TCSS 558: Applied Distributed Computing — Institute of Technology
Fall 2017 — University of Washington – Tacoma
http://faculty.washington.edu/wlloyd/courses/tcss558
Instructor: Wes Lloyd

Assignment 2

Version 0.14

Replicated Key Value Store

Due Date: Friday December 1st, 2017 @ 11:59 pm, tentative

Objective

The purpose of assignment 2 is to extend your TCP client/server key-value store to support replication and node discovery. Replication can be implemented using a two-phase-commit algorithm.

Two-Phase Commit Algorithm

Any TCP server receiving a client request to put or delete a node becomes the leader of the transaction. Every TCP server maintains an active copy of the node membership directory. Upon receiving a put or delete request the TCP server node sends becomes the leader and sends a round of dput1/ddel1 requests to every TCP server to request the put or delete of the key/value pair. Each node checks if the key in question is presently locked. If the key is ***available***, the TCP server responds with an acknowledgement to the leader indicating that it is able to proceed with the transaction. At this this the node will "lock" the key from other updates. If the leader receives acknowledgements from ***ALL*** known servers, then a second dput2/ddel2 command is sent to all nodes to commit the data. If nodes receive the dput2/ddel2, they proceed with the operation locally and unlock the key. If any one node responds to the leader with an abort message (because the key is locked) **the entire transaction is aborted.** At this point the leader will retry the transaction up to a fixed number of times, let's say 10. If the transaction can not be committed after 10 attempts, it is aborted, and an error message is produced.

Using program code developed for assignment 1, extend "GenericNode" to support additional "internal" commands described in the table to support two-phase commit. Your TCP Server will now leverage your TCP client code for the purpose of sending internal commands to the server nodes. The operations below represent an application-level protocol for a replicated key-value store. **You are welcome to implement a custom protocol (and replication algorithm) if desired, but you will be required to submit *summary* documentation (as a PDF file) describing your protocol in a similar format as in the table.**

You may assume:

- Nodes will NOT fail during transactions.
- Only write transactions (put / delete) need to be synchronized.
- Each node should maintain data for in-progress transactions to a concurrent data structure. You do not have to implement your own from scratch.
- Only one operation on a given key can be performed across the nodes concurrently. Attempts to perform multiple parallel operations on the same key will be aborted by any node detecting conflicts.

Replicated Key-Value Store Application Protocol

Operation	Description
dput1	dput1 <key> <value> Called by the transaction leader, dput1 starts an internal node-to-node transaction to replicate the key/value pair at a remote node. Upon receiving this command the local node locks the key/value pair and sends an acknowledgement message to the leader indicating it is ready to proceed to commit the PUT operation. If the key/value pair is already locked locally, then an abort message is sent to the leader and the transaction will be aborted.
dput2	dput2 <key> <value> Called by the transaction leader, dput2 finishes an internal node-to-node transaction to put the key/value pair at a remote node. Upon receiving this command the local node performs the "put" operation on its local datastore. After the key is put, the key/value pair is unlocked enabling other updates to be performed by other nodes in the replicated key/value store. The node responding to dput2 will respond with a message equivalent to the put message from assignment 1.
dputabort	dputabort <key> <value> Called by the transaction leader, dputabort aborts an internal node-to-node transaction across the replicated key-value store. Transactions are aborted by the leader if at least one node sent an abort message as a result of dput1. Upon receiving the dputabort, the local node removes the lock from the local key/value store.
ddel1	ddel1 <key> Called by the transaction leader, ddel1 begins a replicated delete operation across the nodes of the replicated key/value store. Locking and messaging follow the same scheme as for dput1.
ddel2	ddel2 <key> Called by the transaction leader, ddel2 finishes a replicated delete operation across the nodes of the replicated key/value store. Locking and messaging follow the same scheme as for dput2. The node responding to ddel2 will respond with a message equivalent to the del message from assignment 1.
ddelabort	ddelabort <key> <value> Called by the transaction leader, ddelabort aborts an internal node-to-node transaction across the replicated key-value store. Transactions are aborted by the leader if at least one node sent an abort message as a result of ddel1. Upon receiving ddelabort the local node removes the lock from the local key/value store.

To support a replicated key-value store it is necessary to track node membership in the distributed system to determine the set of nodes across which data will be replicated.

Three membership tracking methods are available. Students must implement at least one membership tracking method. Extra credit is available for implementing method #2 and/or method #3.

Node Membership Tracking Methods

Method	Description
1. Static configuration file	The simplest approach to manage membership is to provide a text file with IP:PORT pairs for all nodes participating in the replicated key-value store. When the GenericNode is deployed using separate docker containers, a static configuration file must be updated and pushed out to each of the containers. The file should be called "/tmp/nodes.cfg". GenericNode should periodically reload this file every few seconds to constantly refresh the view of the system. The format of the file should be a simple list with each node (IP/PORT) represented on a separate line as follows: 10.0.0.5:1234 10.0.0.4:1234 10.0.0.3:1234 As a proof of successful distributed deployment, capture and provide execution times for the bigtest_tc.sh script on a 3-container deployment.
2. UDP Discovery	An alternate approach is to manage membership dynamically by having the server discover nodes via a UDP broadcast protocol. For this approach, devise a ***simple*** protocol where nodes periodically send broadcast messages to each other via UDP, and each node accumulates a complete membership list based on received broadcast messages. If a node is not heard from after 10 seconds, it can be removed from the system. Broadcast should occur using a fixed port, such as #4410. Using UDP discovery, it should be possible to launch multiple docker containers on a single docker host and they will self-discover their existence and begin to collectively replicate all transactional data. Note, due to limitations with Docker overlay networks, it is not presently possible to use UDP discovery to discover nodes on separate Docker host machines. As a proof of successful deployment, capture and provide execution times for the bigtest_tc.sh script on single docker host deployments of 1, 3, and 5 containers.

	See Docker UDP Broadcast Overlay Bug: https://github.com/docker/libnetwork/issues/552
3. Centralized membership key/value store	For approach 3, deploy a centralized, non-replicated instance of your own TCP key value server from assignment #1, to track node membership. Keys will be IP addresses. Values will be ports. The full store list will represent all participating nodes. Add command line arguments to your TCP server startup to specify a centralized key/value store for tracking node membership: New TCP server startup CLI: java -jar GenericNode.jar ts <listen-port> <membership-server-IP> You may use a HARD CODED port for the membership server, for example port 4410. You will need to first launch an instance of your TCP key-value store to listen on this port. You should create a separate docker container for this called "nodedirectory". Once launching the node directory kvstore, check what it's IP address is, and then launch all subsequent keyvalue stores by refering to the membership-server IP address. Detailed instructions are here: http://faculty.washington.edu/wlloyd/courses/tcss558/assignments/a2/DockerSwarmOverlay-howto.txt When your TCP server starts, it will send a TCP client command to publish its own IP/PORT to the centralized server, and then periodically request the complete list of servers every few seconds. As a docker exercise, scripts will be provided to deploy your replicated key-value store as a service across multiple docker hosts. Your centralized membership server will be deployed as a secondary service to a single docker host. The membership server will be deployed first and its IP/PORT can then be statically defined in the runserver.sh script of the docker container for your replicated key-value store. This implementation will enable your key-value store to scale across multiple virtual machines and containers on AWS. As a proof of successful deployment, capture and provide execution times for the bigtest_tc.sh script docker service deployments using 5, and 10 containers deployed across a Docker swarm consisting of 3 and 5 docker host machines.

Students who submit a working, replicated TCP GenericNode server, and performance comparison supporting both UDP (#2), and the Centralized TCP

Membership tracking server (#3) without significant errors will be eligible for **up to 20% extra credit**.

The preferred implementation for assignment #2 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 #2

All solutions must include a Server Dockerfile to support creating server containers. Servers must be able to communicate on the local subnetwork shared among containers of a single Docker host, or shared among multiple containers across a Docker overlay network. The client container will use the docker container private network IP address to facilitate communication with the server.

To support working with Docker containers, Dockerfiles for the client and server have been provided and can be downloaded here: (*feel free to use these, or develop new Dockerfiles...*)

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        IMAGE               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)

It is recommended to develop your code on a local machine, and test the deployments to Docker containers before submitting.

Docker Swarm:

A how-to document is posted here:

http://faculty.washington.edu/wlloyd/courses/tcss558/assignments/a2/DockerSwarmOverlay-howto.txt

The document includes instructions for configuring Docker Swarm using multiple docker-machine hosts and deploying your TCP server as a docker service. Once setting up Docker Swarm an overlay network can be configured to support TCP communication on a private network, or, servers can communicate using AWS public IP addresses.

Running multiple TCP node servers:

Your TCP servers should support the same syntax from Assignment #1, except now multiple can be run in parallel.

#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

Once the IP address of the server is discovered, point your client to this IP address and include the port number for TCP to support client/server interaction.

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

A TCP test script is online at:

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.

To measure performance of TCP, run the script as follows:

#TCP
```
time ../bigtest_tc.sh > /dev/null
```

With your submission, please create a PDF file created with Google Docs.
Include in the file performance numbers obtained for your multi-container / multi-host deployments coinciding with your membership tracking approach (static file, UDP discovery, centralized TCP key/value store) as described in the table.

Please test a 1-node, 3-node, and 5-node deployment. Please include performance numbers in a PDF file. Please label each configuration with the node directory type (Static File, UDP Discovery, or TCP Discover), and the number of server nodes tested so the performance of each is quick to identify and the configure you used is easy to see as follows:

```
# Assignment 2, Static File, 1 Container TCP
TCP   17.513s

# Assignment 2, Static File, 3 Container TCP
TCP   22.143s
```

```
# Assignment 2, Static File, 5 Container TCP
TCP    94.233s
```

Note the example times here are bogus.

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: docker_server and docker_client. The folders must be updated to include your GenericNode.jar file and they must support building a functioning TCP server container.

PDF files with performance results should be submitted as a separate file in Canvas.

Grading Rubric

This assignment will be scored out of 100 points, while as many as 120 points are possible.

Functionality	70 points
10 points	TCP client/server 2-node put replication
3 points	TCP client/server 3-node put replication
2 points	TCP client/server 5-node put replication
10 points	TCP client/server 2-node del replication
3 point	TCP client/server 3-node del replication
2 points	TCP client/server 5-node del replication
20 points	Implementation of **one** membership tracking system: Can be: (1) static node membership file (e.g. nodelist) (2) TCP centralized membership server (3) UDP centralized membership server Instructions should be provided on how to make node discovery work if not obvious.
5 points	Membership tracking system is **dynamic** – nodes can join and leave the system, and configuration is updated. But there is not requirement to bring KV store up to date as in the RAFT protocol. Must indicate in documentation that this feature has been implemented.
10 points	Implementation of a second membership tracking system Can be: (1) static node membership file (e.g. nodelist) (2) TCP centralized membership server (3) UDP centralized membership server Instructions must be provided on how to make the second method of node discovery work if not obvious.
5 points	Implementation of UDP broadcast membership server For UDP broadcast, there should be no configuration (e.g.

IP) Node discovery is accomplished entirely with UDP broadcast messages. No IPs are provided, and configuration is automatic on a single docker host. Must indicate in documentation that this feature has been implemented.

Miscellaneous	50 points
10 points	Use of multiple server threads
10 points	Healthy synchronization, concurrency, locking, etc.
10 points	Performance measurement of multi-node deployments
5 points	Docker container scripts
5 points	Replication protocol style
5 points	Membership protocol style
5 points	Program compiles, instructions provided or Netbeans used

Teams (optional)

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

If choosing to work in pairs, ***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
v.11 Revised to add link to docker swarm notes for configuring network overlay
v.12 Corrected and revised rubric
v.14 Removed legacy comment under rubric related to implemented static member file approach. One approach is required worth 20 pts, a second is 10 pts extra credit. A third approach is 0 pts extra credit.