Assignment #2: Testing Reliability of Quorum Systems

Due: April 30 (Saturday), 23:59 pm CST

In this assignment, we will write code to test the reliability of quorum systems. 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, 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, we prepared a list for you. Please select one from the list and put your name down. Note that all the systems on the list are Raft implementations. You are welcome (and encouraged) to select systems built on other consensus protocols (e.g., Paxos, MultiPaxos, EPaxos, Copilot, etc).

Q1: So, what is your quorum system of choice?

Write your answer here.

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.

Write your answer here.

Q3: What is your client workload?

Write your answer here.

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.

Write your answer here.

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.

Tianyin's group 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

You are welcome to use Slooo which implements most stuff already (it also helps the team to fix bugs and improve the Slooo tool). It is also perfectly fine if you don't want to use Slooo, but to implement something similar yourself. You should choose what helps you the best.

Note that we do not grade your scripts/tools, but the results.

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

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

Write your answer here.

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.

Write your answer here.

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

Write your answer here.

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.

Write your answer here.

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

Write your answer here.