Instructions

In the previous assignment, you implemented a distributed system that extended a single-site key-value store with *replication* in order to make it tolerant to faults such as node crashes and network partitions. To put it another way, we added more computers to make the system more resilient. Distributed systems may be hard to program, but it can be worth it!

Resilience is not the only reason, however, to take on the complexity of distributed programming. By adding more processing power to our system in the form of additional nodes, we can sometimes perform the same amount of work faster (this is sometimes called speedup), or gain the ability to do more work (this sometimes called scaleup or scaleout) than a single machine could do, by dividing the work across machines in an intelligent way.

The trick that will allow our storage systems to achieve speedup and/or scaleout is dividing the data in such a way that each data item belongs in a single shard (sometimes called a partition, but we will try to avoid that overloaded term) or replica group. That way, as we add more shards to the system we increase its capacity, allowing it to store more data and perform more independent, parallel work.

Note that sharding is completely orthogonal to replication; we can add shards to the system and we can add replicas to a shard; the latter increases our fault tolerance and the former (in general) increases our performance by increasing our capacity.

In order to implement a sharded KVS you will need to choose a strategy for deciding which shard each data item belong on. A good sharding policy will distribute the data more or less uniformly among the nodes (can you see why?) As shards are added and removed from the system via view change operations, the data items will need to be redistributed in order to maintain a uniform distribution.

Needless to say, your scaleable KVS must also uphold all of the guarantees of HW1, 2 and 3.

With this assignment, we will be adding one more environment variable:
S - this will be the number of shards to split the nodes into to begin with

You should start containers with a command in the form of:

```
docker run -p LocalPort:ExposedPort --net=YourNetwork --ip=DockerNetworkIP -e
VIEW="YourComputer'sIP:LocalPort,..." -e IP_PORT="YourComputer'sIP:LocalPort"
-e S="NumberOfShards" imageTag
```

Or, for a more concrete example:

```
docker run -p 8082:8080 --net=mynetwork --ip=192.168.0.2 -e
VIEW="192.168.0.2:8080,192.168.0.3:8080,192.168.0.4:8080,192.168.0.5:8080" -e
IP_PORT="192.168.0.2:8080" -e S=”2” testing
```

We ask that you give each shard a unique id, I will be calling it their shard id through the rest of this spec. (I personally recommend making your ids integers, but there is no particular reason why they must be)

The following are the new endpoints we are requiring for your new sharded kvs:

GET /shard/my_id

- Should return the container’s shard id
  - {“id”:<container’sShardId>},
  - Status = 200

GET /shard/all_ids

- Should return a list of all shard ids in the system as a string of comma separated values.
  - {“result”: “Success”,
  - “shard_ids”: “0,1,2”},
  - Status = 200

GET /shard/members/<shard_id>

- Should return a list of all members in the shard with id <shard_id>. Each member should be represented as an ip-port address. (Again, the same one you pass into VIEW)
  - {“result” : “Success”,
  - “members”: “176.32.164.2:8080,176.32.164.3:8080”},
  - Status = 200
- If the <shard_id> is invalid, please return:
  - {“result”: “Error”,
  - “msg”: “No shard with id <shard_id>”},
  - Status = 404

GET /shard/count/<shard_id>

- Should return the number of key-value pairs that shard is responsible for as an integer
  - {“result”: “Success”,
  - “Count”: <numberOfKeys> },
  - Status = 200

- If the <shard_id> is invalid, please return:
    - {“result”: “Error”,
    - “msg”: “No shard with id <shard_id>”},
    - Status = 404

PUT /shard/changeShardNumber -d=”num=<number>”

- Should initiate a change in the replica groups such that the key-values are redivided across <number> groups and returns a list of all shard ids, as in GET /shard/all_ids
    - {“result”: “Success”,
    - “shard_ids”: “0,1,2”},
    - Status = 200
- If <number> is greater than the number of nodes in the view, please return:
    - {“result”: “Error”,
    - “msg”: “Not enough nodes for <number> shards”},
    - Status = 400
- If there is only 1 node in any partition as a result of redividing into <number> shards, abort the operation and return:
    - {“result”: Error”,
    - “msg”: “Not enough nodes. <number> shards result in a nonfault tolerant shard”},
    - Status = 400
- The only time one should have 1 node in a shard is if there is only one node in the entire system. In this case it should only return an error message if you try to increase the number of shards beyond 1, you should not return the second error message in this case.

The endpoints from the previous assignments are back as well. Notice that there has been a slight change to GET /keyValue-store/<key> and GET /keyValue-store/search/<key>, in that you should also return the shard id of the node who is holding that particular key.

Additional logic will also be needed in the adding and removing of nodes to the view to keep the number of nodes in each shard about the same and to maintain fault tolerance.

For example, imagine you had 5 nodes split into 2 shards like so: [A,B,C],[D,E]. Now imagine I removed E, we would be left with: [A,B,C],[D]. In this case we would want to move one of the other nodes into the second shard, like so: [A,B],[D,C]

Alternatively, if we had 6 nodes split into 3 shards: [A,B],[C,D],[E,F] deleting one node would leave us with: [A,B],[C,D],[E]. In this case the only thing we can do is to remove a shard, like so: [A,B],[C,D,E]

Key Value Store Operations:

GET /keyValue-store/<key> -d “payload=<payload>”

- should return the value of the <key> in question, like so:
    - {"result": “Success”,
    - "value" : <value>,
    - “owner”: <shardId>, //the shard id of the node who is currently responsible for this key-value
    - “payload”: <payload>},
    - Status = 200
- if the key in question does not exist, it should return an error:
    - {"result": “Error”,
    - "msg" : “Key does not exist”,
    - “payload”: <payload> },
    - Status = 404

PUT /keyValue-store/<key> -d “val=<value>&&payload=<payload>”

- should add the key <key> with value <value> to your key-value store. It should return a confirmation which looks like either:
    - if a new key:
        - {"replaced": False,
        - "msg" : “Added successfully”,
        - “payload”: <payload> }
        - Status = 201
    - if an existing key whose value we’ve updated:
        - {"replaced": True,
        - "msg" : “Updated successfully”,
        - “payload”: <payload> },
        - Status = 200

DELETE /keyValue-store/<key> -d “payload=<payload>”

- Should remove the key <key> from your key-value store as if it never existed. A confirmation should be returned:
    - {"result": “Success”,
    - "msg": “Key deleted”,
    - “payload”: <payload>},
    - Status = 200
- If the key is not in your key-value store an error message should be returned:
    - {“result”: “Error”,
    - “msg”: “Key does not exist”,
    - “payload”: <payload>},
    - Status = 404

GET /keyValue-store/search/<key> -d “payload=<payload>”

- Should return either the key <key> is in the key value store, as so:
    - {"isExists": True,
    - "result": “Success”,
    - “owner”: <shardId>, //the shard id of the node who is currently responsible for this key-value
    - “payload”: <payload>},
    - Status = 200
- Or that the key is not in the key value store:
    - {"isExists": False,
    - "result": “Success”,
    - “payload”: <payload>},
    - Status = 200

If for any of the above /keyValue-store endpoints, you are unable to answer the request for causal consistency reasons, please return:

- {"result":"Error",
- "msg": "Unable to serve request and maintain causal consistency",
- "payload": <payload>},
- Status=400

If for any of the above /keyValue-store endpoints, you are unable to answer the request because the entire shard which owns the key in question is unreachable (ie down or network partitioned away) please return:

- {"result":"Error",
- "msg": "Unable to access key: <key>",

- "payload": <payload>},
- Status=400

View Operations:
GET /view

- Should return that container's view of the system. i.e. A comma separated list of all of the ip-ports your containers are running on (the same string you pass to the environment variable VIEW when you run a container). This should look like this:
    - {'view': "176.32.164.2:8080,176.32.164.3:8080"},
    - Status = 200

PUT /view -d "ip_port=<NewIPPort>"

- Should tell container to initiate a view change, such that all containers in the system should add the new container's ip port <NewIPPort> to their views. Should return a confirmation which looks like:
    - {'result': "Success",
    - 'msg' : "Successfully added<NewIPPort> to view"},
    - Status = 200
- If the container is already in the view, this should return an error message like so:
    - {'result': "Error",
    - 'msg': "<NewIPPort> is already in view"},
    - Status = 404

DELETE /view -d "ip_port=<RemovedIPPort>"

- Should tell container to initiate a view change, such that all containers in the system should remove the old container's ip port <RemovedIPPort> from their views. Should return a confirmation which looks like:
    - { 'result': "Success",
    - 'msg' : "Successfully removed <RemovedIPPort> from view"},
    - Status = 200
- If the container is already in the view, this should return an error message like so:
    - {'result': "Error",
    - 'msg': "<RemovedIPPort> is not in current view"},

- Status = 404
-

Turning in your assignment:
0. Preconditions - You have a project folder on your local machine. (So you know, have done the assignment)

1. Create a file members.txt. It will contain one member per line, member being the ucsc id. There is no need to include your full names or other information here. e.g.
members.txt:
palvaro
elaolive

2. Create a contribution-notes.txt file. This can be empty until the moment of final submission.
Contribution-notes.txt, for example:
palvaro
Implemented search service
elaolive
Wrote the test script

3. Create a local git repo (git init, add, commit)

4. Create a team (Skip if using same team, update members.txt) as specified in last assignment

a. Team ID needs to be unique at bitbucket, but prefix it with cmps128 e.g. cmps128team1.

b. Add team members and Cmps128teachingstaff to the repo.

5. Develop as before. COMMIT FREQUENTLY, we will check contributions against commit history, we would like to see your progress. (Its also just good practice)

6. Closer to the dead line we will release a google form for you to put your commit id and repository url into.

A test script to help your development:
hw4_test.py - be aware, this is just a subset of the tests that will be used to grade your assignment
docker_control.py - contains some helpful functionality for building images and starting and killing containers. Must be in the same folder as hw4_test.py for the test script to function.

Edits:

11/29 - the example for starting containers had the wrong letter as the environment variable! It has been fixed to reflect the written description

- added an error message for the /keyValue-store endpoints if an entire shard is unreachable

12/2 - releasing test script

12/4 - removed reference to Blockade from test script