Project #3 Single Server Key-Value Store

Hamel Ajay Kothari (de), Maliena Guy (dd), Bryan Cote-Chang (ed), EunSeon Son (et)

Section 106, TA: Kevin Klues

Task 1: Message Parsing

Correctness Constraints

- An error has to be raised in any case of messages cannot be built and processed properly.

Declarations

There are no declarations necessary for this section.

Description

We will define a helper method which will make bulletproofing of message types much easier:

```
boolean validMsgType(String msgType):
    if (msgType is one of getreq, putreq, delreq, or resp)
        return true
    return false
```

The two String based constructors are fairly simple to implement:

```
KVMessage(String msgType) {
    if (validMsgType(msgType))
        this.msgType = msgType;
    } else {
        throw new KVException("resp", "Message format incorrect");
    }
}
KVMessage(String msgType, String message) {
    if (validMsgType(msgType)) {
        this.msgType = msgType;
        this.message = message;
    } else {
        throw new KVException("resp", "Message format incorrect");
    }
}
```

KVMessage() takes in a socket is from the network connection that is then parsed. Using the Document-Builder class from the Java XML libraries, we can parse the input and obtain the msgType.

```
KVMessage(Socket sock) {
    try {
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(new NoCloseInputStream(sock.getInputStream()));
        doc.getDocumentElement().normalize();
        NodeList nList = doc.getElementsByTagName("KVMessage");
        Node nNode = nList.item(0);
        this.msgType = nNode.getAttribute("type");
        if (this.msgType == "getreq" || this.msgType == "delreq") {
            this.key = ((Element)nNode).getElementsByTagName("Key").item(0).getTextContent();
        } else if (this.msgType ==  "putreq") {
            this.key = ((Element)nNode).getElementsByTagName("Key").item(0).getTextContent();
            this.value = ((Element)nNode).getElementsByTagName("Value").item(0).getTextContent();
        } else if (this.msgType == "resp") {
            if (getresponse) {
                this.key = ((Element)nNode).getElementsByTagName("Key").item(0).getTextContent();
                this.value = ((Element)nNode).getElementsByTagName("Value").item(0).getTextContent();
            } else {
                this.message = ((Element)nNode).getElementsByTagName("Message").item(0).getTextContent(
            }
        } else {
            throw new KVException(new KVMessage("resp", "Unknwon Error: msgType is unknown"));
        }
    } catch (SAXException e) {
        throw new KVException(new KVMessage("resp", "XML Error: Received unparseable message"));
    } catch (IOException e) {
        throw new KVException(new KVMessage("resp", "Network Error: Could not receive data"));
    } catch (ParserConfigurationException e) {
        throw new KVException(new KVMessage("resp", "XML Error: Received unparseable message"));
    }
    if (this.key.length() > 256) {
        throw new KVException(new KVMessage("resp", "Oversized key"));
    }
    if (this.value.length() > 262144) {
        throw new KVException(new KVMessage("resp", "Oversized value"));
    }
}
```

toXML() generates XML string for messages. It first checks msgType and tries to output the correct format of XML string. When there are not enough data or invalid data, it throws an exception.

```
toXML() {
    try {
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(new NoCloseInputStream(sock.getInputStream()));
        set docs root KVMessage
        if (key > 256 byte) {
            throw new KVException(new KVMessage("resp", "Oversized key"));
        }
        if (value > 262144 byte) {
            throw new KVException(new KVMessage("resp", "Oversized value"));
        }
        if (msgType is getreq) {
            add type and key as childNode;
            return generated get request;
        } else if (msgType is putreq) {
            add type, key and value as childNode;
            return generated put request;
        } else if (msgType is delreq) {
            add type and key as childNode;
            return generated delete request;
        } else if (msgType is resp) {
            if (put or delete resp) {
                add type and msg as childNode;
            } else {
                add type, key and value as childNode;
            }
            return generated response;
        } else {
            throw an exception;    //invalid msgType
        }
    } catch (KVException e) {
        //not enough data is available to generate a valid KV XML message
    }
}
```

sendMessage() takes in a socket and calls toXML() to generate the XML message. then writes it on the socket using java built-in printWriter.

```
sendMessage(Socket sock) {
    try {
        xml = toXML();
        pw = new printWriter(NoCloseOutputStream(sock));
        pw prints the message and writes on the socket;
        socket.shutdownOutput();
        pw.close();
    } catch (IOException e) {
        throw new KVException(new KVMessage("resp", "Network Error: Could not receive data"));
    }
}
```

Testing Plan

In order to unit test our KVMessage we made 8 test cases, 3 for creating and generating XML for messages and 5 for reading a message from a socket. Our cases are as follows:

1. To test that we could properly generate the right information for a put request, we create a KVMessage of type "putreq" and set the key and value to test values. Then we called *toXML()* on the message and asserted that the result contained a "¡Key¿" tag with the test key and a value tag with the test value.

2. To test that we could properly for a get request, we did something similar to what was said above but instead of type "putreq" we used type "getreq", set only the key and then asserted that the *toXML()* result contained only the key tag, and that it did not contain the value tag.

3. Next we generated a test response as above with the "resp" type and a test mesage, asserted that the message was there but that there was no "¡Key¿" or "¡Value¿" tags.

4. For reading from a socket, we mocked a socket using mockito, and provided some test XML to put into a byte array input stream that would be read from in the KVMessage. We then created a KVMessage using the mocked socket and ensured that it read the key, value and type fields correctly by asserting the return values of the corresponding get methods.

5. We repeat the following with a get request, ensuring that we can get the key, and that the value is null as well as the message.

6. We also repeat this with a ”resp” type message and ensure that we get the correct message string in return and null for the key and value.

7. Then we try to read an invalid message type from a socket, and ensure that the KVMessage(Socket) constructor throws a KVException.

8. Finally we create a message to send over the socket (which we are still mocking) this is just a plain string, not XML, and ensure that when we try to create a KVMessage with this socket an exception is thrown as well because it is not valid XML.

Task 2: KVClient

Correctness Constraints

- The client must generate proper XML messages to send to the server.
- The client must be fault tolerant and throw exceptions when it recieves and error from the server.

Declarations

There are no additional declarations eneded for this section.

Description

For KVClient we must implement the *connectHost*, *closeHost*, *put*, *get* and *del* methods.

The *connectHost* method is rather trivial to implement and requires only the creation of a new Socket with the hostname and port number of the server. The *closeHost* method is rather trivial as well because it takes the socket that is returned from *connectHost* after we're done with it and calls *close()* on that socket.

The *put*, *get* and *del* methods are a little more complicated but pseudo-code for each of these is below:

```
put(key, value)
    KVMessage message = new KVMessage("putreq");
    message.setKey(key);
    message.setValue(value);
    Socket connection = connectHost();
    message.sendMessage(connection);
    KVMessage response = new KVMessage(connection);
    if(!"Success".equals(response.getMessage())
        throw new KVException(response);

get(key)
    KVMessage message = new KVMessage("getreq");
    message.setKey(key);
    Socket connection = connectHost();
    message.sendMessage(connection);
    KVMessage response = new KVMessage(connection);
    if(!key.equals(response.getKey())
        throw new KVException(response);
    return response.getValue();

del(key)
    KVMessage message = new KVMessage("delreq");
    message.setKey(key);
    Socket connection = connectHost();
    message.sendMessage(connection);
    KVMessage response = new KVMessage(connection);
    if(!"Success".equals(response.getMessage())
        throw new KVException(response);
```

Testing Plan

For set-up/tear-down of our tests we would instantiate a SocketServer, KVClientHandler, KVServer and KVClient in order to use in each of our tests. In the set-up, after instantiating all the objects, we will add the hander to the SockerServer, and then connect our SocketServer and then spawn off a thread in which it runs. Our KVClient will connect to the same port that the sever is listening on.

Then for each of our tests our work-flows are as follows:

1. testPutGet: Use the client to put a test key and value. Then assert that the client get call with that test key returns the same test value.

2. testDel: Use the client to put a test key and value, then call *del* with that key, then call *get* with the same key and ensure that it throws a KVException.

3. testFail: Use the client to call *get* on a non-existent key and ensure that a KVException is thrown and that the message type is a ”resp” with message of ”Does not exist”

Task 3: Write-Through KVCache

Correctness Constraints

- Our cache should be set-associative.
- It should replace entries using the second chance algorithm.

Declarations

- *ArrayList¡LinkedList¡CacheNode¿¿ caches(numSets)* - An array of LinkedList¡CacheNode¿ which functions as our set of sub-caches.
- *WriteLock locks[numSets]* - An array of WriteLocks which restrict/allow access to the given set.

We also declare a CacheNode class for keeping track of the elements within our cache. Each cache item consists of the key and the value, as well as the reference bit for our second chance policy. Thus we define our CacheNode class to have those fields: String *key*, String *value*, boolean *reference*.

Description

Now we need to implement the following methods: the constructor, *get*, *put*, *del*, *getWriteLock*.

The constructor is fairly straight forward to implement. All we need to do is fill our array *caches* with new empty lists and create corresponding WriteLocks to put in our locks array. The *getWriteLock* method is also fairly simple, all we do is return the lock from the array corresponding to the index specified by *getSetId* of the key for which we are requesting a write lock.

The get method works as following:

```
get(key):
    int setId = getSetId(key);
    for(CacheNode node : caches[setId])
        if(node.key == key)
            node.reference = true;
            return node.key
    return null;
```

The del method is also quite similar:

```
del(key):
    int setId = getSetId(key);
    for(int i = 0; i < caches[setId].size(); i++)
        CacheNode node = caches[setId].get(i);
        if(node.key == key)
            caches[setId].remove(i);
```

The put method is the most complicated though. In order to ensure second-chance works we must check

```
put(key, value):
    int setId = getSetId(key);
    // Make sure our element isn't already there and update if it is
    for(CacheNode node : caches[setId])
        if(node.key == key)
            node.value = value
            node.reference = true
            updated = true
            return;

    if(caches[setId].size() == maxElemsPerSet)
        // We must proceed with second chance.
        CacheNode oldest = caches[setId].poll()
        while(oldest.referenced)
            oldest.referenced = false // Second chance
            caches[setId].offer(oldest)
            oldest = caches[setId].poll()

    // We've just dropped the oldest from the cache
    // Or there was still room
    caches[setId].offer(new CacheNode(key, value, false))
```

Testing Plan

In order to test our cache, we want to make sure that the elementary put/get/del operations all work as intended, but also that our replacement policy works as intended. So, in our set-up method we will initialize a new KVCache for each test with 1 set and 2 elements per set. Then our tests look as follows:

1. testPutGet: Call cache.put with a test key/value and then assert that a get call on the same key returns the correct value.

2. testDel: Puts a key/value in the cache using *put()* then call *del* on the key and then assert that a *get* call on that key returns null.

3. testOverwrite: Put a key/value into the store, then put the same key with a new value into the store, and finally assert that a get call returns the new value.

4. testReplacement: Put 3 values into the cache. Since our cache has a capacity of only two, the oldest value should get pushed out, so assert that calling get on the first value returns null. Then we call put with the second value, to put it back in, and then add another value. If our referencing policy works correctly, value 2 should have been referenced so when we put in value 4, value 3 will have to be removed. If we put a 5th value back in, 2 should get removed because its referenced value should be false due to second chance and so we can assert that after putting 5 into the cache, 2 is gone. Finally we will call get on 4, to make it referenced and put a value 6 into the cache, which should force 5 which was never referenced to be evicted.

Task 4: dumpToFile, restoreToFile, toXML

- KVStore is a way to store Key-Value pairs.
- dumpToFile() puts what is in a KVStore object into the file filename.
- restoreFromFile() actually replaces the KVStore object with the contents of the file.
- toXML() converts a KVStore to an XML string

Correctness Constraints

- Keys are non-null strings with 0 < length < 256 bytes.
- Values are non-null strings with 0 < length < 256 kilobytes.
- We will use the built-in javio.io.FileWriter and java.io.BufferedWriter classes for dumpToFile in KVStore.

Declarations

There are no new necessary declarations outside of each local declaration inside of each method.

Description

- We will need to import java.io.File, org.w3c.dom.*, org.w3c.dom.NodeList, javax.xml.parsers.*, among many others used as a component API of the Java API for XML Processing.
- toXML()
  KVStore.toXML() generates an XML string for KVStores. See code below:

```
KVStore.toXML(){
  try{
    Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
    Element rootElem = xmlDoc.createElement("KVStore");
    doc.appendChild(rootElem);
    iterate through this.store.keys using Map.Entry<String, String>{
      Get the key and value at the ith iteraition.
      Create a childElement = doc.createElement("KVPair");
      childElement.appendChild(doc.createElement("Key").setTextContent(key at ith iteration));
      childElement.appendChild(doc.createElement("Value").setTextContent(value at ith iteration);
      append the childElement to the rootElem
    }
    Create a transformer using TransformerFactory.newInstance.newTransformer() called transformer;
    Create a new StringWriter() called strWriter;
    Create a new StreamResult(strWriter) called streamResult;
    transformer.transform(new DOMSource(doc), streamResult);
    return strWriter.toString();
  }
  catch (ParserConfigurationException e){ // This could be thrown by the newDocumentBuilder() call
    e.printStackTrace(System.out);
  }
  catch(Exception e){
    e.printStackTrace(System.out);
  }
}
```

- dumpToFile()
  BufferedWriter is a character streams class. It allows us to write strings directly to the file. Because the FileWriter constructor and BufferedWriter.write() possibly throw an IOException, we catch those specifically. We also catch all other errors which could arise and then print a stack trace.

```
public void dumpToFile(String fileName){
  try{
    Create a new FileWriter(file_name) called file_writer;
    Create a new BufferedWriter(file_writer) called buffered_writer;
    String str = this.toXML();
    buffered_writer.write(str);
    buffered_writer.close();
  }
  catch (IOException e) { // From either BufferedWriter.write() or the
    // FileWriter constructor
    e.printStackTrace(System.out);
  }
  catch (Exception e) {
    e.printStackTrace(System.out);
  }
}
```

- restoreFromFile() replaces the KVStore object with the contents of the file using the DocumentBuilder, File, and Document classes.

```
public void restoreFromFile(String fileName){
  resetInventory();
  Create a new DocumentBuilder, called builder, using
    the call DocumentBuilderFactory.newInstance().newDocumentBuilder();
  Create new File(fileName) called file.
  Check if this file exists and is readable (if not, print a statement and return).
  Document d = builder.parse(file);
  Normalize the text representation using d.getDocumentElement().normalize();

  NodeList pairList = d.getElementsByTagName("KVPair");

  For each Node node in pairList{
    get the keyNode using pairList.item(i).getFirstChild();
    get the valueNode using pairList.item(i).getLastChild();
    store these this.store using the call
      this.store.put(keyNode.getNodeValue(), valueNode.getNodeValue());
  }

  // Caused from the DocumentBuilderFactory.newInstance() call
  catch(FactoryConfigurationError f){
    f.printStackTrace(System.out);
  } catch (IOException i) { // Caused by the DocumentBuilder.parse() call
    i.printStackTrace(System.out);
  } catch (SAXException s) { // Caused by the DocumentBuilder.parse() call
    s.printStackTrace(System.out);
  } catch (Exception e) {
    e.printStackTrace(System.out);
  }
}
```

- KVCache.toXML() is essentially the same as KVStore.toXML() except on the first level we iterate over each cache set and wrap all the contents of this within a $< Set >$ tag, then we iterate over each CacheNode and place the key/value contents within a $< CacheEntry >$ tag and keep note of whether the cache entry has been referenced in the attributes of that tag.

Testing Plan

Basically the big thing we want to test with the KVStore dump and restore methods are that the state is persisted to a file with all of the corresponding information and restored to the store back in the exact same state as previously when we restore from the file. So, to do this, we create a temp file using JUnit which we can read and write to. Then we follow the following workflow:

1. store.put("Maliena", "20");
2. assertEquals("20", store.get("Maliena")); // Just make sure that put/get work, for sanity
3. store.put("Hamel", "19");
4. store.put("Bryan", "24");
5. store.put("EunSeon", "20");
6. store.dumpToFile(dumpFile.getAbsolutePath());
7. store.del("Bryan");
8. store.del("Hamel");
9. store.put("Lisa", "50");
10. store.put("Maliena", "21");
11. store.restoreFromFile(dumpFile.getAbsolutePath());
12. assertFails(KVException, store.get("Lisa")); // Added fields should be removed
13. assertEquals("20", store.get("Maliena")); // Changed fields should be the old value
14. assertEquals("19", store.get("Hamel")); // Removed fields should be restored

Task 5: ThreadPool

Correctness Constraints

- Our ThreadPool must be thread-safe, in the case of multiple accesses from our WorkerThreads at the same time.

Declarations

In ThreadPool we will need to add the following:

- *Queue< Runnable > jobQueue* - A queue for holding the list of jobs, ie. a linked list. It will be @GuardedBy(this).

In WorkerThread we will need to add the following:

- *ThreadPool pool* - A reference to the pool in which this belongs.

Description

For the constructor of ThreadPool which takes a size, we will initialize the array of threads and fill it with the required number of threads and start them so they wait for jobs to run:

```
ThreadPool(size):
    threads[] = new Thread[size];
    for(int i = 0; i < size; i++)
        threads[i] = new WorkerThread(this)
        threads[i].start();
```

Then for our *addToQueue* method we need to do two things: add our job to the queue safely, and then notify anyone who is waiting for a job that a new one has been added. In order to make sure that our job is added safely we will synchronize this method and then we will call *notify()* to let waiting threads know a job has been added.

```
synchronized addToQueue(r):
    jobQueue.add(r);
    notify();
```

Finally for our *getJob* method, which will primarily be called by WorkerThreads, we must make sure that is safely removes a job from the queue if there is one present or blocks until there is one. This can be done in the following manner:

```
synchronized getJob():
    while(jobQueue.peek() == null)
        wait();
    return jobQueue.poll();
```

Then in the WorkerThread constructor we need to store the provided thread pool in the *pool* variable so we have a reference to the pool it was created for. After that all we need to do is implement run to get a job from the queue, run it until completion and then repeat endlessly:

```
run():
    while(true)
        Runnable toRun = pool.getJob();
        toRun.run();
```

Testing Plan

For testing our ThreadPool we will test two cases, that one thread can complete all the required jobs, and that 10 concurrent threads can complete all the tasks as well. To do this we will create a bogus test job which is a runnable that just increments an atomic integer and sleeps for 10 milliseconds. Then we test two cases:

1. testIndividualPool: create a pool with one worker thread, add 100 jobs, sleep for 3 seconds (should be more than enough time) and then assert that all the jobs were completed by checking that the atomic integer is 100.
2. testBigPool: create a pool with 10 worker threads, add 100 jobs, sleep for 3 seconds (again, this should be more than enough time) and then assert that the atomic integer has a value of 100.
3.

Task 6: SocketServer, KVClientHandler, KVServer

Correctness Constraints

- GET(Key k): Retrieves the key-value pair corresponding to the provided key.

- PUT(Key k, Value v): Inserts the key-value pair into the store.
- DEL(Key k): Removes the key-value pair corresponding to the provided key from the store.
- Reads (GETs) and updates (PUTs and DELETEs) are atomic
- All operations (GETs, PUTs, and DELETEs) must be parallel across different sets in the KVCache, and they cannot be performed in parallel within the same set.

Declarations

- *SocketServer.running* default initialized to false, tells us whether the server should continue accepting requests.

Description

SocketSever

For this class, all we need to do is implement the *connect*, *run*, *stop* and *closeSocket* methods.

The *connect()* method connects the server to the provided hostname and portname that it was instantiated with. Thus all we need to do is instantiate our server socket, to listen on the provided port.

The *run()* method opens the server to recieve connections and listens for them. Thus we need to keep in an endless loop, listening for clients and then handing them off to the handler. It will work in the following fashion:

```
run():
    while(running)
        Socket client = server.accept()
        handler.handle(client)
```

The *stop()* method stops the server from recieving new connections. This can just be done by setting the *running* variable to false. That way, after the current connection that is to be handled, the run method will exit.

The *closeSocket()* method closes and releases the socket that the server has been using to listen for connections. To do this we just need to call, *close()* on our ServerSocket.

KVClientHandler

This handler is expected to take a socket connection representing a client's request and read that data and perform the appropriate request to the KVServer. This can be done by creating a KVMessage on that socket to parse the message sent to the server. Then based on the request being made, we perform one of the KVServer methods of put/get/del. Depending on whether that request was a success, we create a message to return to the end user and we send it over the provided socket. This is also where we perform any final bulletproofing that needs to be done. This method should not throw any exceptions, but rather it should be catching all exceptions and returning a error KVMessage to the end user.

Our *ClientHandler.run()* method looks like the following:

```
run():
    KVMessage incoming = new KVMessage(client);
    String type = incoming.getMsgType()
    String key = incoming.getKey()
    String value = incoming.getValue()
    try
        if(type == "putreq")
            server.put(key, value)
        else if(type == "getreq")
            server.get(key)
        else if(type == "delreq")
            server.del(key)
        // We should not hit the else case ever because it's impossible to form a
        // KVMessage with an invalid type, but just in case we do the following
        else
            throw new KVException(new KVMessage("resp", "Unknown error: invalid message type provided"))
        // Since we got this far without an error our message must have been a success
        new KVMessage("resp", "Success").sendMessage(client)
    catch(KVException e)
        e.getMsg().sendMessage(client);
    finally
        client.close()
```

KVServer

This class delegates all requests for putting/getting keys through the cache and if necessary into the data store. It is also responsible for ensuring atomicity for example to make sure that a PUT/DEL request aren't executed at the same time causing problems. The methods which need to be implemented are *put*, *get*, *del*. Also, since our cache is a write-through cache, we must make sure that our writing methods, not only write to the cache, but also write to main memory.

For *put* we must put the value into our keystore, also if it is in the cache we must update it in the cache. Our atomicity requirements state that this must be a fully atomic operation but that it must be able to be run in parallel across different cache sets. For this to work we must wrap the cache call in the cache lock, but then we must wrap the whole thing in a synchronized call in order to ensure cache/keystore consistency.

```
synchronized put(key, value):
    synchronized(cache.getWriteLock(key)) {
        if(cache.get(key) != null)
            cache.put(key, value)
    }
    keystore.put(key, value)
```

For *get* we first check to see if the value is in our cache, if so we return it, otherwise we read out of our keystore and put it into our cache. To do this, we only need to acquire the cache lock, and then if that fails then we need to access the server so then we wrap the server commands in a synchronized block.

```
get(key)
    synchronized(cache.getWriteLock(key)) {
        result = cache.get(key)
        if(result == null)
            synchronized {
                result = server.get(key)
            }
            if(result == null)
                throw new KVException(new KVMessage("resp", "non existent key"))
            else
                cache.put(key, result)
    }
    return result
```

Finally for *del* we must check to see if the key is in our cache, if so, remove it from the cache, then we must remove it from the keystore if it is there. If not we throw an exception notifying the user that the key is non-existent.

```
synchronized del(key)
    if(server.get(key) == null)
```

```
        throw new KVException(new KVMessage("resp", "non existent key"))
    synchronized(cache.getWriteLock(key)) {
        cache.del(key)
    }
    server.del(key)
```

Testing Plan

For this section we will need to write tests for each of our classes.

For **SocketServer** we want to ensure that it can handle multiple requests at the same time, this can be done by having our SocketServer listen on a certain port, then create bunch of bogus request threads that connect to the listening socket, and send some information. We also create a test handler that handles our bogus requests just to ensure that they are all passed to the handler. We also want to ensure that the listening and stopping methods work, so we have the following test cases:

1. testListening: Here we just create a socket to our server host and port and try to send some data. As long as we do not get any exceptions we know all is well.

2. testStop: Here we call stop on the server, wait a second, and then try to send some data. The socket should send the data but it should not be handled so we should get no exceptions, but the handler should show 0 recieved messages. Then we close the server connection and try to send some data and expect that an exception happens.

3. testMultipleConnections: Here we create a bunch of Threads which send our test message to the server. Then we spawn several of them off and join them. After they all have completed we assert that the number of recieved messages by the handler is equal to the number of threads we spawned off.

The during set up and tear-down of each of the tests we create a new SocketServer, add a bogus handler that just counts the number of time it recieved the message ”test” and start the server in a new thread.

For **KVClientHandler** we need to make sure that each type of request is handled correctly, and that our code is bulletproofed so any bad information passed into the system is handled gracefully and fails without taking the rest of the system down. This can be done by passing a variety of sockets into the handler's *handle* method to which we write a information consisting of valid KVMessages as well as garbage messages and make sure we get the correct results written back. This leads to the following test cases:

1. testGoodMessage: Mock a socket connection, create a KVMessage and load it's text into the mock socket, then submit it to the handler and ensure that it performs the correct operations on the KVServer and that we get a KVMessage returned.

2. testBadMessage: Mock a socket connection, create a KVMessage which will inevitably lead to an error such a get of a non-existent key. Then have the handler handle that request and ensure that it handles the failure gracefully and returns the corresponding failure message.

3. testGarbageMessage: Once again we mock a socket connection and load it with a string that isn't even a valid XML message. Then we have the handler handle the message and we assert that it handled it gracefully and that it returned the correct failure message, ie. an XML error message.

For **KVServer** we will need to write tests which ensure that when we *put* a value to the server, a successive *get* call recieved the same value and after we *del* that key, a successive *get* call will fail with a non-existent key error. We will also want to make sure that the values we call *get* on the first time get added to the cache, and that after we *del* that key, it gets removed from the cache as well. To do this we needed to create another KVServer constructor that would allow us to supply a KVCache and KVServer ourselves in order to ensure that the correct behavior took place. After this our test cases look as follows:

1. testPut: Call server.put with a key and value, assert that key is in the store but not in the cache.

2. testGet: Put a value into the server, assert that a get call returns it and that it is put into the cache afterwards.

3. testDel: Put a value into the server, call delete on that value, assert that it's not in the cache and that calling get from the store throws an exception for that value. Then assert that calling get from the server also throws an exception.

4. testCacheDel: Put a value into the server, call get so it gets put into the cache, call delete on that value and then ensure that it is also removed from the cache.