Technische Universität Berlin

ENTERPRISE COMPUTING – ASSIGNMENT 2
WINTER TERM 2019/20

Ankush Sharma
415319

TELEKOM INNOVATION LABORATORIES

Service-centric NETWORKING

Replicate() and ReplicateData()

After client calls put(), replicateData() is called to replicate between nodes. This function calls replicate() that writes the data to memory of other nodes as shown in the picture.

If the qwritesize is 1, the response would be immediately sent to the client. Else, a thread safe variable (cnt) is used to count the number of responses received from the StreamObserver that stores the response array from other nodes. When this variable (cnt) equals qwritesize, the success response would be sent to client. If the timeout is more than 20 seconds or if cnt is not equal to qwritesize, failure response would be sent

```
if (qwritesize > 1) {
    for(int i=0;i<stubs.size();i++) {
        stubs.get(i).replicate(req,repObs);
    }

    try {
        int temp =0;
        int cnt=1;
        synchronized (repResponses){
            long startTime = System.currentTimeMillis();

            while (repResponses.size()<=(qwritesize) & cnt<=(qwritesize)){
               System.out.println("waiting");
                if (temp<repResponses.size()){
                    cnt++;
                }
                temp = repResponses.size();
                //Thread.sleep(21000);
                long endTime   = System.currentTimeMillis();
                float sec = (endTime - startTime) / 1000F;
                if (cnt==(qwritesize) & sec<20){
                    System.out.println(sec + " seconds");
                    System.out.println("final size:"+repResponses.size());
                    return true;
                    //break;

                }
                if (sec>20){
                    break;
                }
```

```
public void replicate(de.tub.ise.KeyValuePair request,
                      io.grpc.stub.StreamObserver<de.tub.ise.Response> responseObserver) {
    // TODO handle replication requests from other nodes
    String key = request.getKey();
    String value = request.getValue();
    Response response;

    logger.debug("Received replicate request with key " + key);
    Memory.put(key,value);
    response = Response.newBuilder().setSuccess(true).setKey(key).build();

    responseObserver.onNext(response);
}
```

getReplica() and gatherdata

- After client calls get(), gatherData() is called to replicate between nodes. This function calls getReplica() that reads the data from memory of other nodes as shown in the picture.

- If the qreadsize is 1, the response would be immediately sent to the client. Else, a thread safe variable (cnt) is used to count the number of responses received from the StreamObserver that stores the response array from other nodes. When this variable (cnt) equals qreadsize, the success response would be sent to client. If the timeout is more than 20 seconds or if cnt is not equal to qreadsize, failure response would be sent

```
    try {
        synchronized (gatResponses){
            long startTime = System.currentTimeMillis();
            int temp =0;
            int cnt =1;
            while (gatResponses.size()<=(qreadsize) & cnt<=(qreadsize)){
                System.out.println("waiting");
                if (temp<gatResponses.size()){
                    cnt++;
                }
                temp = gatResponses.size();
                long endTime   = System.currentTimeMillis();
                float sec = (endTime - startTime) / 1000F;
                if (cnt==(qreadsize) & sec<20){
                    System.out.println(sec + " seconds");
                    System.out.println(gatResponses.size() + " responses");
                    String data = Memory.get(key);
                    return KeyValuePair.newBuilder().setKey(key).setValue(data).build();
                    //break;
                }
                if (sec>20){
                    break;
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
} else{
    // TODO send async. replication requests to nodes
    String data = Memory.get(key);
    if (data == null) {
        logger.warn("Couldn't find data for key " + key);
        return null;
```

```
@Override
public void getReplica(de.tub.ise.Key request, io.grpc.stub.StreamObserver<de.tub.ise.Response> responseObserver) {
    // TODO handle request for local replica from ther nodes
    String key = request.getKey();
    Response response;

    logger.debug("Received get replicate request with key " + key);
    Memory.get(key);
    response = Response.newBuilder().setSuccess(true).setKey(key).build();
    responseObserver.onNext(response);
}
```

Replication Strategy

- For this experiment we have tuned the values of N, R and W to achieve their desired levels of performance, availability and durability
- However, it can be clearly seen that Low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by most of the replicas
- (3,2,2). have always been chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs which we also validated during our experiments.
- We tested our system with three different quorum-configurations (N,R,W):
  - (3,1,3)
  - (3,2,2)
  - (3,3,1)

Benchmarking

- In client.java,Faker() has been used to generate 200 random values which are stored in names
- Random() has been used to generate 200 random 3-digit numbers for keys which
- Put(): These randomly generated values stored in ids and names are used as arguments for client.put()
- Get(): 100 ids from these 200 are randomly selected and used for client.get()
- Similarly We used such system by changing values of quorum and evaluating system

```
Faker faker = new Faker();
List<String> names= new ArrayList<>();
List<String> ids= new ArrayList<>();
Random rand = new Random();
for (int i=0;i<200;i++){
  String name = faker.name().fullName();
  int id = rand.nextInt( bound: 900) + 100;
  ids.add(Integer.toString(id));
  names.add(name);
}
try {
  //put
  for (int i=0;i<200;i++) {
    client1.put(ids[i], names[i]);
  }
  //get
  for (int i=0;i<100;i++){
    int getid = rand.nextInt( bound: 200);
    client1.get(ids[getid], names[getid]);
  }
```

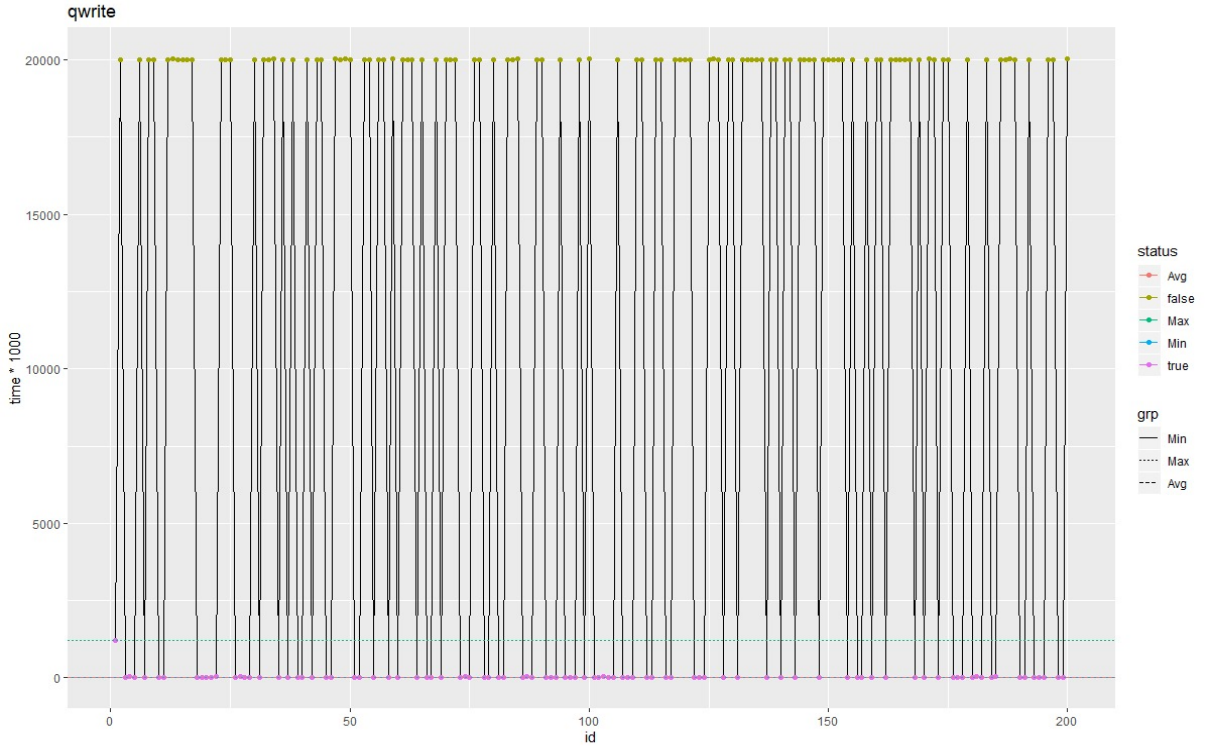
Results and Analysis

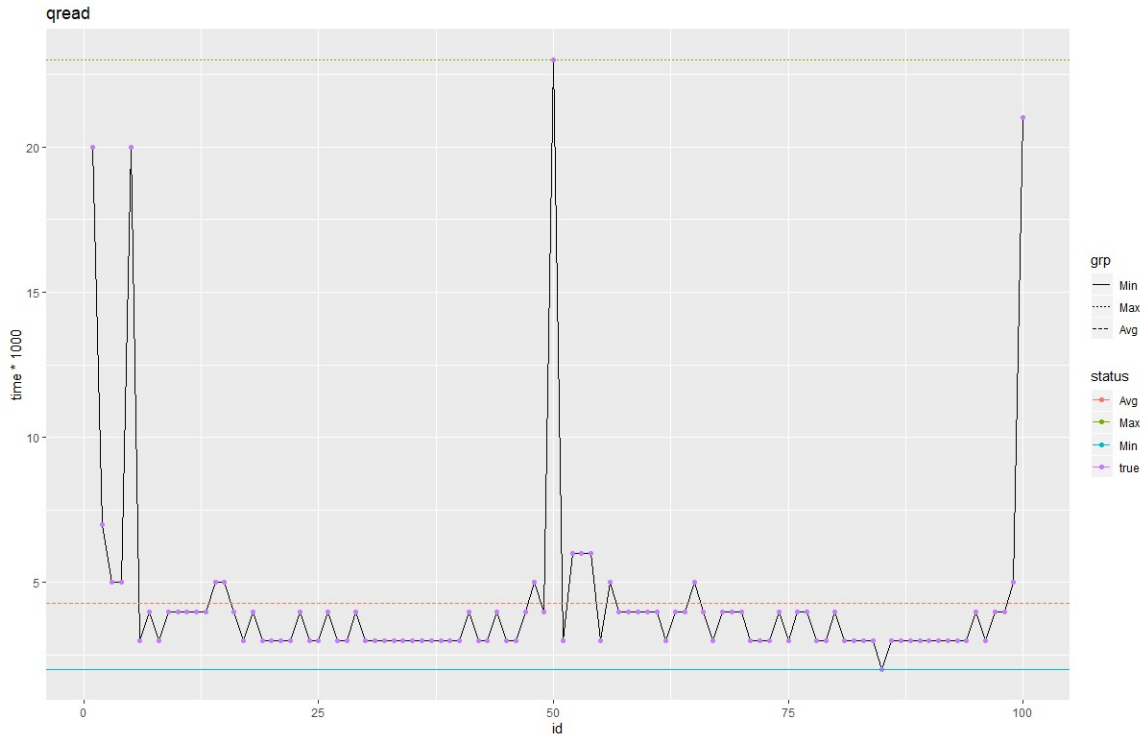
Method	Numbers
Put	200
Get	100

To run the analysis, 200 write request,100 read requests have been generated During this analysis, we recorded few observations as failed(more than 20 secs).To perform further analysis on latency we considered only the successful calls.

Results and Analysis for 1R,3W

Quorum	Min	Max	Average	Failed
Put	5	1221	23,9	106
Get	2	23	4,3	0

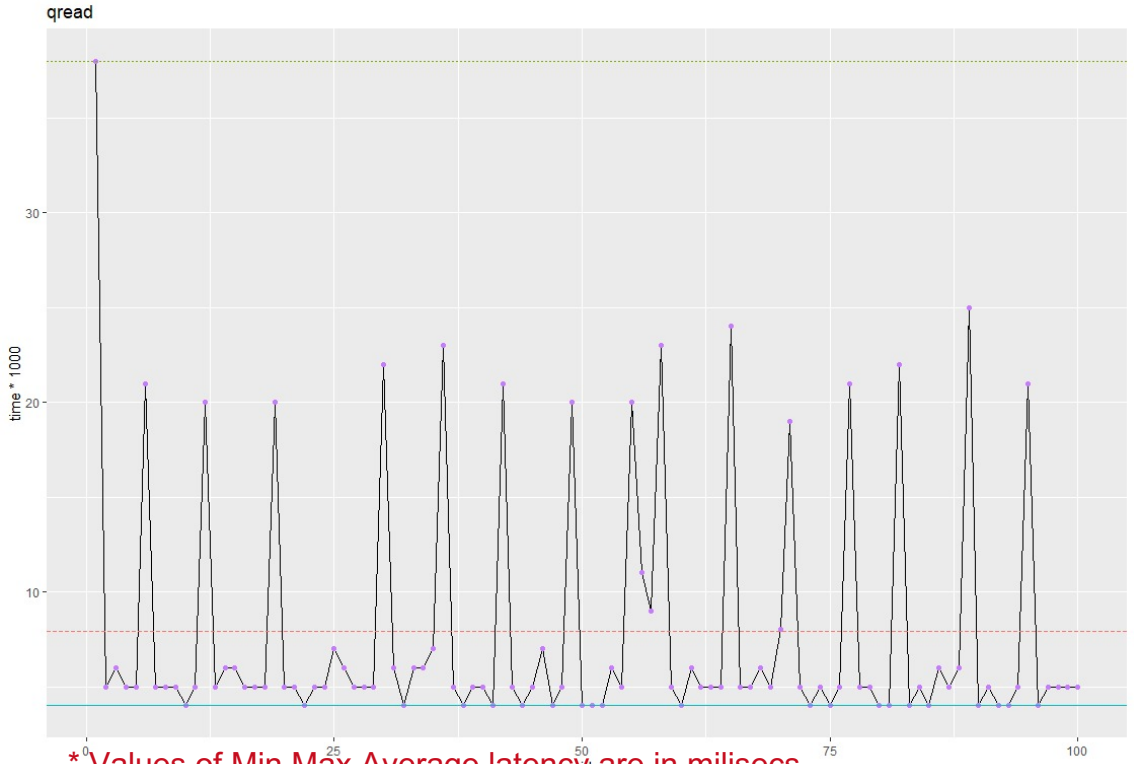


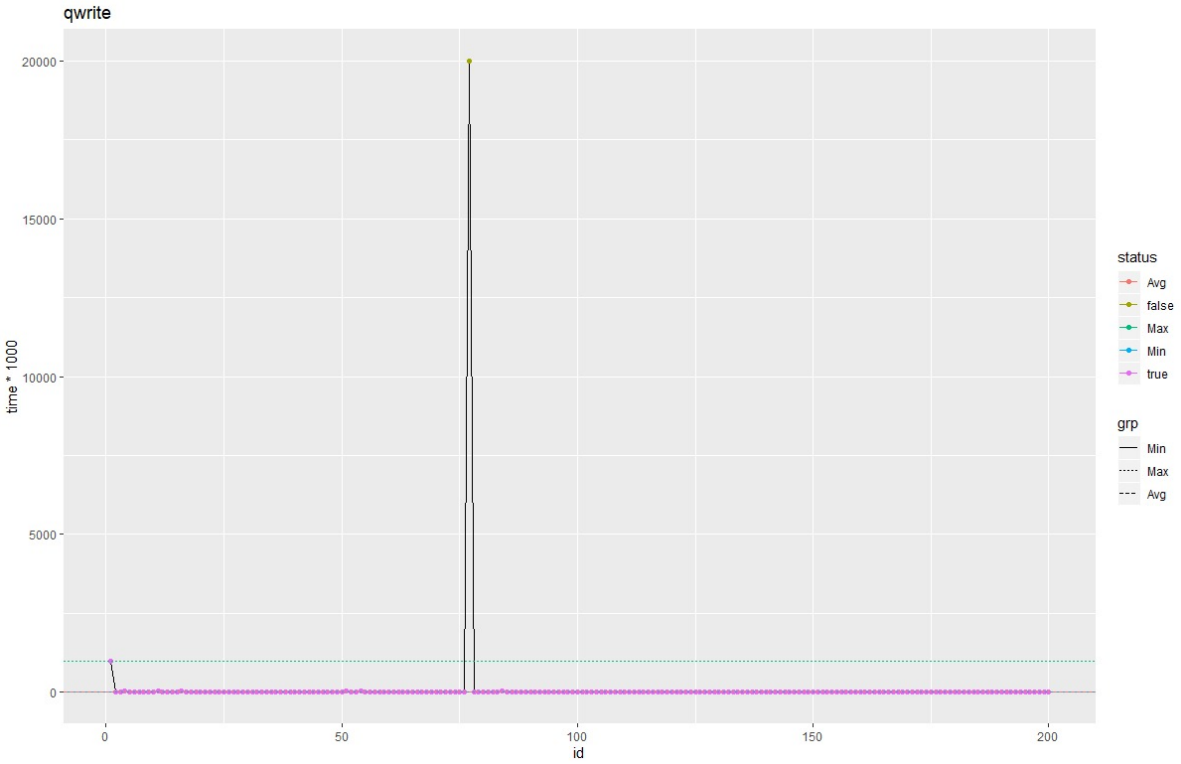


* Values of Min,Max,Average latency are in milisecs

Results and Analysis for 2R,2W

Quorum	Min	Max	Average	Failed
Put	4	997	15,6	1
Get	4	38	7,9	0

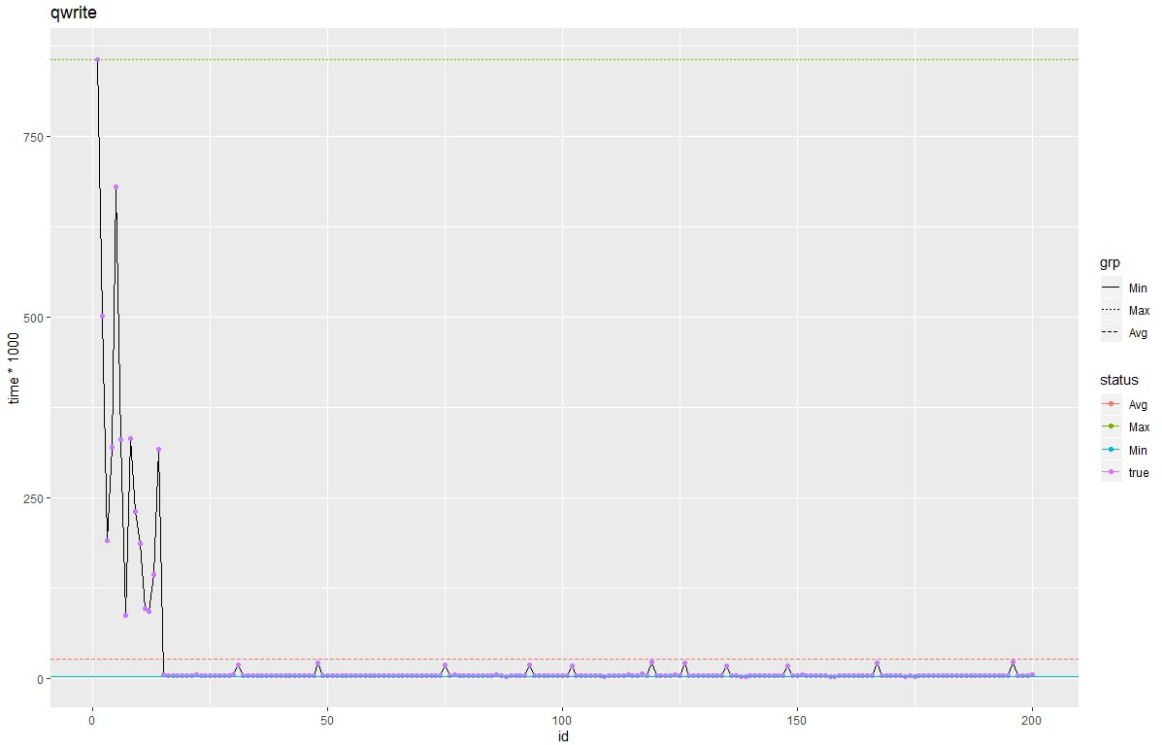


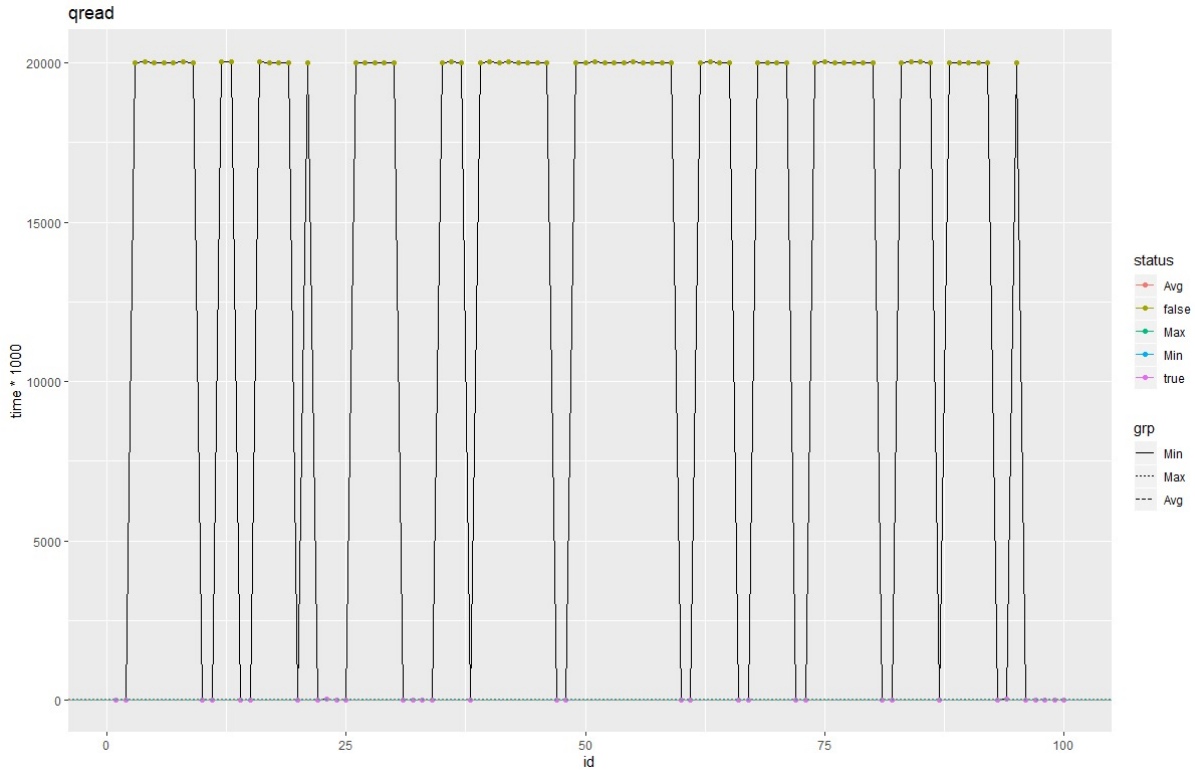


* Values of Min,Max,Average latency are in milisecs

Results and Analysis for 3R,1W

Quorum	Min	Max	Average	Failed
Put	2	856	25.87	0
Get	3	38	8.5	66



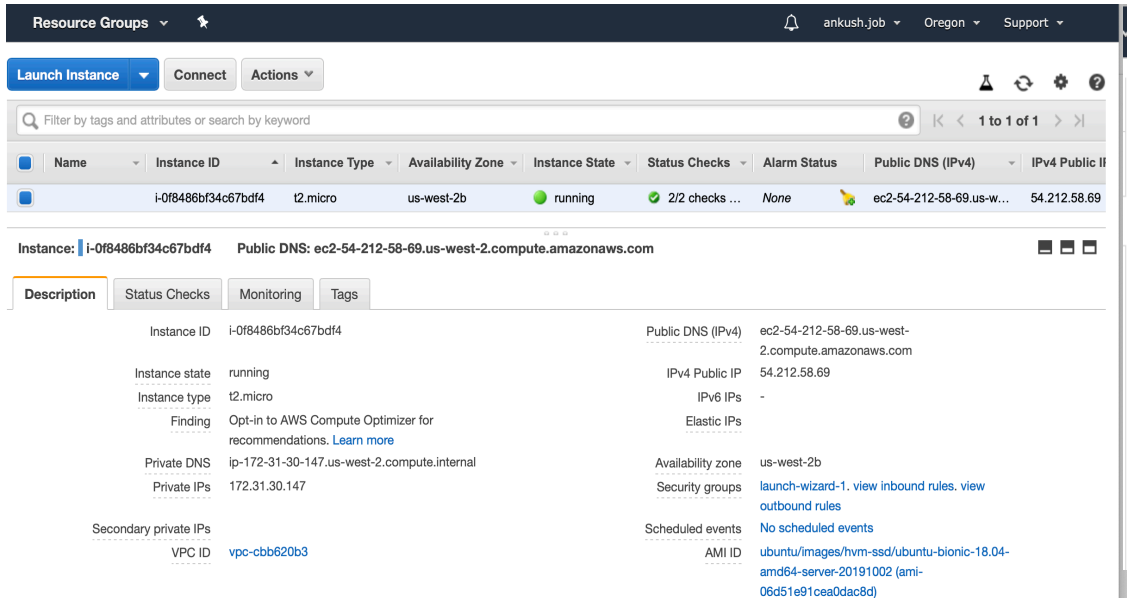


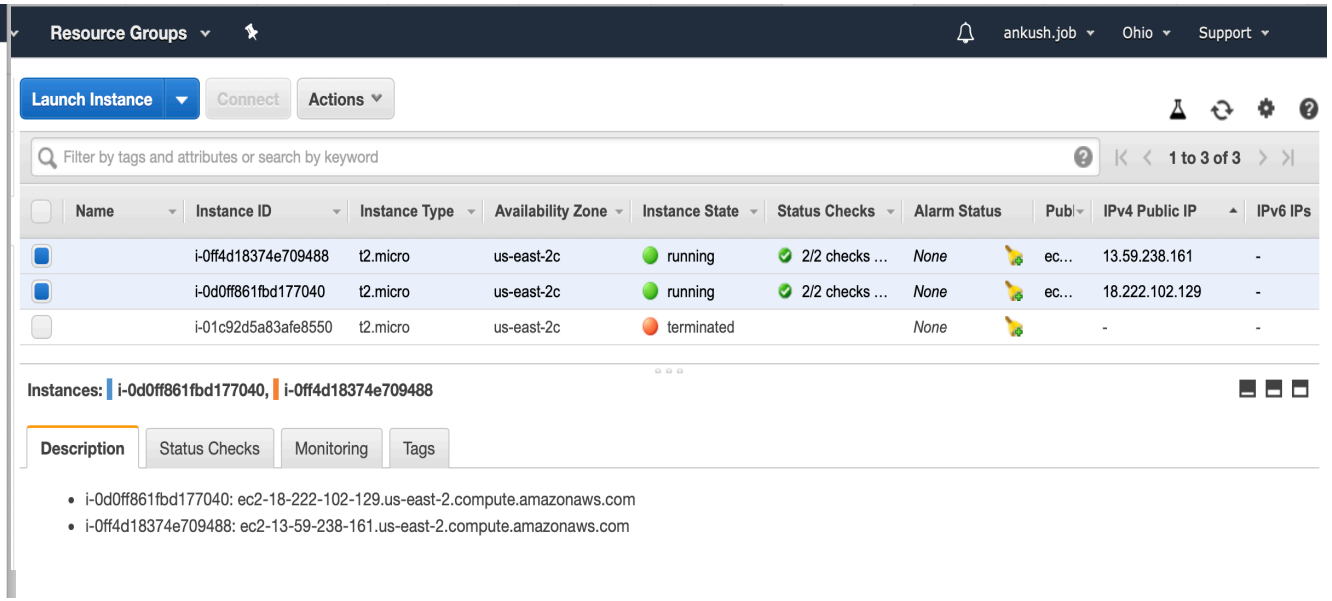
* Values of Min,Max,Average latency are in milisecs

Results and Discussion

- We can check and validate from the above results that quorum-configurations (N,R,W) as (3,2,2) is the most efficient in terms of latency and less failed calls

AWS Cloud Console





For this assignment ,I have used 2 Instances in **US East (Ohio)**us-east-2 Zone (with Public IP Addresses 18.222.102.129 & 13.59.238.161) and 1 instance in **US West (Oregon)** us-west-2 (Public IP Address :- 54.212.58.69)