## Part 1:

The following descriptions perform on Docker container.
Build a TCP socket-based clients and servers in Java. Let's call it "Node". Node will receive command arguments and then take on the role of a client or server for a basic Key-Value store. The following operations should be supported:

| Operation | Description |
|---|---|
| put | put <key> <value><br><br>Put a value to the key-value store. The key uniquely identifies the object to store. A unique key maps to only one object which has a unique value. If multiple clients write to the same key, writes should be synchronized. |
| get | get <key><br><br>Returns the value stored at <key>. |
| del | del <key><br><br>Deletes the value stored at <key> from the key value store. |
| store | store<br><br>Prints the contents of the entire key-value store. You may optionally truncate the output after returning 65,000 characters. If the contents of the key value store exceed 65,000 bytes, the return should start with "TRIMMED:" followed by the content. |
| exit | Exit<br><br>When the exit command is sent by the client, the server is shutdown. |

<u>Servers should support the following syntax:</u>
#TCP Server (ts for TCP server)
# Dummy jar file
**java -jar Node.jar**

# TCP
**java -jar Node.jar ts <server port number>**

<u>Servers should support the following interactions:</u>
#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 Node.jar tc localhost 1234 put a 123
server response:put key=a

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

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

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

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

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

**Part 2:**
Extend your TCP client/server key-value store to support replication and node discovery. Replication can be implemented using a two-phase-commit algorithm.

About 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.

The program should support following operations:

Replicated Key-Value Store Application Protocol

| Operation: | Description |
|---|---|
| dput1 | dput1 <key> <value><br><br>Called by the transaction leader, dput1 starts an internal node-tonode 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><br><br>Called by the transaction leader, dput2 finishes an internal node-tonode 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.<br><br>The node responding to dput2 will respond with equivalent to the put message from part1. |
| dputabort | dputabort <key> <value><br><br>Called by the transaction leader, dputabort aborts an internal nodeto-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><br><br>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><br><br>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.<br><br>The node responding to ddel2 will respond with a message equivalent to the del message from part1. |
| ddelabort | ddelabort <key> <value><br><br>Called by the transaction leader, ddelabort aborts an internal nodeto-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. |

and it should support the following membership tracking methods:

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 Node 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". Node 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:<br>10.0.0.5:1234<br>10.0.0.4:1234<br>10.0.0.3:1234 |
| 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.<br><br>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.<br><br>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. |
| 3. Centralized key/value store membership | For approach 3, deploy a centralized, non-replicated instance of your own TCP key value server from Part #1, to track node membership. Keys will be IP addresses. Values will be ports. The full store list will represent all participating nodes.<br><br>Add command line arguments to your TCP server startup to specify a centralized key/value store for tracking node membership:<br><br>New TCP server startup CLI:<br><br>java -jar Node.jar ts &lt;listen-port&gt; &lt;membershipserver-IP&gt; |

| | |
|---|---|
| | 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.<br><br>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.<br><br>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.<br><br>This implementation will enable your key-value store to scale across multiple virtual machines and containers on AWS. |

Servers should support the following syntax:
# Dummy jar file
**java -jar Node.jar**

#TCP Server – Centralized Node Directory
**java -jar Node.jar ts 4410**

#TCP Server – Key Value Store
**java -jar Node.jar ts 1234 <IP of node directory if used>**

Running multiple TCP node servers:
Your TCP servers should support the same syntax from part 1, except now multiple can be run in parallel.
#TCP Server (ts for TCP server – for #3 centralized membership key-value store:
**java -jar Node.jar ts <server port number> <membership-server-IP> <membership-server-port>**
#Example:
**java -jar Node.jar ts 1234 54.12.44.33 1111**

For part1, 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).

Servers should also support the interaction commands like the part 1 and the part 2 replicated operations above.

Note:
please complete:
1. Use of multiple server threads is required
2. Healthy synchronization is required
3. Proper comments in coding.
4. The program should be able to test with many nodes.

You may assume:
1.Nodes will NOT fail during transactions.
2.Only write transactions (put / delete) need to be synchronized.
3.Each node should maintain data for in-progress transactions to a concurrent data structure. You do not have to implement your own from scratch.
4.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.