Instructions

In this assignment you will be asked to build upon the functionality of your previous assignment. As before you will be storing key-value pairs, but this time your system must be an **available fault tolerant** KVS that provides **causal** and **eventual** consistency. In other words it must be fault tolerant so that it is able to survive the loss of any machine while remaining available and maintaining causal consistency.

In the previous assignment you achieved some fault tolerance through your proxies. The loss of any one of those did not affect the system. However, if the main server went down for any reason your system was screwed. Now we would like you to achieve fault tolerance through **replication**. Under this scheme, all nodes should have a copy of all the data, unlike before where only the main one held information.

But how will we keep the data consistent across nodes?

As we learned in class, providing strong consistency for a storage system is incompatible with guaranteeing availability in the presence of network partitions. But this is not to say that we cannot provide any consistency guarantees in such a situation! In this project, you will implement an available KVS that provides causal consistency and eventual consistency.

Eventual consistency is a liveness property that states that all replicas with eventually "converge" to storing the same values. Eventual consistency is trivially upheld in strongly-consistent storage systems (in some sense, these are "immediately" consistent). But it is a necessary property even for weakly consistent systems (if we want them to be useful).

Replicas may temporarily disagree about the values of variables, and must work together to resolve conflicts. The conflict resolution logic we want to to follow is as follows:

Given two conflicting versions V1 (eg, X=1) and V2 (eg, X=2) of a given variable on different replicas, both replicas should eventually converge to:

1. V1, if V1's causal history dominates that of V2.
2. V2, if the reverse is true.
3. Whichever of V1 or V2 has the later timestamp, otherwise.

As is common practice, the timestamp should be selected by the *first* replica to process the operation.

You probably already have an intuition for what causal consistency is based on our discussions of happens-before, causal broadcast, and consistent global snapshots. This is the first time, however, that we are applying the idea to a storage system supporting operations such as reads and writes.

Causal consistency is a safety property that characterizes the allowable values that reads can return. Simply stated (LOL), it says that if a client reads a particular version vB of data item B, and there is another value vA such that vA -> vB (vA happened-before vB), the client may only read versions of A at least as recent as vA. But we haven't said what it means for a version of a data item to happen before another! The intuition here is straightforward: a client's reads happen before their writes. If I read "X=bob smells" (call this vA) from the datastore, and then write "Y=F you alice!" (call it vB) back, then vA -> vB. If carol reads my comeback and then writes "Z=he told you alice!" then we may say that vB -> vC and vA -> vC as well.

What could go wrong? Well, shoot. If our system is replicated in a weakly-consistent way, we cannot be sure that all replicas are up to date. What happens when carol reads vB from a recent replica, but then reads A from an old one? Causal consistency says that replica MAY NOT return an older value for X than vA. This means that replica may need to block.

Wait, how are we gonna make this work, you are probably thinking. It turns out that to implement causal consistency, the client needs to participate in the propagation of what you might call the "causal context" of its operations. In particular, when a client does a write it needs to indicate *which of its prior reads* may have caused the write!

You will see in the spec that we ask you to return a causal payload with your message responses. This might be a lamport or vector clock, a log of transactions, or something else entirely (we might recommend vector clocks, though). Your system should use it to determine what version of a key's value X you should return. We are not interested in how you choose to use your causal payload, but we will be sure to include it in all our requests to your key value store after getting it the first time.

To do this you will need to set a couple different environment variables:

- VIEW - this will be a comma separated list of ip-ports that your containers will use to talk to each other (ex: "176.32.164.10:8082,176.32.164.10:8083")
- IP_PORT - this will be the container's ip-port. (ex: "176.32.164.10:8082")

In short, you should run your docker containers with the following command:

```
docker run -p <LocalPort>:<ExposedPort>  -e
VIEW="<YourComputer's/DockerMachine'sIP>:<LocalPort>,..." -e
IP_PORT="<YourComputer's/DockerMachine'sIP>:<LocalPort>" <imageTag>
```

For example, on my system, starting two containers, I run:

```
docker run -p 8082:8080 -e VIEW="176.32.164.10:8082,176.32.164.10:8083" -e
IP_PORT="176.32.164.10:8082" testing
```

```
docker run -p 8083:8080 -e VIEW="176.32.164.10:8082,176.32.164.10:8083" -e
IP_PORT="176.32.164.10:8083" testing
```

Your system must respond to the following endpoints:

- GET /keyValue-store/<key> -d “payload=<payload>”
    - should return the value of the <key> in question, like so:
        - {"result": “Success”,
        - "value" : <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 = 200
    - if an existing key whose value we’ve updated:
        - {"replaced": True,
        - "msg" : “Updated successfully”,
        - “payload”: <payload> },
        - Status = 201
- 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”,
        - “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, you are unable to answer the request because the payload is too old, please return:
    - {"result":"Error",
    - "msg": "Payload out of date",
    - "payload": <anUptoDatePayload>},
    - Status=400
- 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': "176.32.164.10:8082,176.32.164.10:8083"},
        - 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.

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

To evaluate a homework submission, I am going to create a number of docker containers using your Dockerfile in your project directory. I will then run the test script which may (you should probably go ahead and read that “may” as “will”) have more tests than I release to you all.

Use these to help your development, but certainly do not stop testing with these:
hw3_test.py - a nonexhaustive tests to get you started on developing
docker_control.py - a set of functions which make interacting with docker easier from within python.
Make sure you have docker_control.py in the same folder as hw3_test.py for it to run as expected.

Edit:

11/10 - the test script had an error in the expected value of GET view. It was looking for a list of ipPorts, it now (correctly) looks for a comma separated string of ipPorts

11/15 - fixed a typo in DELETE /view. Added an error message for out of date payloads for /keyValue-store

11/17 - altered the test script and docker_control.py to use subnets when starting containers, it should (*fingers crossed*) work on Mac and Linux now. When comparing the view, it does it on a per ipPort basis, rather than comparing the entire string (so ordering them in a particular order in no longer necessary). Note: The command line interface for docker_control.py has changed slightly, so look that over if you were using that.

11/19 - cleaned up the last several tests to not try to delete the addresses the host would use for communication, but rather an address they would use to communicate between each other. docker_control.py has been updated to use subprocess.getoutput with strings rather than lists.