COL733: Fundamentals of Cloud Computing

Lab-3: Network partition

Dipen Kumar (2018CS50098)
Manoj Kumar (2018CS50411)
Sanjali Agrawal (2018CS50419)

AvailableRedis:

Give reasons for the correctness of your implementation. Cover all failure scenarios in your analysis: celery worker failure, Redis instance failure, and Redis network partitions.

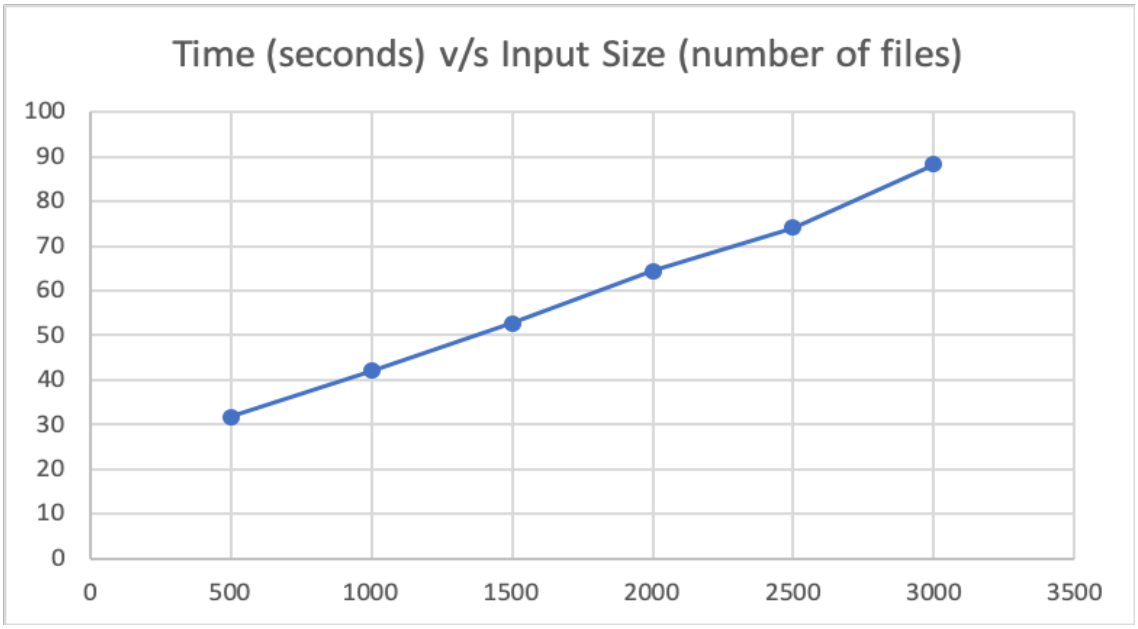
Our Implementation add a task in RabbitMQ for workers to process one file per job. Once the file is processed by the worker, we create a dictionary of words and its count which we convert to string with json dumps function. We then create a tuple of filename, word count dictionary in string format which we further encode in string and the add this value in redis set at in all three redis server running in three different nodes. We say our task is completed only when it writes to at least two redis servers else it keeps on trying.

1. Correctness against celery worker failure:
Since I am dumping a filename, value pair in redis set, My task is idempotent. That is if celery worker fails which means job is reschedule by RabbitMQ which means another worker will process the same job which means it will again write filename, value pair in redis set. Now if it was earlier not written then it will be written now else no problem it will again write with no effect/change in previous value. Our implementation will also handle stragglers. It will also handle network partition between RabbitMQ and celery worker. All these are possible because we designed idempotent tasks.

2. Correctness against Redis instance failure:
As long as a worker is able to write to two redis instances it will return and pick next job. In case of redis instance failure if we have two available redis for the worker it will successfully return else it will keep waiting and keep trying to at least write to two redis instances. Once a file is successfully written by the job => it is present on at least two instances. And during read time if at max one redis is failed then we can still read all successful writes else some successful writes are lost.

3. Correctness against Redis network partitions:
As stated in previous part, as long as a worker is able to write to two redis instances it will return and pick next job. In case of redis network partition if we have two available redis for the worker it will successfully return else it will keep waiting and keep trying to at least write to two redis instances. Once a file is successfully written by the job => it is present on at least two instances. When the partition heals, we do run an algorithm that will bring all redis instances to consistent state.

Plot the time taken to achieve strong eventual consistency vs the input size (N) after the network partition is healed.



What happens if there is a full partition, i.e., all 3 VMs can't talk to Redis instance on each other?

In above mentioned situation, all workers at a particular node can only communicate with the redis instance running at the same node. That means all worker can only write to just one redis instance i.e. write on at max 1 redis instance => write is not successful because for successful write it must write to at least two instance => it will keep trying => our program is waiting for the partition to heal up. Work done would be zero during that time. Waste of resources.

ConsistentRedis:

If you were instead using a consistent Redis, which implementation performs better in the absence of network partitions?

In absence of network partition, consistent Redis would perform better. This is because no partition => writes are always successful for consistent redis because all instances are available and of course for available redis it is also successful. But during read time available redis also have a eventual consistent algorithm which time to time runs and ensures eventual consistency. even if in case of no network partition when all redis instances are consistent for available redis, available redis will still run the algorithm and cross check to confirm that all redis instance are consistent. Whereas consistent redis don't have such eventual consistency algorithm because it always ensure consistency by failing writes which are not written to all instances. In case of the absence of network partition no case will arise to fail writes and hence consistent redis will perform better than available redis.

Which implementation is expected to perform better when there is a majority-minority partition, i.e., only 2 Redis instances can talk to each other but 1 can't talk to any of the other 2?

In this case available redis will perform better because all writes in consistent redis will fails during such majority-minority partition because no node is connected to every other node, hence no worker at any node can write to all redis instances => write is failed => waste of resources. workers are running and failing and work is not been done. Whereas in case of available redis, all workers on nodes in minority partition will fail writes because they can only write to one redis instance i.e. redis instance running on the same node, but still all other workers running on nodes in majority partition will be working and successfully writing since they can write to two redis instances i.e. one redis instance running on its own node and other redis instance running on the other node in majority partition (partition of two nodes). Hence in available redis not all resources are wasted. 2/3 of the recourses are still producing output while only 1/3 is wasted unlike consistent redis where whole resources was wasted.