Assignment 3

Version 0.10
Fault Tolerant Key-Value Store

Due Date: Friday December 15th, 2017 @ 11:59 pm, tentative

Objective

The purpose of assignment 3 is to extend/modify your TCP client/server key-value store from assignment #1 and assignment #2 to provide fault tolerance via the Raft consensus protocol as detailed in the Raft paper. Using Raft each node will have its own "log" for tracking data changing events, in this case, put and del commands. Raft's function is to replicate logs across nodes, and when there is a majority consensus, initiate replication of data. Using consensus it should be possible to tolerate failures of several nodes as long as a majority remains to sustain consensus.

A very useful summary of the Raft consensus protocol appears on page 308 of the paper:

http://faculty.washington.edu/wlloyd/courses/tcss558/papers/InSearchOfAnUnderstandableConsensusAlgorithm.pdf

Raft Consensus Protocol

In Raft every node is in either one of three states: ***follower***, ***candidate***, or ***leader***. The Raft consensus protocol works by always having a "leader" node which coordinates all interaction among nodes. In Raft there are "rounds", known as terms, where there is just one "leader" for a term. The term lasts until the leader node fails. When the leader node fails, follower nodes detect the failure, and one of the followers "expires first" from a random sleep delay, and then begins a new election as the "candidate" node to be leader. The leader must be voted by the majority of followers of the system to take over as leader. Followers will not vote for the candidate if the candidates log is not at least as up-to-date as the followers. This way, elections ***filter-out*** candidates which lack information to be leader. The assumption for the over-all system is that there will always be a majority of nodes that have not failed such that one will eventually be able to take over leadership in the event of the failure of the leader node.

Implementing Raft only requires two new methods to be implemented: AppendEntries and RequestVote. These methods however, require transferring substantial data from the leader to the follower nodes. If desired you may consider using an RMI based key-value store, as it may be easier to implement this data passing, though a TCP-based solution is possible, but the data must be manually "marshalled" over the socket for these new methods.

Additionally, the "put" and "del" methods from any candidate should always be forwarded to the leader node. The leader will then add these requested changes to its log. At the leader, changes will be committed from the log to the leader's key-value store when a majority of nodes report having replicated the log entries. Log entries are committed to the follower's key-value stores when during regular heartbeats from the leader to the followers, the leader reports having committed entries to its key-value store. In other words, the leader commits first based on succesfully learning that log entries have been replicated. Once the leader commits this "opens the door" for follower nodes to then subsequently commit.

For implementing Raft, it is recommended to thoroughly and carefully read sections 1, 2, and 5 of the paper, while making notes to understand how the protocol works.

Using program code developed for assignment 1 & 2, modify "GenericNode" to support additional "internal" commands for Raft as described in the table to support fault tolerance. For testing, on a 3-node system it should be possible to periodicaly stop and start one-node without any loss of data. For a 5-node system, it should be possible to periodically stop and start two nodes on the system without data loss. When nodes are resurrected they will begin with NO DATA, and the raft protocol will bring them up to date.

For Raft, you will need to have a Node membership directory so that you can know the set of nodes to work with. For Assignment #3 it is ok to assume that the set of nodes remains fixed. For example, once the system is started with 3, or 5 nodes, these same nodes (with their IPs and Ports) remain fixed. When nodes fail and come back up, they will be on the same IPs and ports. To implement this in Docker it is simply a matter of running a container, stopping, and restarting it. Docker will always populate container IPs in order:

172.17.0.2	Start your membership server first (KV store)
172.17.0.3	Raft node 1
172.17.0.4	Raft node 2
172.17.0.5	Raft node 3
172.17.0.6	Raft node 4
172.17.0.7	Raft node 5

Raft Protocol Key-Value Store

Operation	Description
requestvote	requestvote <see raft paper for parameters> Candidate node will communicate with each node to acquire a vote to determine leadership status. When the candidate is unable to acquire a majority number of votes before the randomly selected electionTimeout elapses, the election expires, the term number incremented, and a new election is started.
appendentries	The leader node periodically sends an "appendentries" command to every node (except itself) as a heartbeat. This occurs at a regular interval of approx. ~200ms for example. When there are log updates, log updates are included in these heartbeat calls. The leader maintains a list of log indexes for each follower node which indicates where it believe the followers log is with respect to

	updates. The follower can reject an update request with a "FALSE" response. When "FALSE" is received the leader walks back by one its index value for the followers log and sends incrementally more data each time until the follower accepts the request with "TRUE". When the follower accepts the request it adds its data to the log. The leader should use a set of threads, one for each follower node to continuously enact the heartbeat. You may use a thread pool for simplicity. (see Java ExecutorService class)
put / del	put and del These commands from the previous assignments will need to be modified. When a node receives a put or del, if it is not the leader, then it should forward the request directly to the leader node. The leader node is the only node which will initially process these requests. The leader will append put and del commands to its log and then push out updates the followers.
dput1, dput2, ddel1, ddel2	All of these commands from two-phase commit should be commented out, disabled, or deleted for assignment #3.

It is assumed that the get, store, and exit commands work as in Assignment #1.

To support Raft it is necessary to track node membership in the distributed system to determine the set of nodes across which data will be replicated. To simplify implementation, you can assume that the configuration of the system will never change once brought online. So if there is a 5-node system (node1, node2, node3, node4, node5), even if two nodes fail (e.g. node2, node5), your system will continue to try to communicate with all 5 nodes forever, with the hope that the nodes will eventually come back on line.

You may use any of the membership tracking methods described in assignment #2 including: (1) stack configuration file, (2) UDP Discovery, and Centralized membership key/value store.

The preferred implementation for assignment #3 is in Java 8. Students are free to implement assignment #2 in C, C++, or Python if preferred. **Solutions in alternate languages must include documentation to describe how to operate the client and server in the alternate language.** All operations including setup must be explained. Components must be deployed using server and client docker containers.

Docker for Assignment #3

All solutions must include a **(#1)** a Server Dockerfile for your ***membership server*** (UDP or TCP key-value store), and **(#2)** a Server Dockerfile for your raft nodes. Docker files should be in a separate folder with all supporting files. It should be possible to quickly perform a "sudo docker build" and "sudo docker run" for each container. If using a TCP key-value store, then it is expected that the runserver.sh script will need to be updated with the fixed IP address of the centralized key-value store before running raft containers. (e.g. the IP is hard coded as a startup parameter for the GenericNode so it knows where to find the membership server).

Sample dockerfiles can be downloaded here:

http://faculty.washington.edu/wlloyd/courses/tcss558/assignments/a1/a1_dockerfiles.tar.gz

To extract a tar gzip file use the command: (x for extract, z for unzip, f for file)

tar xzf a1_dockerfiles.tar.gz

Then cd into the individual docker_server or docker_client directories to build the docker images.

The sample dockerfiles includes a placeholder GenericNode.jar Java class archive file. For assignment #2, you're to focus on extending the GenericNode.jar TCP server.

Inside the docker_server directory, a runserver.sh script has been provided.
This script includes a command to start a server of one of the given types.

When building your docker_server container, you should uncomment the TCP server by removing the "#":

```
# Dummy jar file
#java -jar GenericNode.jar

#TCP Server – centralized node directory if used
#java -jar GenericNode.jar ts 4410

#TCP Server – KV store
java -jar GenericNode.jar ts 1234 <IP of node directory if used>
```

Once running, to discover the internal IP address of your server running on a Docker host, use the following sequence:

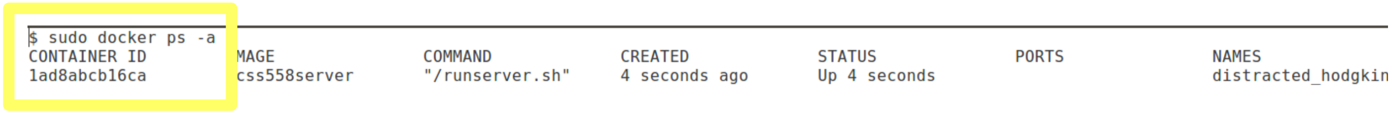
First, build the docker_server container:

```
$ cd docker_server
$ sudo docker build -t tcss558server .
Sending build context to Docker daemon   5.12kB
Step 1/7 : FROM ubuntu
 ---> ccc7a11d65b1
Step 2/7 : RUN apt-get update
 ---> Using cache
 ---> 1413c1a1f91b
Step 3/7 : RUN apt-get install -y default-jre
 ---> Using cache
 ---> b23e154d7af3
Step 4/7 : RUN apt-get install -y net-tools
 ---> Using cache
 ---> 1d81d5652fc2
Step 5/7 : COPY GenericNode.jar /
 ---> Using cache
 ---> f74d73c86c5c
Step 6/7 : COPY runserver.sh /
 ---> Using cache
 ---> f23167bd7d09
Step 7/7 : ENTRYPOINT /runserver.sh
 ---> Using cache
 ---> e921fbb5db7a
Successfully built e921fbb5db7a
Successfully tagged tcss558server:latest
```

Then, run the docker container:

```
$ sudo docker run -d --rm tcss558server
1ad8abcb16cae530322464099487d028154a2452072e5e20f6007ff3e5f1a66d
```

Now, grab (copy and paste) your unique CONTAINER ID. The Name can also be used (here distracted hodgkin):



```
$ sudo docker ps -a
CONTAINER ID        MAGE                COMMAND             CREATED             STATUS              PORTS               NAMES
1ad8abcb16ca        css558server        "/runserver.sh"     4 seconds ago       Up 4 seconds                            distracted_hodgkin
```

Next, execute bash interactively on this container

```
$ sudo docker exec -it 1ad8abcb16ca bash
```

Then, use the "ifconfig" command inside the container to query the local IP address:

```
root@1ad8abcb16ca:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:48 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6527 (6.5 KB)  TX bytes:0 (0.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Testing:

For testing your raft implementation, the focus will be on using a single Docker host where all raft servers run inside separate containers on a single server. While it should be simply just "details" to deploy across multiple Docker hosts and scale out the system at a much larger scale than 5-nodes this will not be tested extensively, if at all for evaluation of assignment #3.

Assumptions and suggestions to simplify the assignment:

You may make the following assumptions and simplifications of your RAFT implementation to ease development:

1. When a candidate performs RequestVote() the voting process does not have to be multi-threaded. It is OK to simply loop through all of the nodes to obtain a vote. You should, however, retry each node up to a FIXED number of times (e.g. 10) before moving on.

2. Your TCP client should catch exceptions so that when a node has failed your server doesn't crash. Catching exceptions allows you to retry, and eventually if the node is restored, resume normal operation. Catch IOException for new Socket(), getOutputStream(), and getInputStream().

3. To implement some of the random timeouts required by Raft, try setting .setSoTimeout() of your TCP socket.

4. All Node IP/Port pairs are unique. There will be no duplicates.

5. Your NodeDirectory should track a node ID. It is helpful if it has methods to retrieve a node by ID, or a nodeID by IP/Port.

6. For leader election, it is ok to assume that the node should reply after ~10 retries, and then skip and move on. When skipping, no vote should be recorded.

7. It is ok to push out data during heartbeats, instead of calling AppendEntries when the Leader node receives a put request. AppendEntries can be called using a fixed interval such as every ~200ms. For optimal performance, when "put" is called on the leader, AppendEntries would be called for all nodes, but this is not required.

Your TCP servers should support the same syntax from Assignment #1, except now multiple servers are run in parallel coordinating using the Raft protocol.

#TCP Server (ts for TCP server – for #3 centralized membership key-value store:
```
java -jar GenericNode.jar ts <server port number> <membership-server-IP> <membership-server-port>
```

#Example:
```
java -jar GenericNode.jar ts 1234 54.12.44.33 1111
```

For Assignment #1, you may optionally have your servers output debugging information. But there are no formal output requirements for servers to generate output either to the console or to logfile(s).

Testing the servers

Operation of the servers will be as in Assignment 1 & 2. Get, put, delete, store, and exit commands should be supported as in assignment #1:

TCP CLIENT TO SERVER INTERACTION

The first parameter is "tc" for TCP client.
The second parameter is the server IP address.
The third parameter is the server port.

Replace localhost with your server IP address.
"1234" represents the service port. The client and server allow the port number to be specified. Replace with the port used.

```
$ java -jar GenericNode.jar tc localhost 1234 put a 123
server response:put key=a

$ java -jar GenericNode.jar tc localhost 1234 put b 456
server response:put key=b

$ java -jar GenericNode.jar tc localhost 1234 get a
server response:get key=a get val=123

$ java -jar GenericNode.jar tc localhost 1234 del a
server response:delete key=a

$ java -jar GenericNode.jar tc localhost 1234 store
server response:
key:b:value:456:

$ java -jar GenericNode.jar tc localhost 1234 exit
<the server then exits>
```

TCP / UDP References

These may be helpful:
https://systembash.com/a-simple-java-tcp-server-and-tcp-client/
https://docs.oracle.com/javase/tutorial/networking/sockets/index.html
https://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html

Two Phase Commit Protocol References

The internet will contain helpful references. We will discuss also in class. See book chapter 8.5 as well.

Suggested Integrated Development Environment (IDE) / Project Build Files

For maximum potential for ***partial credit***, students may submit their project as a Netbeans project created with the Netbeans IDE, Oracle's Java IDE. By providing projects as a Netbeans project, it will be possible for the grader to build your source

and fix potential issues to support partial credit. For example, a student may nearly have the code correct, but a small detail prevents operation. If the grader can rapidly fix the code, a lot of partial credit may be awarded.
Download Netbeans 8.2 here:
https://netbeans.org/downloads/

Alternatively, students not using Netbeans may submit all requisite project build files, as well as descriptive documentation which clearly states how code can be rebuilt. If the grader is easily able to rebuild your projects, then there is higher potential for partial credit.

If no build files are provided, and/or no subsequent documentation describing how to build your projects, **it will not be possible to issue partial credit** for functionality that is nearly complete with minor bugs. In this case, only by reading the code will the grader attempt to issue partial credit if portions of the program do not work correctly.

Testing Function and Performance

Your RAFT implementation will be tested by validating that the system retains the same key-value store log when nodes are terminated and restored. For a 5-node deployment, at most two nodes may be terminated. For a 3-node deployment, at most one node will be terminated.

The test script can be used to provide initial data:

TCP:
http://faculty.washington.edu/wlloyd/courses/tcss558/assignments/a1/bigtest_tc.sh
To run the scripts, adjust the server and port BASH variables as needed to test your deployments.

You can check that ***each*** server node has replicated the same contents of the script by counting the number of resulting lines in the key value store at the conclusion of the test script as follows:

```
$java -jar GenericNode.jar tc localhost 1234 store | wc -l
```

Assuming no blank lines, the count should be 70. Also textual diffs can be performed of the outputs.

What to Submit
To submit the assignment, teams should build a single tar gzip archive file that contains **all** project source code in a main project directory. This could be the Netbeans project folder. In the project directory there should be two directories for the Docker containers: membership_server and raft_server. The folders must be updated to include your GenericNode.jar file and they must support building a functioning server container. If you don't use a membership server, but instead require a fixed file to specify raft nodes, only the raft_server container is required.

Grading Rubric

This assignment will be scored out of 100 points, while as many as 110 points are possible.

Functionality	90 points
20 points	3-node system supports loss and restoration of one node
10 points	5-node system supports loss and restoration of two nodes
15 points	Logs update kv stores correctly (e.g. replication & synchronization)
10 points	Raft follower node role: votes and updates logs
10 points	Raft candidate election mode: RequestVote requests
10 points	Raft leader node role: sends heartbeats
10 points	Raft leader node role: sends AppendEntries Updates
5 points	Node membership server working

Miscellaneous	20 points
5 points	Use of multiple server threads
5 points	Healthy synchronization, concurrency, locking, etc.
5 points	Docker container scripts
5 points	Program compiles, instructions provided or Netbeans used

Teams (optional)

O*ptionally*, this programming assignment can be completed with two or three person teams.

If choosing to work in a team, ***only one*** person should submit the team's tar gzip project source archive file and the performance report PDF file to Canvas.

Additionally, ***EACH*** member of a team should submit an **effort report** on team participation. **Effort reports** are submitted INDEPENDENTLY and in confidence (i.e. not shared) by each team member.

Effort reports are not used to directly numerically weight assignment grades.

Effort reports should be submitted as a PDF file named: "effort_report.pdf". Google Docs and recent versions of MS Word provide the ability to save or export a document in PDF format.
For assignment 0, the effort report should consist of a one-third to one-half page narrative description describing how the team members worked together to complete the assignment. The description should include the following:

1. Describe the key contributions made by each team member.
2. Describe how working together was beneficial for completing the assignment. This may include how the learning objectives of using EC2, Docker, Docker-machine, and haproxy were supported by the team effort.
3. Comment on disadvantages and/or challenges for working together on the assignment. This could be anything from group dynamics, to commute challenges, to faulty technology.

4. At the bottom of the write-up provide an effort ranking from 0 to 100 for each team member. Distribute a total of 100 points among both team members. Identify team members using first and last name. For example:

   John Doe
   Research 65
   Design 42
   Coding 30
   Testing 80

   Jane Smith
   Research 35
   Design 58
   Coding 70
   Testing 20

Team members may not share their **effort reports**, but should submit them independently in Canvas as a PDF file. Failure of one or both members to submit the **effort report** will result in both members receiving NO GRADE on the assignment…

Disclaimer regarding pair programming:
The purpose of TCSS 558 is for everyone to gain experience developing and working with distributed systems and requisite compute infrastructure. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer simply watching the champion! The tasks and challenges should be shared as equally as possible.

Helpful Hints

```
# Docker build
sudo docker build -t tcss558server .
sudo docker build -t tcss558client .

# Run docker container in the background
sudo docker run -d --rm tcss558server
sudo docker run -d --rm tcss558client

# Docker shell to a container
sudo docker exec -it <container-id> bash
```

To display all containers running on a given docker node:
```
docker ps –a
```

To stop a container:
```
docker stop <container-id>
```
For example:
```
docker stop cd5a89bb7a98
```

Also **`docker kill`** will kill a running container and **`docker rm`** will remove a container which has exited but is no longer running.

Document History:

v.10   Initial version