Project 3: KVStore

Getting Started

The project skeleton has been pushed to GitHub in the repository `group<0>` in the directory `kvstore`. You can define additional classes and methods as you deem fit, but you must not modify the defined prototypes/interfaces/variables. The Javadocs provided should be helpful as a guide to the implementation of each method.

If you're using Vagrant, you can pull the skeleton by running the following commands:

- `cd ~/group`
- `git pull staff master`

If this doesn't work for you, double-check that the group0 repo is setup as a tracked repository:

- `git remote -v`

You should see staff linked to the group0 repo. If not, add the group0 repo and try again:

- `git remote add staff git@github.com:Berkeley-CS162/group0.git`

If you'd like to start from scratch, you can branch from staff/master:

- `git checkout -b staff/master`

The project is packaged for ant (http://ant.apache.org/manual/) and includes a basic testing framework. The files `SampleClient` and `SampleServer` are given as examples on how to run a client and server in a standalone fashion. To use them, simply run the main method of the `SampleServer` class followed by the main method of the `SampleClient` class. `EndToEndTemplate` includes this sequence in JUnit setup and teardown methods. The provided tests should serve as a sanity check. You are expected to add tests to existing classes and add new classes of tests to your suite.

Command Line

To run all JUnit tests (up to the first failure), from within your `kvstore` directory:

```
$ ant test
```

To run a specific class of tests:

```
$ ant runtest -Dtest=KVClientTest
```

You can also use the following commands to clean and compile the project (though the above commands automatically compile):

```
$ ant clean
$ ant compile
```

Note: Our ant build.xml file requires JDK 7 (javac 1.7.x). Upgrade if you do not have that installed locally. Use an inst machine if you do not want to upgrade locally.

Eclipse

You are welcome to use Eclipse exclusively while developing. However, it is STRONGLY recommended you run tests through Ant on vagrant (or through Eclipse) since you have to manually pass in arguments to the run configurations for each test that would otherwise be supplied by the ant build file.

To import the project, create a new Java Project with your `kvstore` directory as the location of the project and click Finish. Do not import the project as an Java Project with Ant as the build.xml file we provided does not directly support the `javac` target.

After importing the project, you can still run `ant` from within Eclipse by going to `Run > External Tools > External Tools Configurations`. Then create a new `Ant Build` configuration, specifying the path of your `Buildfile` (`build.xml`). Under the `Targets` configuration tab, choose `test [default]` and save this configuration with some name like "run all tests". Similarly, you can create configurations to run only single suites of tests at a time by selecting the target `runtest` and supplying the `-Dtest=<Testclass>` (like above) as an argument on the `Main` configuration tab. This *should* be

equivalent to running the JUnit tests directly. We will, however, be grading on `hive` with `ant`, so you should make sure your project passes our tests there too before you submit. Note: If you are ever unable to get the tests working locally and you notice that the local port of the client socket does not match the remote port on the server-side socket, it is possible that Cisco AnyConnect is the culprit. You should uninstall it if you don't use it.

Checkpoint 1 (11/21/2014)

In checkpoint 1, you will implement a single-node key-value server. Multiple clients will be issuing PUT, GET, and DEL requests to a single KVServer, which contains a cache backed by a KVStore. Please see the introduction page for more information.

`KVClient`

Implement the `KVClient` class, which issues requests and handles responses. To send a request to the server over the network, requests must be serialized (read: converted to XML) to be sent through a socket (`java.net.Socket`). Use the `KVMessage` functionality you will implement in task 2 to send requests and receive responses. Note: sockets are only good for sending data and receiving data once, meaning every request must be made with a new socket (and as a result, the server processes each request completely independently). Do not use the `KVMessage` constructor with a timeout (read about it in task 2).

`KVMessage`

You will be implementing a message class with serialization and deserialization. While parsing the `InputStream` of a socket, many parsers call close on the stream when finished reading. This inadvertently closes the `OutputStream` of the socket as well, preventing the user from sending a response through the same socket. To deal with this issue, we provide a private class, `NoCloseInputStream`, which you should use as appropriate. The `KVMessage` constructor with timeout should not be directly used in this checkpoint, as we have not given you a reason to use it yet. You will still need to implement it, as another constructor (that you should be using) calls it with a timeout of 0. Check out the Socket API for more information. You will need to implement the `KVMessage(Socket sock, int timeout)` constructor for checkpoint 1, but you should only directly call `KVMessage(Socket sock)` in checkpoint 1 since we will not worry about timeouts (there should always be a response). In checkpoint 2 and 3, you will call the timeout constructor directly.

A `KVMessage` must serialize to the following formats. Italicized fields should be replaced with the appropriate values.
Newlines/indentation are added only for readability on the spec.

putreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="putreq">`
	`<Key>`*key*`</Key>`
	`<Value>`*value*`</Value>`
	`</KVMessage>`

getreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="getreq">`
	`<Key>`*key*`</Key>`
	`</KVMessage>`

delreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="delreq">`
	`<Key>`*key*`</Key>`
	`</KVMessage>`

successful putreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="resp">`
	`<Message>Success</Message>`
	`</KVMessage>`

successful getreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="resp">`
	`<Key>`*key*`</Key>`
	`<Value>`*value*`</Value>`
	`</KVMessage>`

successful delreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="resp">`
	`<Message>Success</Message>`
	`</KVMessage>`

Unsuccessful put/get/del request	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="resp">`
	`<Message>`*Error Message*`</Message>`
	`</KVMessage>`

`KVCache`

Implement a set-associative `KVCache` with the second-chance eviction policy within each set. Each set in the cache will have a fixed number of entries, and evict entries (when required) using the second-chance algorithm. Note that this is not the clock algorithm, thus do not keep track of clock hands.

- When get is called on a key, the reference bit of that entry is set to `True`.
- If put is called on an existing key, the value is overwritten in the existing entry and the reference bit is set to `True`. The entry's position in the queue remains the same.
- If put is called while the set is not full, the entry should be added to the end of the queue immediately.
- If put is called when the set is full, the first entry with a `False` reference bit while cycling through the queue should be replaced.

`KVCache.toXML()` output example:

```
 <?xml version="1.0" encoding="UTF-8"?>
<KVCache>
  <Set Id="id">
    <CacheEntry isReferenced="true/false">
      <Key>key</Key>
      <Value>value</Value>
    </CacheEntry>
  </Set>
</KVCache>
```

There should be as many `Set` elements as there are sets, and within each set, there should be as many `CacheEntry` elements as there are entries *that exist* in each set. As mentioned earlier, the number of sets and the number of elements in each sets will be given in the constructor. Sets must have ids starting from `0`.

`KVServer`, `KVStore`

All requests (get/put/del) are atomic in that they must modify the state of both the cache and the store together. Requests must be parallel across different sets (as decided by `KVCache`) and serial within the same set. To enforce atomicity from `KVServer`, obtain a lock from the `KVCache` for the set the key belongs to. You do not need to implement readers/writers. You should follow a **write-through** caching policy. If a key exists in the cache for a get request, do not access the store (imagine the store access takes a long time, even though it doesn't as provided).

`KVStore.dumpToFile` output format:

```
 <?xml version="1.0" encoding="UTF-8"?>
<KVStore>
  <KVPair>
   <Key>key1</Key>
   <Value>value1</Value>
  </KVPair>
  <KVPair>
    <Key>key2</Key>
    <Value>value2</Value>
  </KVPair>
</KVStore>
```

`SocketServer`, `ServerClientHandler`, `ThreadPool`

Your server must be able to service requests from the client. The `SocketServer` class will create a serversocket (`java.net.ServerSocket`) that listens on a port for connections. A socket should be passed to the `ServerClientHandler` for each request that comes in. The `ThreadPool` will then execute jobs created by the handler in parallel and return responses for each request. A thread pool allows a system to service jobs concurrently, yet limiting the total number of threads spawned in the system (rather than spawning a thread per request).

Test Cases:

All of the tests in the following files should pass for Checkpoint 1. Additionally, you should use EndToEndTemplate.java to construct your own end to end test.

As per usual, your design document is due along with checkpoint 1.

- `KVCacheTest.java`
- `KVClientTest.java`
- `KVMessageTest.java`

- `KVServerTest.java`
- `KVStoreTest.java`
- `SocketServerTest.java`
- `ThreadPoolTest.java`

Checkpoint 2 (12/03/2014)

In checkpoint 2 you will extend the implementation of checkpoint 1 to a distributed case. The client will now communicate with the master server instead of the storage nodes (KVServers) directly, and the master will handle communicating with the slave servers and returning the appropriate response. In checkpoint 2, you will only implement the setup of the distributed system along with get requests. You will not need to implement Two Phase Commit logic. You may assume your slaves are invincible and never die.

Message Passing

Modify KVMessage to allow two-phase commit messages. See message specs in tables below.

Slave Registration

Implement registration logic in the `TPCRegistrationHandler`, `TPCSlaveInfo`, and `TPCMasterHandler.registerWithMaster`. Implement the methods `registerSlave`, `findFirstReplica` and `findSuccessor` in `TPCMaster` which will be used to find the slaves involved in a transaction. Then implement `TPCClientHandler`.

Master Server GET handling

Implement `handleGet` in `TPCMaster` to correctly handle GET requests. GET requests from the client are serial within the same set but parallel across different sets. Sets are defined by the KVCache of the Master. You do not need to implement two-phase commit logic (Note that GET is not a TPC command). If the key requested by the client is resident in the cache, return it and do NOT proceed to forward the request to the slave.

Slave Server GET handling

Implement enough of `TPCMasterHandler` to correctly respond to GET requests on the slaves.

Note that this project may require a bit more collaboration and understanding between tasks than previous projects as some tasks are dependent on the implementation of others: (2,3), (3,4), (4,5), (5,1).

TPC message formats

`put/del/get` requests should remain unchanged from project 3 since they have no two-phase semantics. There are no newlines in these messages. Italicized fields should be replaced with the actual value.

putreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="putreq">`
	`<Key>`*key*`</Key>`
	`<Value>`*value*`</Value>`
	`</KVMessage>`

delreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="delreq">`
	`<Key>`*key*`</Key>`
	`</KVMessage>`

getreq	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="getreq">`
	`<Key>`*key*`</Key>`
	`</KVMessage>`

ready vote	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="ready">`
	`</KVMessage>`

abort vote	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="abort">`
	`<Message>`*Error Message*`</Message>`
	`</KVMessage>`

commit decision	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="commit">`
	`</KVMessage>`

abort decision	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="abort">`
	`</KVMessage>`

ack	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="ack">`
	`</KVMessage>`

Checkpoint 3 (12/10/2014)

`TPCLog and Slave Durability`

Implement proper logic for logging the state of slave servers and for rebuilding from the log after unexpected slave server termination. Note that appendAndFlush is implemented for you in TPCLog and requires you to log strictly the ACTIONS asked of the slave server in the form of KVMessages. You therefore may NOT devise some organic implementation that allows you to literally save the contents of the slave's KVStore and rebuild resurrected slaves by reading off a list of inserted keys/values.

`Two Phase Commit logic`

Implement `handleTPCRequest` in `TPCMaster` such that the master correctly handles all TPC operations, including timeouts and slave death. Finish the slave logic in `TPCMasterHandler` to correctly respond to PUTs and DELs.

Testing

Testing for this project is complicated. Expect to spend a significant portion of your time testing (~half the time of the project). We expect extensive use of Mockito and/or PowerMock. We have provided you with all of the unit tests for checkpoint 1 to ensure you are implementing the most basic single node case correctly. However, you are only provided with a subset of the tests required to gain full credit for checkpoints 2 and 3 and no end to end test for checkpoint 1. We expect you to leverage your knowledge of multithreaded programming and mocking to test your own code, both at the unit level and for full end to end functionality. To get you started, a basic end-to-end test setup has been provided in `TPCEndToEndTemplate` and `TPCEndToEndTest`. We expect you to submit your test cases along with the respective checkpoints. You will be required to describe them in your final design doc.

Important Tips and Resources

- Java Threads: (https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html)Up until now, you have been programming low level functionalities under the assumption of multithreading semantics. To successfully complete and test this project, you must now actually use your multithreading and synchronization skills to create, run, and synchronize multiple threads. For this you can use the Java Thread API. Threading is useful when you need to accomplish tasks asynchronously and concurrently; an example of this is using threads (pulled from a finite threadpool) in ServerClientHandler to service requests from the client asynchronously, or forking off multiple threads acting as clients that concurrently but asynchronously send TPC requests to the master server.

- Java Runnable interface: (https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) If you looked at the Java Thread API, then you probably noticed that the constructors expect a Runnable object as an argument. A Java Thread targeting some runnable object R will simply execute R.run() (a function that must be defined as part of the Runnable interface) when the thread executes. The thread finishes when the end of run() is reached. If you desire a scenario using multiple threads that concurrently perform different tasks, simply define an object that extends Runnable for each task and define the run() method appropriately. **This also means that any class that services requests using a threadpool will need to have some Runnable class defined that has an appropriate run() function to be executed by each thread.**

- Mocking and Mockito: (https://code.google.com/p/mockito/)Mocking is a simple but amazingly powerful tool that allows unit testing of otherwise hopelessly entangled subsections of your code that have many interdependencies. A "mocked" class fools your program into thinking it is an instance of the class it is mocking, but under the hood you have many tools to freely define what it will do in response to certain stimuli (such as what to return when certain member functions are called, or even what do do when specific arguments are used). We are requiring you to write at least three unit tests for checkpoints 2 and 3 (three each) that we will be looking for while grading your final design.

XML Serialization

Some skeleton code for XML serialization has been provided so that direct modification of DOM nodes is not required. In the files `KVCache.java`, `KVMessage.java`, and `KVStore.java`, you will see methods `marshalTo(OutputStream os)` and `String toXML()`, which will marshal the result of `getXMLRoot()` to a string. It is your job to fill out `getXMLRoot()` to return a `JAXBElement` which represents the XML representation of the object being serialized.

We have included autogenerated classes to define a schema for serializing each kind of serializable object, each containing methods to modify the appropriate fields in its serialization. For example, a serializable `KVMessage` object is represented by a `KVMessageType`, which contains methods to populate every possible field in its serialization. The serialization of a `KVMessage` could potentially contain any of the following fields or parameters: "key", "value", "type", "message", so `KVMessageType` will have setter and getter methods for all of them in the form `setKey`, `setValue`, `setType`, `setMessage`, `getKey`, etc.

To return the appropriate `JAXBElement` for a `KVMessage`, you will need to call a factory method to create a `KVMessageType`, set its contents, and then construct a `JAXBElement` from the `KVMessageType`. Consider the following output format for a "putreq" `KVMessage`, and the code snippet that might be used to generate the appropriate `JAXBElement`

putreq

```
<?xml version="1.0" encoding="UTF-8"?>
<KVMessage type="putreq">
<Key>key</Key>
<Value>value</Value>
</KVMessage>
```

```
ObjectFactory factory = new ObjectFactory();
KVMessageType xmlKVMessage = factory.createKVMessageType();
xmlKVMessage.setKey(key);
xmlKVMessage.setValue(value);
xmlKVMessage.setType("putreq");
JAXBElement root = factory.createKVMessage(xmlKVMessage);
```

The attributes of such a schema class need not only be strings; they can also be lists or other complex data structures. See the example in `KVCache.getXMLRoot()` for more details.

General Rules

- DO NOT MODIFIY ANY PART OF THE SKELETON YOU ARE PROVIDED. This includes any constructors and variables initialized within them. Tampering with the skeleton can break the test cases used to grade your code.
- All XML serialization and deserialization must use Java XML libraries (you may want to look into `javax.xml`, `org.w3c.dom` and `org.xml.sax`). You are not allowed to use string concatenation for any XML-related tasks. You do not need to use these built-in libraries, but note that several tasks involve XML parsing and that your group should be consistent in library usage (use the same one across the project).
- You may use external libraries by adding jar files to your `lib` directory. The `build.xml` will include all `lib/*.jar` files in your classpath (see testing section below for more details). We will grade your submission with your `lib` directory but with our own `build.xml` which will be similar to the one provided). You should check with your TA or on Piazza before adding jar files as we generally discourage using external libraries for anything other than XML [de]serialization as it increases the possibility of compatibility issues during grading.
- Do not use any built-in thread-safe data structures unless we have provided it in our skeleton. This means that you should use the `synchronized` keyword or a `ReentrantLock` to protect non thread-safe data structures like `HashMap` and `LinkedList`.
- Bulletproof your code, such that the key-value server does not crash while processing a request.
- If an error occurs at any point while processing a request *that prevents a successful response*, a `KVException` should be propogated to the client. The user of `KVClient` should see a `KVException` containing a `KVMessage` with an error string from `KVConstants.java`. We will be testing using this set of error messages. You may define additional error message if you feel the need, but the provided set of errors should be sufficient for all major cases. **Every member should read through `KVConstants`.**

Submission

As per usual, push submissions to release/project3/checkpoint<#> by 11:59 PM on the due date. The final design document will be due December 12th.

Acknowledgements

Thanks to George Yiu for reorganizing this project in Spring 2014 and to Nick Chang for his contributions to the autograder :)