systems

This task is designed for students who are interested in the DepFast research project.

To select candidates, we are assigning a **challenging** technical task. If you cannot make progress now, please feel free to contact us later (best by email) when you gain more experience with computer science. If you make good progress, we can consider your application for our research groups. We hope that you will **enjoy** the challenge and **learn** valuable information, even if we may not select you.

You should do the task **individually** but feel free to search online (e.g., on StackOverflow) for any problems you may run into.

We understand that the task is challenging, and that you may run into problems. Do your best to try to resolve them. Research is open-ended by nature, and you won't always have someone available to help, hence why this task is good for us to evaluate students :)

Task

In this assignment, we will write code to test the reliability of a quorum system. Quorum systems implement Replicated State Machines (RSM) that are linearizable, fault-tolerant groups of replicas coordinated using a consensus algorithm (e.g., Paxos, Raft, etc). Given their properties of fault tolerance and strong consistency, quorum systems are at the core of large-scale distributed infrastructures. Taking Kubernetes (a large-scale cluster management system) as an example. Kubernetes uses etcd, a quorum system that implements Raft, as the centralized data store to maintain the cluster states.

In this assignment, we will write code to test the reliability of quorum systems. We will evaluate whether real-world quorum system implementations deliver the fault tolerance properties that are promised by the consensus protocols.

In fact, our research shows that real-world quorum system implementations often fall short. For example, Yoo et al. made the following observation – "*existing RSM system implementations cannot consistently tolerate fail-slow faults on one follower node.*" Take RethinkDB, another Raft implementation, as an example. They report that if they slow down a follower node in RethinkDB, the leader will crash (yes, it is true!). This is counterintuitive, because the consensus algorithm is supposed to tolerate a minority of faulty nodes.

1. Select a Quorum System

You can select **any** quorum system. If you do not have an idea what system to pick, you can use a Raft implementation listed on the Raft website.

You are welcome (and encouraged) to select systems built on other consensus protocols (e.g., Paxos, MultiPaxos, EPaxos, Copilot, etc).

Q1: What is your quorum system of choice?
RethinkDB

2. Run The Quorum System and Measure The Baseline Performance

You are responsible for compiling, building, and running the quorum systems you decide.

You can run your quorum system on one machine in a pseudo-distributed mode, where each node runs as a process connected through the local loopback. **All the experiments in this assignment can be done with a pseudo-distributed setup.**

Once you have a quorum system, find a client workload so that you can measure the performance of your quorum system. You should be able to find a client workload from the source-code repo, because developers need them to test their systems, too.

After all the hard work, you have a quorum system running happily on your machine.

Q2: Please describe your configuration.

I set three cpu as server node and the rest one as client.

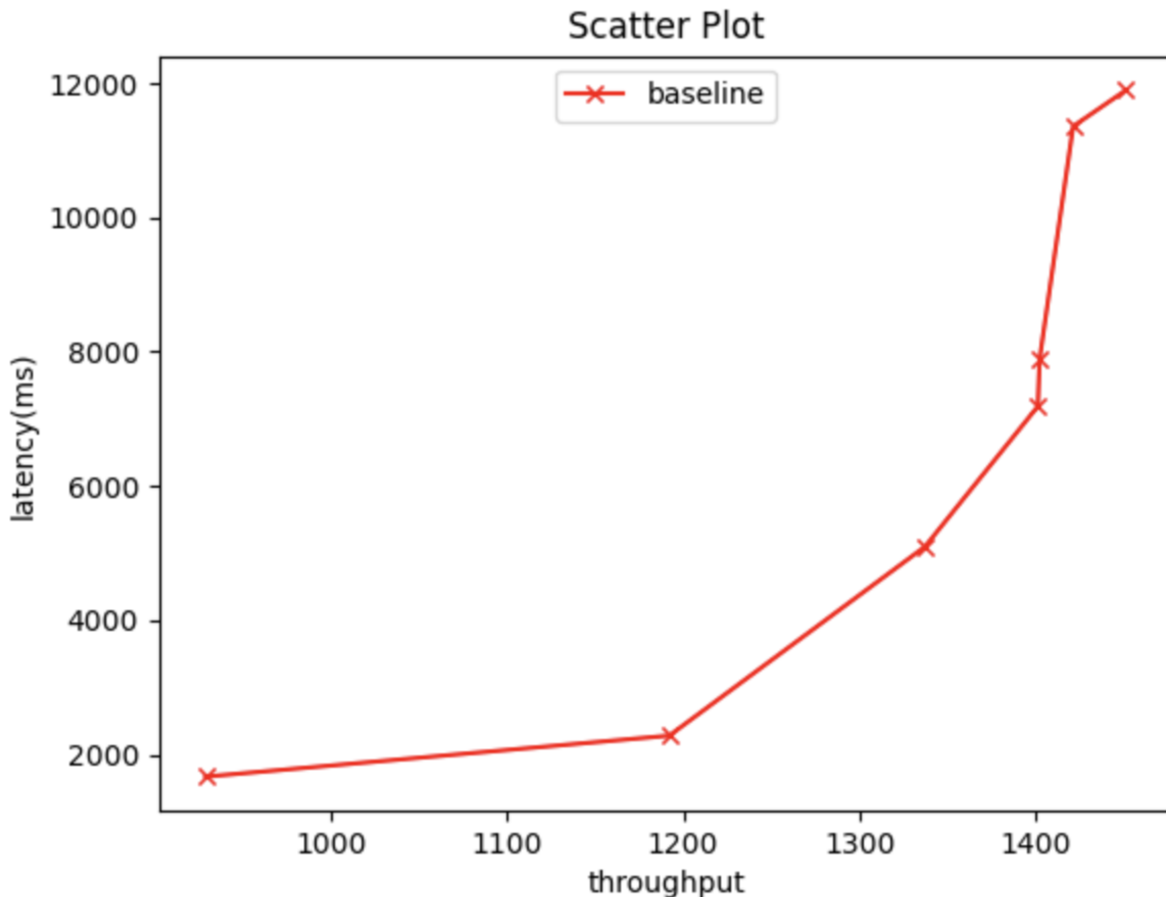
Q3: What is your client workload?

I choose workloada in YCSB as client workload. I set operation count to 250000 and update ratio to 1.0 to make the result distinctive.

Please run the client workload on your quorum system and record the performance, which is referred to as *baseline performance* (i.e., performance without any faults).

Q4: What is your baseline performance? Plot the throughput-latency figure (how does such a paper look like? the x-axis is the throughput and the y-axis is the latency, see Figure 7 in this paper). The latency should be average or P50 latency.

After much testing I found out that when the number of clients is around 14, the performance is the highest. I also tried out numbers like 256 and 512 just like mentioned in the paper and the performance didn't show any improvement.



3. Fail-Injection Testing

Now, let's assess the fault tolerance of the quorum system of choice. The way we are going to do it is to inject the following types of faults **during the client workload** into a node:

- Crashing behavior
- Slow CPU
- Memory contention

We then measure the performance in the same way you measure your baseline performance. We will compare the performance with faults with the baseline performance (without faults). The difference indicates the fault tolerance level – ideally, there is no difference.

Please inject the above three types of faults into both a leader node and a follower node respectively. So, you will have the following 6 different cases:

- Crashing behavior on leader
- Crashing behavior on follower
- Slow CPU on leader
- Slow CPU on follower

- Memory contention on leader
- Memory contention on follower

Later, you will report the performance with the above faults and compare it with the baseline performance in each scenario.

You can choose how to simulate the three types of faults above, but you need a programmable way rather than manually doing things.

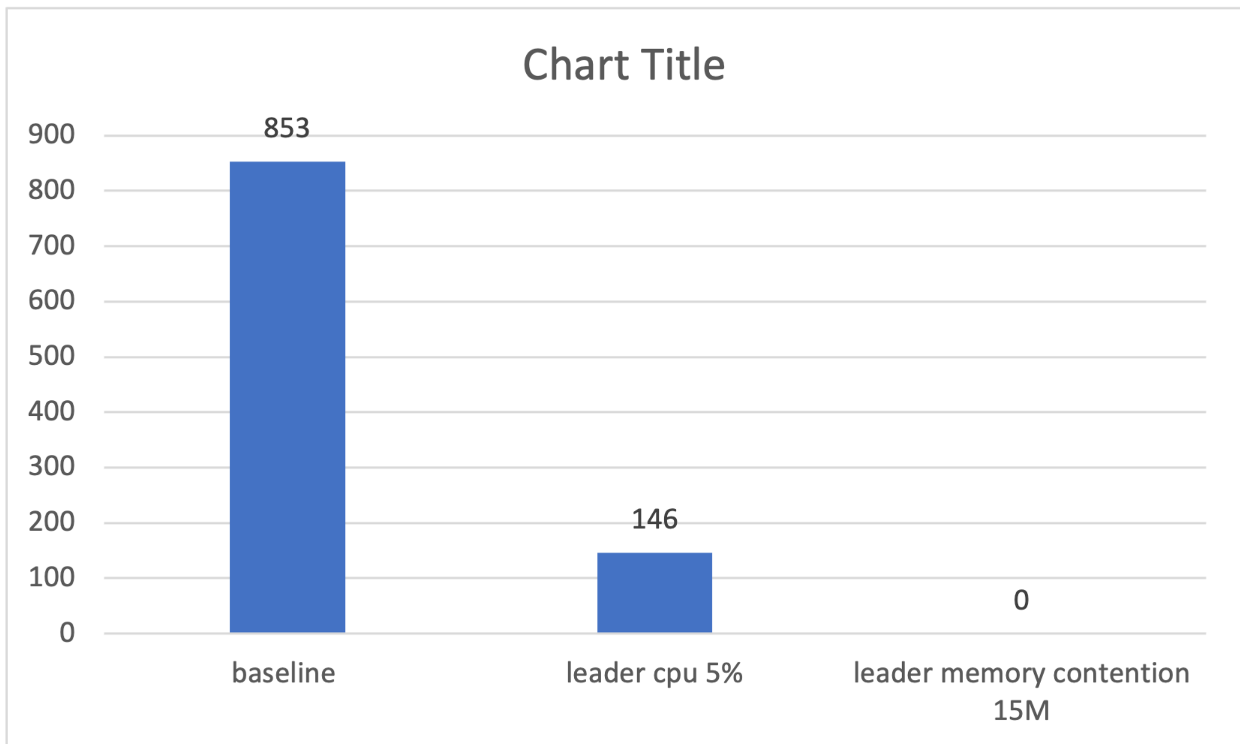
We developed a [xonsh]-based fault-injection tool named [Slooo] which already implements the faults in a simple framework, https://github.com/xlab-uiuc/slooo/tree/main/faults

Please use Slooo to implement the fault injection so that everything is codified and easy to manage.

Q5: How do you simulate crash, slow CPU and memory contention?

Use cgroup to restrict cpu usage to simulate slow CPU. Use cgroup to restrict the memory usage of processes.

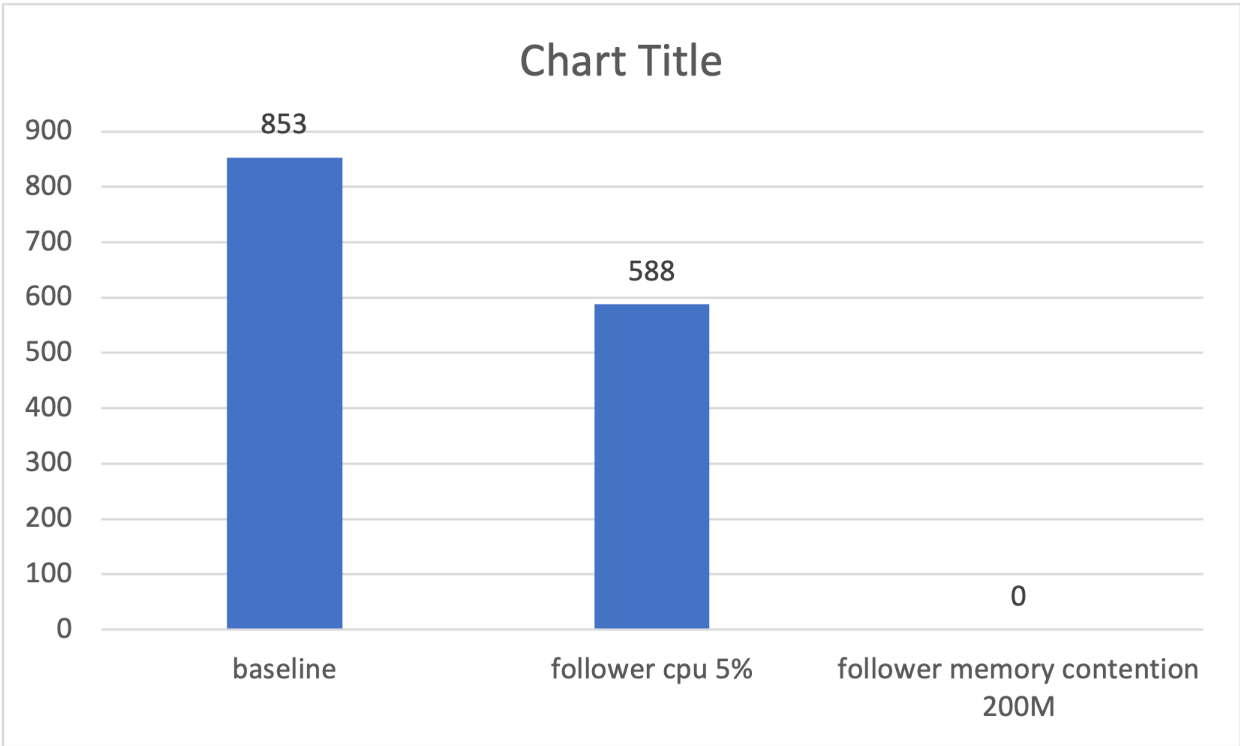
Q6: Please plot the performance with faults on the *leader* node and compare it with the baseline performance.



Q7: Please explain the above results. Is it expected? Why or why not? You will receive bonus points if you are able to pinpoint the code.

When faults injected in leader node, the performance degraded significantly. As shown in the graph, the latency is much higher and the throughput is much lower under the condition of the same number of clients and the same workload. The result is expected, the performance should be debased since leader is responsible for state replication and send PRC to the follower periodically. Since there is only one leader at a specific term, once a leader node become slow the latency will increase.

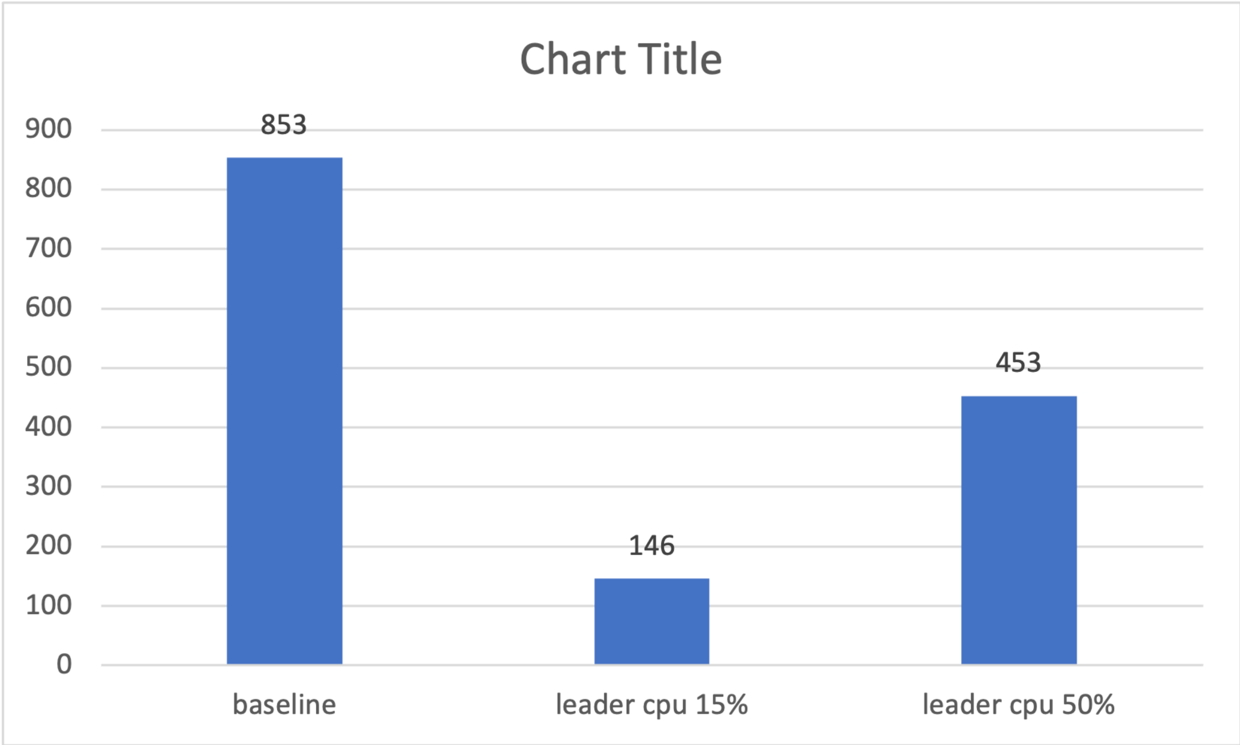
Q8: Please plot the performance with faults on the *follower* node and compare it with the baseline performance.



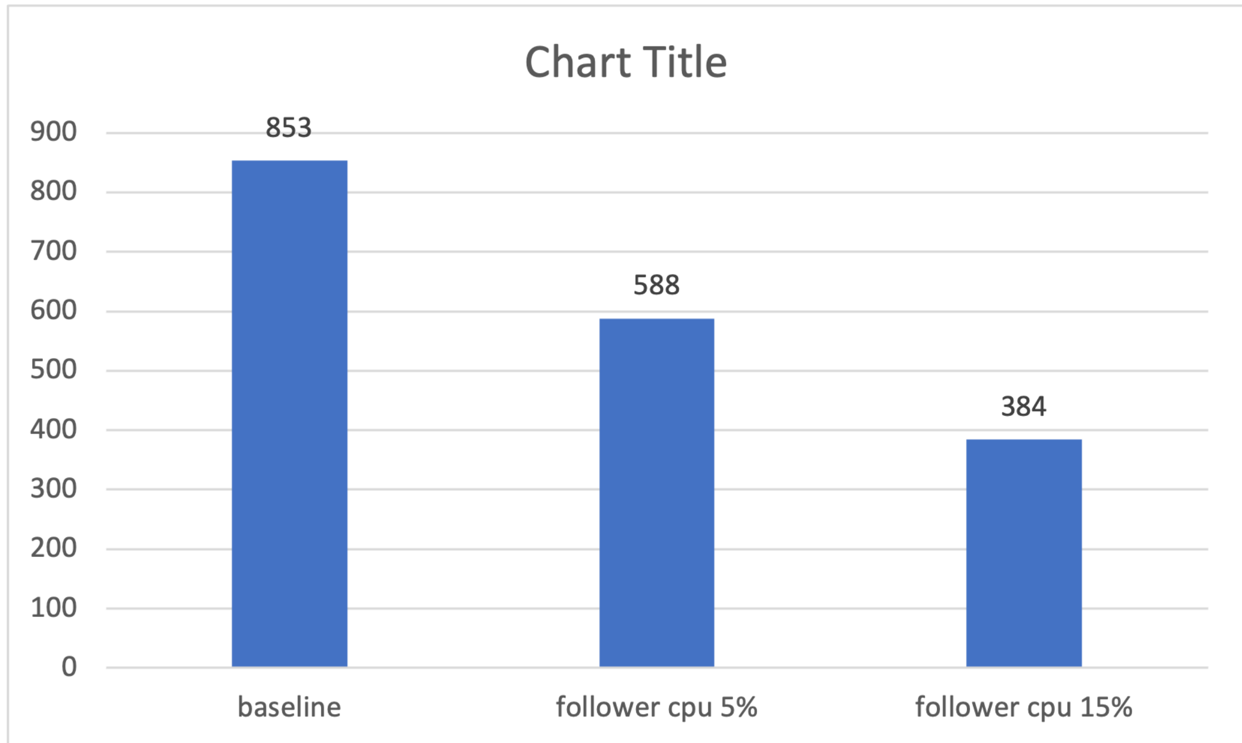
Q9: Please explain the above results. Is it expected? Why or why not? You will receive bonus points if you are able to pinpoint the code.

The graph above shows when the follower node becomes slow, the performance degrades severely. The result is not as expected. Theoretically, when a minority of followers become slow, it will not affect the performance of the whole system since the quorum system is fault tolerance for minority node faults.

Q10: For the slow CPU and memory contention, could you vary the level of slowness/contention and report the results?



When the percentage usage of cpu decrease, the latency increase and the throughput decrease.



When setting memory limit to 15M, 20M, 200M both follower and leader result in timeout.

Submission Instruction

Write your answer for Q1–Q10 (with figures) in a PDF file. Please put your code and answers in a repository on GitHub and tell us (tyxu) about the repository link.