Initial Design Document: Project 3

Group #6: Elise McCallum (cs162-bc), Danube Phan (cs162-aw), Thamine Dalichaouch (cs162-co), Vladimir Ponomarenko (cs162-bb), Jack Deng (cs162-fb)
GSI: Prashanth Mohan : Section 102

Part I. PingPong Program

Overview:

First an EC2 account needs to be set up as per the instructions given by the EC2 Access Guide. Then one must create a new EC2 instance, launch it, and SSH into it. To create the PingPong program, one must make use of TCP sockets that listen for connections on port 8081 and service each incoming request with the string ‘pong’. The java.net.Socket and java.net.ServerSocket classes are to be used for this.

Correctness Constraints:

· Each incoming TCP request on the socket must be serviced with the string ‘pong’.

Declarations:

No additional declarations in existing classes.

Descriptions:

The PingPong class has to be created.

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;

class PingPong {
    public static void main(String args[]) {
        create a new String called output and set it equal to ‘pong’;
        try {
            create a ServerSocket called srvr and bind it to port 8081;
            while (True) {
                try {
                    create a Socket called skt and set it equal to srvr.accept();
                    try {
                        create a PrintWriter called out from the
                        skt.getOutputStream();
                        print output to out;
                        close out;
                    }
                    catch IOException e {
                        //Failed to print output to out
                    }
                    close skt;
                }
                catch IOException e {
                    //Error accepting on port 8081
                }
            }
```

```
                }
                catch IOException e {
                        //Error listening on port 8081
                }
        }
}
```

Testing Plan:

1. Create a client that opens a socket to port 8081 and prints out the response from the server. Run the server, then run the client, and check if the output is equal to 'pong'. The pseudocode for the client is as follows:

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;

public class PingPongTest {
        public static void main (String args[]) {
                try {
                        create a Socket called skt and connect it to port 8081;
                        create a new BufferedReader called in and read the skt input stream into it;
                        create a String called line and set it equal to in.readLine();
                        print out line;
                        close in;
                }
                catch IOException e {
                        //something went wrong with the I/O
                }
        }
}
```

Part II. Implementing KVClient and KVMessage

Overview:

This part of the project deals with the implementation of KVClient and KVMessage. KVClient issues requests to the server to put, get, or delete key-value pairs and parses the responses it receives to return the appropriate values. The KVMessage class manages all the information needed to construct such requests, and can construct messages from given values or an XML input steam. KVMessage also handles the marshalling logic to convert an Object of an arbitrary type to a String, and to convert a String to an Object of an arbitrary type. The KVMessage class also converts requests to XML strings.

Correctness Constraints:

· If a KVMessage contains invalid XML, it should throw a KVException.
· Any malformed message should generate a KVException.
· A put request should insert a new key-value pair into the KVStore, and return a message of "Success" if successful (regardless if a value was overwritten or not).
· If a put request is unsuccessful, it should return an error message.
· A get request should return the unmarshalled value stored with that key.

· A delete request should throw an exception if the key to be deleted does not exist.

Declarations:
In KVMessage.java

```
private String status = null;
private String message = null;

//additional constructor added to accommodate requests using status and message variables
public KVMessage(String msgType, String key, String value, String status, String message) {
        set this msgType to msgType;
set this key to key;
set this status to status;
set this value to value;
set this message to message;
}
```

Descriptions:
In KVMessage.java
The following five methods are used to protect outside programs from manipulating the set values of message type, key, value, status, and message but still allow them to retrieve the stored values of said variables in a given KVMessage.

```
public String getType() {
        return this.msgType;
}
public String getKey() {
        return this.Key;
}
public String getValue() {
        return this.Value;
}
public String getStatus() {
        return this.status;
}
public String getMessage() {
        return this.message;
}
```

This method is provided to convert a String object to a general type by converting it first to an array of bytes, creating and object input stream from those bytes and reading the object created, which is now the object of the type specified in the input String.

```
public static Object convertToGeneralType(String info) throws KVException {
        try {   create a byte array of the input info using the char set UTF-8;
                create an object input stream using the data array of bytes as an input stream;
                read the object from the input stream;
                return the object read;   }
        catch (Exception e) { throw new KVException(null); }
}
```

This method is provided to convert a Serializable object to a String by first converting it to a byte array output stream and then writing the input information to the output stream created with those bytes. The stream is then closed and the String is created using the byte array and the char set UTF-8, and returned.

```
public static String convertToString(Serializable info) throws KVException {
    try {   create a new byte array output stream;
            construct a new object output stream using the byte array;
            write the input object info to the output stream;
            close the stream;
            create a string object using the byte array and the char set UTF-8;
            return that string;     }
    catch (Exception e) { throw new KVException(null); }
}
```

This is another method used for creating a KVMessage. Using an XML DOM Parser, this method creates a DOM tree for the possible information that could be stored in the message, parses the tree, and sets the appropriate values as they are found. The message type is then set using the same XML parser. An exception is thrown if any of these steps fail.

```
public KVMessage(InputStream input) throws KVException {
    try {
        set dbf to a new instance of DocumentBuilderFactory;
        set db to dbf.newDocumentBuilder();
        set the dom to db.parse(input);
        set docElement to the document element from the dom;
        create a String array called nodes consisting of {"Key", "Value", "Status", "Message"};
        create a String array called values of length 4;
        for each element in the array nodes {
            create a NodeList for the elements matching the tag of the specific element;
            if (elementNodes is not null and the length of the NodeList is 1)
                create an element which is the first and only element in the node list;
                set the corresponding value in the values array to be the node value;
        }
        set this key to the first element of values array;
        set this value to the second element of values array;
        set this status to the third element of values array;
        set this message to the fourth element of values array;
        create a NodeList of elements matching the KVMessage tag;
        if (node list is not null)
            create an element which is the first element in the node list;
            set this msgType to the value of the attribute type;
    }
    catch (Exception e) {
        throw new KVException(null);
    }
}
```

This method converts a KVMessage to XML using an XML DOM Parser to generate XML. It first creates an XML tree, and then attaches as the root node a KVMessage tag. The appropriate values of message information are added as children to the root node if they are present. The type is also set as an attribute for the KVMessage node. The tree is then transformed to a String and returned. Any malformed XML will trigger an exception.

```
public String toXML() throws KVException {
        if (this.msgType == null)
                return null;
        else {
                try {   set dbf to a new instance of DocumentBuilderFactory;
                        set db to dbf.newDocumentBuilder();
                        set dom to new Document;
                        create the root element as an element with tag KVMessage;
                        set the type attribute of root element to this msgType;
                        append this root as a child to the dom;
                        if (this key is not null) {
                                create an element called keyNode with tag Key;
                                set the text of the node to be the value of this.key;
                                append this node to the root;
                        }
                        if (this value is not null) {
                                create an element called keyNode with tag Value;
                                set the text of the node to be the value of this.value;
                                append this node to the root;
                        }
                        if (this status is not null) {
                                create an element called keyNode with tag Status;
                                set the text of the node to be the value of this.status;
                                append this node to the root;
                        }
                        if (this message is not null) {
                                create an element called keyNode with tag Message;
                                set the text of the node to be the value of this.message;
                                append this node to the root;
                        }
                        create a new instance of a transformerFactory;
                        create a new transformer from that factory;
                        create a StreamResult result using a new StringWriter;
                        set the source to be a new DOMSource using the dom;
                        transform data from source to result;
                        convert the stringwriter to a string;
                        return xml string;
                } catch (Exception e) {
                        throw new KVException(null);
                }
        }
}
```

In KVClient.java
The put method first converts the key and value parameters to Strings using the method defined in

KVMessage.java and utilizes these Strings to create a KVMessage of type put request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, returning true if the transaction is successful and false otherwise.

```
public boolean put(K key, V value) throws KVException {
        set strKey to marshalled value of key;
        set strValue to marshalled value of value;
        set request to be new KVMessage("putreq", strKey, strValue);
        set s to new Socket(server, port);
        connect s to the server’s address for 10000000 milliseconds (timeout);
        set out to new ObjectOutputStream(s.getOutputStream());
        convert the KVMessage to XML;
        write the request Object XML to the OutputStream out;
        set response to new KVMessage(s.getInputStream());
        if (response message type is “resp”) {
                if (response message is “Success”)
                        return true;
                else if (response message is not “Success”) {
                        throw new KVException(response);
                }
        }
        return false;
}
```

The get method first converts the key parameter to a String using the method defined in KVMessage.java and utilizes this String to create a KVMessage of type get request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, returning true if the unmarshalled value, if one exists, and null otherwise.

```
public V get(K key) throws KVException {
                set strKey to marshalled value of key;
                set request to new KVMessage("getreq", strKey, null);
                set s to new Socket(server, port);
                connect s to the server’s address for 10000000 milliseconds (timeout);
                set out to new ObjectOutputStream(s.getOutputStream());
                convert the KVMessage to XML;
                write the request Object XML to the OutputStream out;
                set response to new KVMessage(s.getInputStream());
                if (response message type is “resp”) {
                        if (response message is not null)
                                throw new KVException(response);
                        else {
                                set value to Value of response;
                                unmarshal the Value to a string;
                                return stringValue;
                        }
                }
                return null;
        }
```

The delete method first converts the key parameter to a String using the method defined in KVMessage.java and utilizes this String to create a KVMessage of type get request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, throwing an exception if the response message is not null, i.e. is an error message.

```
public void del(K key) throws KVException {
        set strKey to marshalled value of key;
        set request to new KVMessage("delreq", strKey, null);
        set s to new Socket(server,port);
        connect s to the server’s address for 10000000 milliseconds (timeout);
        set out to new ObjectOutputStream(s.getOutputStream());
        convert KVMessage to XML;
        write the request Object XML to the OutputStream out;
        set response to new KVMessage(s.getInputStream());
        if (response type is “resp” and response message is not “Success”)
                throw new KVException(response);
}
```

Testing Plan:

1. Create dummy messages of types putreq, getreq, delreq, and the various responses and test that they correctly convert to properly formatted XML.
2. Create an InputStream and test that it correctly extracts information or throws an exception if given malformed XML.
3. Convert an Object of a certain type that is not String to a String.
4. Convert a String to an Object that is not of type String.
5. Issue and parse a get request.
6. Issue and parse a put request.
7. Issue and parse a delete request.

Part III. Implementing a ThreadPool

Overview:

The thread pool should accept different tasks and execute them asynchronously. The threadpool should maintain a queue of tasks submitted to it, and should assign it to a free thread as soon as it is available. Ensure that the addToQueue interface to the threadpool is non-blocking. The Key-Value server will use the threadpool to parallelize data storage into the dummy storage system provided (KVStore). The logic is shown in this diagram:

Initial Design Document: Project 3

Group #6: Elise McCallum (cs162-bc), Danube Phan (cs162-aw), Thamine Dalichaouch (cs162-co), Vladimir Ponomarenko (cs162-bb), Jack Deng (cs162-fb)

GSI: Prashanth Mohan : Section 102

Part I. PingPong Program

Overview:

First an EC2 account needs to be set up as per the instructions given by the EC2 Access Guide. Then one must create a new EC2 instance, launch it, and SSH into it. To create the PingPong program, one must make use of TCP sockets that listen for connections on port 8081 and service each incoming request with the string ‘pong’. The java.net.Socket and java.net.ServerSocket classes are to be used for this.

Correctness Constraints:

- Each incoming TCP request on the socket must be serviced with the string ‘pong’.

Declarations:
No additional declarations in existing classes.

Descriptions:
The PingPong class has to be created.

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;

class PingPong {
        public static void main(String args[]) {
                create a new String called output and set it equal to ‘pong’;
                try {
                create a ServerSocket called srvr and bind it to port 8081;
                while (True) {
                create a Socket called skt and set it equal to srvr.accept();
                create a PrintWriter called out from the skt.getOutputStream();
                print output to out;
                close out;
                close skt;
        }
        catch (Exception e) {
        }
}
}
```

Testing Plan:

1. Create a client that opens a socket to port 8081 and prints out the response from the server. Run the server, then run the client, and check if the output is equal to ‘pong’.

Part II. Implementing KVClient and KVMessage

Overview:
This part of the project deals with the implementation of KVClient and KVMessage. KVClient issues requests to the server to put, get, or delete key-value pairs and parses the responses it receives to return the appropriate values. The KVMessage class manages all the information needed to construct such requests, and can construct messages from given values or an XML input steam. KVMessage also handles the marshalling logic to convert an Object of an arbitrary type to a String, and to convert a String to an Object of an arbitrary type. The KVMessage class also converts requests to XML strings.

Correctness Constraints:

- If a KVMessage contains invalid XML, it should throw a KVException.
- Any malformed message should generate a KVException.
- A put request should insert a new key-value pair into the KVStore, and return a message of “Success”

if successful (regardless if a value was overwritten or not).

- If a put request is unsuccessful, it should return an error message.
- A get request should return the unmarshalled value stored with that key.
- A delete request should throw an exception if the key to be deleted does not exist.

Declarations:

In KVMessage.java

```
private String status = null;
private String message = null;

//additional constructor added to accommodate requests using status and message variables
public KVMessage(String msgType, String key, String value, String status, String message) {
        set this msgType to msgType;
set this key to key;
set this status to status;
set this value to value;
set this message to message;
}
```

Descriptions:

In KVMessage.java

The following five methods are used to protect outside programs from manipulating the set values of message type, key, value, status, and message but still allow them to retrieve the stored values of said variables in a given KVMessage.

```
public String getType() {
        return this.msgType;
}
public String getKey() {
        return this.Key;
}
public String getValue() {
        return this.Value;
}
public String getStatus() {
        return this.status;
}
public String getMessage() {
        return this.message;
}
```

This method is provided to convert a String object to a general type by converting it first to an array of bytes, creating and object input stream from those bytes and reading the object created, which is now the object of the type specified in the input String.

```
public static Object convertToGeneralType(String info) throws KVException {
        try {   create a byte array of the input info using the char set UTF-8;
                create an object input stream using the data array of bytes as an input stream;
                read the object from the input stream;
                return the object read;     }
        catch (Exception e) { throw new KVException(null); }
}
```

This method is provided to convert a Serializable object to a String by first converting it to a byte array output stream and then writing the input information to the output stream created with those bytes. The stream is then closed and the String is created using the byte array and the char set UTF-8, and returned.

```
public static String convertToString(Serializable info) throws KVException {
        try {   create a new byte array output stream;
                construct a new object output stream using the byte array;
                write the input object info to the output stream;
                close the stream;
                create a string object using the byte array and the char set UTF-8;
                return that string;           }
        catch (Exception e) { throw new KVException(null); }
}
```

This is another method used for creating a KVMessage. Using an XML DOM Parser, this method creates a DOM tree for the possible information that could be stored in the message, parses the tree, and sets the appropriate values as they are found. The message type is then set using the same XML parser. An exception is thrown if any of these steps fail.

```
public KVMessage(InputStream input) throws KVException {
        try {
                set dbf to a new instance of DocumentBuilderFactory;
                set db to dbf.newDocumentBuilder();
                set the dom to db.parse(input);
                set docElement to the document element from the dom;
                create a String array called nodes consisting of {"Key", "Value", "Status", "Message"};
                create a String array called values of length 4;
                for each element in the array nodes {
                        create a NodeList for the elements matching the tag of the specific element;
                        if (elementNodes is not null and the length of the NodeList is 1)
                                create an element which is the first and only element in the node list;
                                set the corresponding value in the values array to be the node value;
                }
                set this key to the first element of values array;
                set this value to the second element of values array;
                set this status to the third element of values array;
                set this message to the fourth element of values array;
                create a NodeList of elements matching the KVMessage tag;
                if (node list is not null)
                        create an element which is the first element in the node list;
                        set this msgType to the value of the attribute type;
        }
        catch (Exception e) {
                throw new KVException(null);
        }
}
```

This method converts a KVMessage to XML using an XML DOM Parser to generate XML. It first creates an XML tree, and then attaches as the root node a KVMessage tag. The appropriate values of message information are added as children to the root node if they are present. The type is also set as an attribute for the KVMessage node. The tree is then transformed to a String and returned. Any malformed XML will trigger an exception.

```
public String toXML() throws KVException {
    if (this.msgType == null)
        return null;
    else {
        try {   set dbf to a new instance of DocumentBuilderFactory;
                set db to dbf.newDocumentBuilder();
                set dom to new Document;
                create the root element as an element with tag KVMessage;
                set the type attribute of root element to this msgType;
                append this root as a child to the dom;
                if (this key is not null) {
                    create an element called keyNode with tag Key;
                    set the text of the node to be the value of this.key;
                    append this node to the root;
                }
                if (this value is not null) {
                    create an element called keyNode with tag Value;
                    set the text of the node to be the value of this.value;
                    append this node to the root;
                }
                if (this status is not null) {
                    create an element called keyNode with tag Status;
                    set the text of the node to be the value of this.status;
                    append this node to the root;
                }
                if (this message is not null) {
                    create an element called keyNode with tag Message;
                    set the text of the node to be the value of this.message;
                    append this node to the root;
                }
                create a new instance of a transformerFactory;
                create a new transformer from that factory;
                create a StreamResult result using a new StringWriter;
                set the source to be a new DOMSource using the dom;
                transform data from source to result;
                convert the stringwriter to a string;
                return xml string;
        } catch (Exception e) {
                throw new KVException(null);
        }
    }
}
```

In KVClient.java

The put method first converts the key and value parameters to Strings using the method defined in KVMessage.java and utilizes these Strings to create a KVMessage of type put request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, returning true if the transaction is successful and false otherwise.

```
public boolean put(K key, V value) throws KVException {
        set strKey to marshalled value of key;
        set strValue to marshalled value of value;
        set request to be new KVMessage("putreq", strKey, strValue);
        set s to new Socket(server, port);
        connect s to the server’s address for 10000000 milliseconds (timeout);
        set out to new ObjectOutputStream(s.getOutputStream());
        convert the KVMessage to XML;
        write the request Object XML to the OutputStream out;
        set response to new KVMessage(s.getInputStream());
        if (response message type is “resp”) {
                if (response message is “Success”)
                        return true;
                else if (response message is not “Success”) {
                        throw new KVException(response);
                }
        }
        return false;
}
```

The get method first converts the key parameter to a String using the method defined in KVMessage.java and utilizes this String to create a KVMessage of type get request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, returning true if the unmarshalled value, if one exists, and null otherwise.

```
public V get(K key) throws KVException {
                set strKey to marshalled value of key;
                set request to new KVMessage("getreq", strKey, null);
                set s to new Socket(server, port);
                connect s to the server’s address for 10000000 milliseconds (timeout);
                set out to new ObjectOutputStream(s.getOutputStream());
                convert the KVMessage to XML;
                write the request Object XML to the OutputStream out;
                set response to new KVMessage(s.getInputStream());
                if (response message type is “resp”) {
                        if (response message is not null)
                                throw new KVException(response);
                        else {
                                set value to Value of response;
                                unmarshal the Value to a string;
                                return stringValue;
                        }
```

```
        }
        return null;
    }
```

The delete method first converts the key parameter to a String using the method defined in KVMessage.java and utilizes this String to create a KVMessage of type get request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, throwing an exception if the response message is not null, i.e. is an error message.

```
public void del(K key) throws KVException {
    set strKey to marshalled value of key;
    set request to new KVMessage("delreq", strKey, null);
    set s to new Socket(server,port);
    connect s to the server's address for 10000000 milliseconds (timeout);
    set out to new ObjectOutputStream(s.getOutputStream());
    convert KVMessage to XML;
    write the request Object XML to the OutputStream out;
    set response to new KVMessage(s.getInputStream());
    if (response type is "resp" and response message is not "Success")
            throw new KVException(response);
}
```

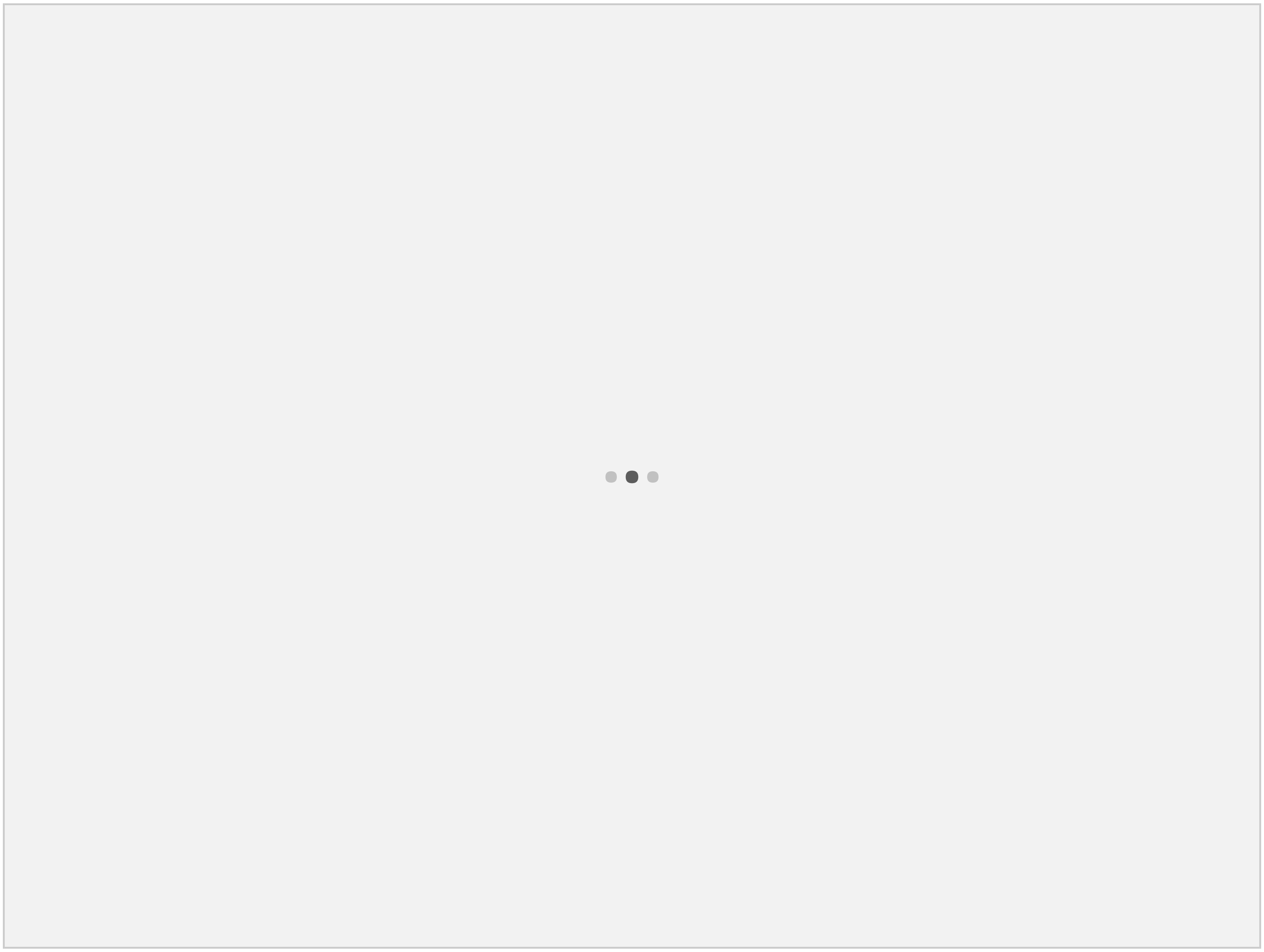
Testing Plan:

1. Create dummy messages of types putreq, getreq, delreq, and the various responses and test that they correctly convert to properly formatted XML.
2. Create an InputStream and test that it correctly extracts information or throws an exception if given malformed XML.
3. Convert an Object of a certain type that is not String to a String.
4. Convert a String to an Object that is not of type String.
5. Issue and parse a get request.
6. Issue and parse a put request.
7. Issue and parse a delete request.

Part III. Implementing a ThreadPool

Overview:

The thread pool should accept different tasks and execute them asynchronously. The threadpool should maintain a queue of tasks submitted to it, and should assign it to a free thread as soon as it is available. Ensure that the addToQueue interface to the threadpool is non-blocking. The Key-Value server will use the threadpool to parallelize data storage into the dummy storage system provided (KVStore). The logic is shown in this diagram:



Correctness Constraints:
1. Execution of tasks is asynchronous.
2. Maintains a queue of tasks.
3. Tasks are assigned to free threads as soon as they are available
4. Add to queue must be non-blocking.

Declarations:
<u>In ThreadPool.java</u>
Queue TaskQueue;
Lock taskQueueLock;
private Condition taskAdded;
private ReentrantLock conditionLock;

<u>In WorkerThread (subclass in ThreadPool.java)</u>
Queue ThreadPool;

Descriptions:
The ThreadPool class will have a queue of tasks and a queue of threads. The TaskQueue will contain all tasks waiting to be run. The ThreadQueue contains all threads instantiated for this ThreadPool. The

ThreadQueue contains WorkerThreads that was spawned upon initialization. The WorkerThreads and the ThreadPool reference each other in a mutual reflexive scheme. Each WorkerThreads knows the ThreadPool it belongs to; Each ThreadPool knows all the WorkerThreads within it. This is a many:one relationship, meaning there are many WorkerThreads to a ThreadPool, one ThreadPool for many WorkerThreads. Each WorkerThread polls the TaskQueue within the ThreadPool continuously to acquire tasks and run them. When a WorkerThread finishes with one task, it immediately runs waiting tasks on the TaskQueue. If there are no more tasks, the WorkerThreads waits on the TaskQueue and sleeps. When a task is added to the TaskQueue, it will call wake on the WorkerThreads.

```
In ThreadPool.java:
create TaskQueue
conditionLock = new ReentrantLock();
taskAdded =  conditionLock.newCondition();

public ThreadPool(size)
create ThreadQueue of given size;
for (each element in ThreadQueue)
        create WorkerThread for each array element;
        workerthread.run();

public void addToQueue(task):
    enqueue task to TaskQueue
    lock the conditionLock;
    signal waiting threads on cond var taskAdded;
    unlock the conditionLock;

In WorkerThread
declare local ThreadPool reference
WorkerThread(ThreadPool o)
    this.ThreadPool = o

public void run()
    infiniteLoop:
        acquire TaskQueue lock
        task = get task from ThreadPool.TaskQueue
        release TaskQueue lock;
        if (task != null)
            task.run();
        lock the conditionLock;
ask thread to wait for new task to be added uninterruptibly on cond var taskAdded;
unlock conditionLock;
```

Testing Plan:

1. Create ThreadPool
2. Add tasks < number of threadPool threads
3. Add tasks == number of threadPool threads
4. Add tasks > number of threadPool threads

5. Check the idle/active status of threads for corresponding tasks.

Part IV. Implement LRU Cache
Overview:
Implement a fully-associative LRU cache and its methods: get, put, del, to access data and process requests more efficiently that are sent by threads in the threadpool. Synchronize access to the cache to keep data from being corrupted.

Correctness Constraints:
- The cache must be fully-associative.
- If the cache is full, remove the least recently used (LRU) entry from the cache.
- When you retrieve or add an entry to the cache, that entry must become the most recently used (MRU) entry in the cache.
- Deleting a key removes the entry associated with that key from the cache.
- The number of entries in the cache cannot exceed the maximum number of entries specified by *cacheSize*.

Declarations:
<u>In KVCache.java:</u>
- private final int cacheSize;

//modification: final; maintains maximum size of cache so as not to be corrupted
- private LinkedHashMap<K, V> cache;
- private HashMap<K, ReentrantLock> keyLocks; //maps locks to each key/entry in the cache
- private ReentrantLock Lock;

Descriptions:
<u>In KVCache.java</u>:
```
public KVCache(int cacheSize) throws KVException { //added throws modification
        try {   this.cacheSize = cacheSize passed in;
                cache = create new LinkedHashMap<K, V> with initial capacity of cacheSize, load factor
of 0.75, and ordering of the cache based on order of access (from LRU at the head to MRU at the tail);
} catch (IllegalArgumentException e) {    throw new KVException(null);    }
        keyLocks = new HashMap<K, ReentrantLock>();
        Lock = new ReentrantLock();
}
```

Get(key) retrieves an entry from the cache. That entry becomes MRU.
```
public V get(K key) {
V tmpValue = null;
        Lock.lock();
        try {   try {   lock(key); //lock the entry
                } catch (KVException e) {        e.getMsg();     }
        } finally {
                Lock.unlock();
        }
        //check if it is: 1) possible for cache to have keys, 2) cache has at least one entry, and 3) cache
contains key
```

```
        if (cacheSize > 0 && number of entries currently in the cache > 0 && cache contains key)
                tmpValue = get value stored at key in cache;
        unlock(key); //unlock the entry
        return tmpValue;
}
```

Put(key, value) adds a new entry if the key didn’t already exist as an entry in the cache, or overwrites the previous value stored at key with the new value. This new entry becomes MRU entry. If cache has reached full capacity, then replaces the LRU entry with new entry. Returns true if the value at key was overwritten, false otherwise.

```
public boolean put(K key, V value) {
boolean overwritten = false;
 //if key already exists in cache, this call to put will overwrite previous value
        Lock.lock();
        try {   try {   lock(key);      }
catch (KVException e) { e.getMsg();     }
        } finally {
        Lock.unlock();
}
//if cache is at full capacity, remove the LRU entry to allow addition of a new entry
        while (cacheSize > 0 && reachedMaxCapacity()){
Map.Entry<K, V> eldest = get the LRU entry stored at the head of the cache;
del(eldest.getKey());
        }
        //double checks that the cache has room for more entries
        if (cacheSize > 0 && number of entries currently in cache < cacheSize) {
                if (cache contains key)
                        overwritten = true; //key already exists in cache and is going to be overwritten
                put new entry into cache;
        }
        unlock(key);
        return overwritten;
}
```

Del(key) removes the entry with specified key from the cache.

```
public void del(K key) {
        Lock.lock();
        try {   try { lock(key);}
catch (KVException e) { e.getMsg();}
} finally { Lock.unlock(); }
if (cache contains key)
        remove key from cache;
unlock(key);
}
```

```
//reachedMaxCapacity: returns true if the cache has reached maximum capacity, else false
public boolean reachedMaxCapacity() {
        return (number of entries currently in cache >= cacheSize);
}
```

//lock: Locks only the entry that a request is trying to access by creating a *keyLock* for every entry that exists in the cache. If *keyLock* is being held by another thread, then wait on that *keyLock* (may be interrupted) and wait to be notified when the *keyLock* is unlocked. Else, lock the *keyLock* for specifed key.

```
private void lock(K key) throws KVException {
        ReentrantLock keyLock;
        if (keyLocks does not contain key)
                keyLock = create new ReentrantLock();
                put keyLock as value for key into keyLocks;
        else
                keyLock = get lock stored at key in keyLocks;
        while (keyLock is locked) {
                try { keyLock.wait();}
catch (InterruptedException e) {throw new KVException(null); }
}
lock keyLock;
}
```

//unlock: unlocks the *keyLock* for specified key only if the current thread trying to unlock it is actually holding the *keyLock*, else do nothing

```
private void unlock(K key) {
        ReentrantLock keyLock;
        if (keyLocks does contain key)
                keyLock = get lock stored at key in keyLocks;
                if (keyLock is held by the current thread)
                        notify any waiting threads waiting for keyLock;
                        unlock keyLock;
}
```

Testing Plan:

1. Create an instance of the cache. Create multiple threads to access the cache with a varied amount of different orderings of requests: get, put, del. Ensure correctness of data: correct retrievals, new additions, and removals. Make the cache full and check that the cache is correctly removing the LRU entry.
2. Intentionally try to break the code to check that exceptions are being properly handled.

Part V: Implementing KVServer

Overview:

Implement a Key-Value server which will handle requests by KVClients via a NetworkHandler. The NetworkHandler will create tasks which it will append to the Queue of tasks waiting on worker threads. As soon as a worker thread is available it will start the next queued runnable task, which will create a KVmessage from the socket input stream. The server processes the task and sends back a response to the Client that issued the request. Depending on whether the task was correctly completed the response may contain an error message; additionally, the keys and values are checked so that they do not exceed their respective upper bounds.

Correctness Constraints:

- Tasks should run asynchronously.
- Write through policy must be implemented. KVStore and KVCache should always be in agreement.
- The submitted key must be smaller than or equal to 256 B (unless the request is get).
- The submitted value must be smaller than or equal to 128 KB.
- Must send a response KVMessage after the server has processed a request.
- Throw KVExceptions appropriately.

Declarations:

In SocketServer:

Socket clientsocket;

In KeyServer:

```
        private Lock t;
```

Descriptions:

In KeyServer.java:

```
public boolean put(K key, V value) throws KVException {
        create a ByteArrayOutputStream called stream;
        create an ObjectOutputstream called objectstream from stream;
        write the key into the objectstream;
        convert the stream into a bytes array called bytes;
        create a ByteArrayOutputStream called valuestream;
        create an ObjectOutputstream called valueobjectstream from valuestream;
        write the key into the valueobjectstream;
        convert the stream into a bytes array called valuebytes;
        boolean temp;
        if(bytes.length > 256 || valuebytes.length > 1024 *128)
                throw new KVException;
        else{   try {   put key value into Cache;
                        acquire the lock;
                        put key value into KVStore and set temp to the returned value;
                        release the lock;       }
catch(Exception e)      {       release the lock;
                                        throw new KVException;          }
        }
return temp;
}
public V get (K key) throws KVException {
        try {   set x to the value associated with key in Cache;
                if(x != null)
                        return x;
                acquire the lock;
set x to the value associated with key in KVstore;
        release the lock;
        dataCache.put(key,x);   }
catch(Exception e) {    release the lock;
                        throw new KVException(null); }
return x;
}
public void del(K key) throws KVException {
                create a ByteArrayOutputStream called stream;
                create an ObjectOutputstream called objectstream from stream;
                write the key into the objectstream;
                convert the stream into a bytes array called bytes;
                if(bytes.length > 256)
                        throw new KVException;
                else{   try {   delete the key from Cache;
                                acquire the lock;
                                delete the key from KVstore;
                                release the lock;       }
                        catch(Exception e) {    release the lock;
                                                throw new KVException(null);    }
                }
        return false;
}
```

<u>In KVClientHandler</u>:

```
public void handle(Socket client) throws IOException {
        try{    Runnable task = new ProcessSocket(client);
                threadpool.addToQueue(task);  }
        catch(IOexception){     throw new KVException;          }
}

private class ProcessSocket implements Runnable {
        private Socket socket;
        public ProcessSocket(Socket c)          {
                socket = c;     }
        public void run()       {
                try{
                        KVMessage t = new KVMessage(socket.getInputStream());
                        KVMessage output;
                        set msgtype variable to the msgtype in the KVMessage t;
                        set key variable to the key in the KVMessage t;
                        set value variable to the value in the KVMessage t;
                        boolean status;
                        if(msgtype is "getreq")
                                set a temp variable to keyserver.get(key);
                                set the string value to marshall(temp);
                                set output to new KVMessage("resp", key,value);
                        else if(msgtype is "putreq")
                                set status to keyserver.put(key,value);
                                set output to new KVMessage("resp", null, null, status, "Success");
                        else if(msgtype is "delreq")
                                set output to new KVMessage("resp", null,null, null, "Success");
                        else
                                set output to new KVMessage("resp", null,null,null, "Error Message");
                        create a PrintWriter called out to new PrintWriter(socket.getOutputStream(), true);
                        print the XML message from output to the PrintWriter out;
// this is the “resp” pckt }
        catch(Exception e) // if anything goes wrong
        {       set output to new KVMessage("resp", null,null,null, "Error Message");
                set out to new PrintWriter(socket.getOutputStream(), true);
                print the XML message from output to out;  // “resp” pckt
                throw new KVException;          }
}
```

<u>In SocketServer.java:</u>

```
public void connect() throws IOException {
        try { set server to new ServerSocket(port);         }
        catch (IOException e) {  throw new KVException;          }
}

public void run() throws IOException {
        while(true) {
        try {   set clientSocket to null;
                try {   set clientSocket to server.accept(); }
                catch (IOException e) { throw new KVException;}
                handler.handle(clientSocket);
         } catch (IOException e) {  throw new KVException; }
  }

public void addHandler(NetworkHandler handler) {
        set this.handler to handler;
}
```

Testing Plan:

1. Start server and create 1 client. Test the functionality of put, del, and get by sending several messages to the server. Confirm that the server returns the correct response messages upon finishing requests.
2. Create multiple clients. Test that the server processes multiple requests asynchronously by checking the number of working threads at a given time.
3. Check that the server implements write through policy correctly by checking that updates cache

values are reflected in KVStore.
4. Check that the server does not process keys greater than 256B and values greater than 128 KB by passing large keys and values in KV messages.

Initial Design Document: Project 3

Group #6: Elise McCallum (cs162-bc), Danube Phan (cs162-aw), Thamine Dalichaouch (cs162-co), Vladimir Ponomarenko (cs162-bb), Jack Deng (cs162-fb)
GSI: Prashanth Mohan : Section 102

Part I. PingPong Program

Overview:

First an EC2 account needs to be set up as per the instructions given by the EC2 Access Guide. Then one must create a new EC2 instance, launch it, and SSH into it. To create the PingPong program, one must make use of TCP sockets that listen for connections on port 8081 and service each incoming request with the string ‘pong’. The java.net.Socket and java.net.ServerSocket classes are to be used for this.

Correctness Constraints:

· Each incoming TCP request on the socket must be serviced with the string ‘pong’.

Declarations:

No additional declarations in existing classes.

Descriptions:

The PingPong class has to be created.

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;

class PingPong {
        public static void main(String args[]) {
                create a new String called output and set it equal to ‘pong’;
                try {
                create a ServerSocket called srvr and bind it to port 8081;
                while (True) {
                create a Socket called skt and set it equal to srvr.accept();
                create a PrintWriter called out from the skt.getOutputStream();
                print output to out;
                close out;
                close skt;
        }
        catch (Exception e) {
        }
}
}
```

Testing Plan:

1. Create a client that opens a socket to port 8081 and prints out the response from the server. Run

the server, then run the client, and check if the output is equal to ‘pong’.

Part II. Implementing KVClient and KVMessage

Overview:

This part of the project deals with the implementation of KVClient and KVMessage. KVClient issues requests to the server to put, get, or delete key-value pairs and parses the responses it receives to return the appropriate values. The KVMessage class manages all the information needed to construct such requests, and can construct messages from given values or an XML input steam. KVMessage also handles the marshalling logic to convert an Object of an arbitrary type to a String, and to convert a String to an Object of an arbitrary type. The KVMessage class also converts requests to XML strings.

Correctness Constraints:

· If a KVMessage contains invalid XML, it should throw a KVException.
· Any malformed message should generate a KVException.
· A put request should insert a new key-value pair into the KVStore, and return a message of “Success” if successful (regardless if a value was overwritten or not).
· If a put request is unsuccessful, it should return an error message.
· A get request should return the unmarshalled value stored with that key.
· A delete request should throw an exception if the key to be deleted does not exist.

Declarations:

<u>In KVMessage.java</u>

```
private String status = null;
private String message = null;

//additional constructor added to accommodate requests using status and message variables
public KVMessage(String msgType, String key, String value, String status, String message) {
        set this msgType to msgType;
set this key to key;
set this status to status;
set this value to value;
set this message to message;
}
```

Descriptions:

<u>In KVMessage.java</u>

The following five methods are used to protect outside programs from manipulating the set values of message type, key, value, status, and message but still allow them to retrieve the stored values of said variables in a given KVMessage.

```
public String getType() {
        return this.msgType;
}
public String getKey() {
        return this.Key;
}
public String getValue() {
        return this.Value;
}
```

```
public String getStatus() {
        return this.status;
}
public String getMessage() {
        return this.message;
}
```

This method is provided to convert a String object to a general type by converting it first to an array of bytes, creating and object input stream from those bytes and reading the object created, which is now the object of the type specified in the input String.

```
public static Object convertToGeneralType(String info) throws KVException {
        try {   create a byte array of the input info using the char set UTF-8;
                create an object input stream using the data array of bytes as an input stream;
                read the object from the input stream;
                return the object read;     }
        catch (Exception e) { throw new KVException(null); }
}
```

This method is provided to convert a Serializable object to a String by first converting it to a byte array output stream and then writing the input information to the output stream created with those bytes. The stream is then closed and the String is created using the byte array and the char set UTF-8, and returned.

```
public static String convertToString(Serializable info) throws KVException {
        try {   create a new byte array output stream;
                construct a new object output stream using the byte array;
                write the input object info to the output stream;
                close the stream;
                create a string object using the byte array and the char set UTF-8;
                return that string;          }
        catch (Exception e) { throw new KVException(null); }
}
```

This is another method used for creating a KVMessage. Using an XML DOM Parser, this method creates a DOM tree for the possible information that could be stored in the message, parses the tree, and sets the appropriate values as they are found. The message type is then set using the same XML parser. An exception is thrown if any of these steps fail.

```
public KVMessage(InputStream input) throws KVException {
        try {
                set dbf to a new instance of DocumentBuilderFactory;
                set db to dbf.newDocumentBuilder();
                set the dom to db.parse(input);
                set docElement to the document element from the dom;
                create a String array called nodes consisting of {"Key", "Value", "Status", "Message"};
                create a String array called values of length 4;
                for each element in the array nodes {
                        create a NodeList for the elements matching the tag of the specific element;
                        if (elementNodes is not null and the length of the NodeList is 1)
                                create an element which is the first and only element in the node list;
```

```
                    set the corresponding value in the values array to be the node value;
            }
            set this key to the first element of values array;
            set this value to the second element of values array;
            set this status to the third element of values array;
            set this message to the fourth element of values array;
            create a NodeList of elements matching the KVMessage tag;
            if (node list is not null)
                create an element which is the first element in the node list;
                set this msgType to the value of the attribute type;
        }
        catch (Exception e) {
            throw new KVException(null);
        }
}
```

This method converts a KVMessage to XML using an XML DOM Parser to generate XML. It first creates an XML tree, and then attaches as the root node a KVMessage tag. The appropriate values of message information are added as children to the root node if they are present. The type is also set as an attribute for the KVMessage node. The tree is then transformed to a String and returned. Any malformed XML will trigger an exception.

```
public String toXML() throws KVException {
    if (this.msgType == null)
        return null;
    else {
        try {   set dbf to a new instance of DocumentBuilderFactory;
                set db to dbf.newDocumentBuilder();
                set dom to new Document;
                create the root element as an element with tag KVMessage;
                set the type attribute of root element to this msgType;
                append this root as a child to the dom;
                if (this key is not null) {
                    create an element called keyNode with tag Key;
                    set the text of the node to be the value of this.key;
                    append this node to the root;
                }
                if (this value is not null) {
                    create an element called keyNode with tag Value;
                    set the text of the node to be the value of this.value;
                    append this node to the root;
                }
                if (this status is not null) {
                    create an element called keyNode with tag Status;
                    set the text of the node to be the value of this.status;
                    append this node to the root;
                }
                if (this message is not null) {
                    create an element called keyNode with tag Message;
                    set the text of the node to be the value of this.message;
                    append this node to the root;
                }
                create a new instance of a transformerFactory;
                create a new transformer from that factory;
                create a StreamResult result using a new StringWriter;
                set the source to be a new DOMSource using the dom;
                transform data from source to result;
                convert the stringwriter to a string;
                return xml string;
            } catch (Exception e) {
                throw new KVException(null);
            }
        }
}
```

In KVClient.java

The put method first converts the key and value parameters to Strings using the method defined in KVMessage.java and utilizes these Strings to create a KVMessage of type put request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, returning true if the transaction is successful and false otherwise.

```
public boolean put(K key, V value) throws KVException {
    set strKey to marshalled value of key;
    set strValue to marshalled value of value;
    set request to be new KVMessage("putreq", strKey, strValue);
    set s to new Socket(server, port);
    connect s to the server's address for 10000000 milliseconds (timeout);
    set out to new ObjectOutputStream(s.getOutputStream());
    convert the KVMessage to XML;
    write the request Object XML to the OutputStream out;
    set response to new KVMessage(s.getInputStream());
    if (response message type is "resp") {
        if (response message is "Success")
            return true;
        else if (response message is not "Success") {
            throw new KVException(response);
        }
    }
    return false;
}
```

The get method first converts the key parameter to a String using the method defined in KVMessage.java and utilizes this String to create a KVMessage of type get request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, returning true if the unmarshalled value, if one exists, and null otherwise.

```
public V get(K key) throws KVException {
        set strKey to marshalled value of key;
        set request to new KVMessage("getreq", strKey, null);
        set s to new Socket(server, port);
        connect s to the server’s address for 10000000 milliseconds (timeout);
        set out to new ObjectOutputStream(s.getOutputStream());
        convert the KVMessage to XML;
        write the request Object XML to the OutputStream out;
        set response to new KVMessage(s.getInputStream());
        if (response message type is “resp”) {
            if (response message is not null)
                throw new KVException(response);
            else {
                set value to Value of response;
                unmarshal the Value to a string;
                return stringValue;
            }
        }
        return null;
    }
```

The delete method first converts the key parameter to a String using the method defined in KVMessage.java and utilizes this String to create a KVMessage of type get request. This message is converted to XML and sent through a socket used to establish a connection to the server at the specified port. The response is recorded using an InputStream, and parsed as appropriate, throwing an exception if the response message is not null, i.e. is an error message.

```
public void del(K key) throws KVException {
    set strKey to marshalled value of key;
    set request to new KVMessage("delreq", strKey, null);
    set s to new Socket(server,port);
    connect s to the server’s address for 10000000 milliseconds (timeout);
    set out to new ObjectOutputStream(s.getOutputStream());
    convert KVMessage to XML;
    write the request Object XML to the OutputStream out;
    set response to new KVMessage(s.getInputStream());
    if (response type is “resp” and response message is not “Success”)
            throw new KVException(response);
}
```

Testing Plan:

1. Create dummy messages of types putreq, getreq, delreq, and the various responses and test that they correctly convert to properly formatted XML.
2. Create an InputStream and test that it correctly extracts information or throws an exception if given malformed XML.
3. Convert an Object of a certain type that is not String to a String.
4. Convert a String to an Object that is not of type String.
5. Issue and parse a get request.
6. Issue and parse a put request.
7. Issue and parse a delete request.

Part III. Implementing a ThreadPool

Overview:
The thread pool should accept different tasks and execute them asynchronously. The threadpool should maintain a queue of tasks submitted to it, and should assign it to a free thread as soon as it is available. Ensure that the addToQueue interface to the threadpool is non-blocking. The Key-Value server will use the threadpool to parallelize data storage into the dummy storage system provided (KVStore). The logic is shown in this diagram:

Correctness Constraints:
1. Execution of tasks is asynchronous.
2. Maintains a queue of tasks.
3. Tasks are assigned to free threads as soon as they are available
4. Add to queue must be non-blocking.

Declarations:
<u>In ThreadPool.java</u>
Queue TaskQueue;
Lock taskQueueLock;
private Condition taskAdded;
private ReentrantLock conditionLock;

In WorkerThread (subclass in ThreadPool.java)
Queue ThreadPool;

Descriptions:
The ThreadPool class will have a queue of tasks and a queue of threads. The TaskQueue will contain all tasks waiting to be run. The ThreadQueue contains all threads instantiated for this ThreadPool. The ThreadQueue contains WorkerThreads that was spawned upon initialization. The WorkerThreads and the ThreadPool reference each other in a mutual reflexive scheme. Each WorkerThreads knows the ThreadPool it belongs to; Each ThreadPool knows all the WorkerThreads within it. This is a many:one relationship, meaning there are many WorkerThreads to a ThreadPool, one ThreadPool for many WorkerThreads. Each WorkerThread polls the TaskQueue within the ThreadPool continuously to acquire tasks and run them. When a WorkerThread finishes with one task, it immediately runs waiting tasks on the TaskQueue. If there are no more tasks, the WorkerThreads waits on the TaskQueue and sleeps. When a task is added to the TaskQueue, it will call wake on the WorkerThreads.

In ThreadPool.java:

```
create TaskQueue
conditionLock = new ReentrantLock();
taskAdded =  conditionLock.newCondition();

public ThreadPool(size)
create ThreadQueue of given size;
for (each element in ThreadQueue)
        create WorkerThread for each array element;
        workerthread.run();

public void addToQueue(task):
    enqueue task to TaskQueue
    lock the conditionLock;
    signal waiting threads on cond var taskAdded;
    unlock the conditionLock;
```

In WorkerThread

```
declare local ThreadPool reference
WorkerThread(ThreadPool o)
    this.ThreadPool = o

public void run()
    infiniteLoop:
        acquire TaskQueue lock
        task = get task from ThreadPool.TaskQueue
        release TaskQueue lock;
        if (task != null)
            task.run();
        lock the conditionLock;
ask thread to wait for new task to be added uninterruptibly on cond var taskAdded;
unlock conditionLock;
```

Testing Plan:

1. Create ThreadPool
2. Add tasks < number of threadPool threads
3. Add tasks == number of threadPool threads
4. Add tasks > number of threadPool threads
5. Check the idle/active status of threads for corresponding tasks.

Part IV. Implement LRU Cache

Overview:

Implement a fully-associative LRU cache and its methods: get, put, del, to access data and process requests more efficiently that are sent by threads in the threadpool. Synchronize access to the cache to keep data from being corrupted.

Correctness Constraints:

- The cache must be fully-associative.
- If the cache is full, remove the least recently used (LRU) entry from the cache.
- When you retrieve or add an entry to the cache, that entry must become the most recently used (MRU) entry in the cache.
- Deleting a key removes the entry associated with that key from the cache.
- The number of entries in the cache cannot exceed the maximum number of entries specified by *cacheSize*.

Declarations:

In KVCache.java:

- private final int cacheSize;

//modification: final; maintains maximum size of cache so as not to be corrupted

- private LinkedHashMap<K, V> cache;
- private HashMap<K, ReentrantLock> keyLocks; //maps locks to each key/entry in the cache
- private ReentrantLock Lock;

Descriptions:

In KVCache.java:

```
public KVCache(int cacheSize) throws KVException { //added throws modification
        try {   this.cacheSize = cacheSize passed in;
                cache = create new LinkedHashMap<K, V> with initial capacity of cacheSize, load factor
of 0.75, and ordering of the cache based on order of access (from LRU at the head to MRU at the tail);
} catch (IllegalArgumentException e) {    throw new KVException(null);   }
        keyLocks = new HashMap<K, ReentrantLock>();
        Lock = new ReentrantLock();
}
```

Get(key) retrieves an entry from the cache. That entry becomes MRU.

```
public V get(K key) {
V tmpValue = null;
        Lock.lock();
        try {   try {   lock(key); //lock the entry
                } catch (KVException e) {       e.getMsg();     }
        } finally {
                Lock.unlock();
        }
        //check if it is: 1) possible for cache to have keys, 2) cache has at least one entry, and 3) cache
contains key
        if (cacheSize > 0 && number of entries currently in the cache > 0 && cache contains key)
                tmpValue = get value stored at key in cache;
        unlock(key); //unlock the entry
        return tmpValue;
}
```

Put(key, value) adds a new entry if the key didn’t already exist as an entry in the cache, or overwrites the previous value stored at key with the new value. This new entry becomes MRU entry. If cache has reached full capacity, then replaces the LRU entry with new entry. Returns true if the value at key was overwritten, false otherwise.

```
public boolean put(K key, V value) {
boolean overwritten = false;
 //if key already exists in cache, this call to put will overwrite previous value
        Lock.lock();
        try {       try {       lock(key);           }
catch (KVException e) { e.getMsg();        }
        } finally {
        Lock.unlock();
}
//if cache is at full capacity, remove the LRU entry to allow addition of a new entry
        while (cacheSize > 0 && reachedMaxCapacity()){
Map.Entry<K, V> eldest = get the LRU entry stored at the head of the cache;
del(eldest.getKey());
        }
        //double checks that the cache has room for more entries
        if (cacheSize > 0 && number of entries currently in cache < cacheSize) {
                if (cache contains key)
                        overwritten = true; //key already exists in cache and is going to be overwritten
                put new entry into cache;
        }
        unlock(key);
        return overwritten;
}
```

Del(key) removes the entry with specified key from the cache.

```
public void del(K key) {
        Lock.lock();
        try {       try { lock(key);}
catch (KVException e) { e.getMsg();}
} finally { Lock.unlock(); }
if (cache contains key)
        remove key from cache;
unlock(key);
}
//reachedMaxCapacity: returns true if the cache has reached maximum capacity, else false
public boolean reachedMaxCapacity() {
        return (number of entries currently in cache >= cacheSize);
}
```

//lock: Locks only the entry that a request is trying to access by creating a *keyLock* for every entry that exists in the cache. If *keyLock* is being held by another thread, then wait on that *keyLock* (may be interrupted) and wait to be notified when the *keyLock* is unlocked. Else, lock the *keyLock* for specifed key.

```
private void lock(K key) throws KVException {
        ReentrantLock keyLock;
        if (keyLocks does not contain key)
                keyLock = create new ReentrantLock();
                put keyLock as value for key into keyLocks;
        else
                keyLock = get lock stored at key in keyLocks;
        while (keyLock is locked) {
                try { keyLock.wait();}
catch (InterruptedException e) {throw new KVException(null); }
}
lock keyLock;
}
```

//unlock: unlocks the *keyLock* for specified key only if the current thread trying to unlock it is actually holding the *keyLock*, else do nothing

```
private void unlock(K key) {
        ReentrantLock keyLock;
        if (keyLocks does contain key)
                keyLock = get lock stored at key in keyLocks;
                if (keyLock is held by the current thread)
                        notify any waiting threads waiting for keyLock;
                        unlock keyLock;
}
```

Testing Plan:

1. Create an instance of the cache. Create multiple threads to access the cache with a varied amount of different orderings of requests: get, put, del. Ensure correctness of data: correct retrievals, new additions, and removals. Make the cache full and check that the cache is correctly removing the LRU entry.
2. Intentionally try to break the code to check that exceptions are being properly handled.

Part V: Implementing KVServer

Overview:

Implement a Key-Value server which will handle requests by KVClients via a NetworkHandler. The

NetworkHandler will create tasks which it will append to the Queue of tasks waiting on worker threads. As soon as a worker thread is available it will start the next queued runnable task, which will create a KVmessage from the socket input stream. The server processes the task and sends back a response to the Client that issued the request. Depending on whether the task was correctly completed the response may contain an error message; additionally, the keys and values are checked so that they do not exceed their respective upper bounds.

Correctness Constraints:

- Tasks should run asynchronously.
- Write through policy must be implemented. KVStore and KVCache should always be in agreement.
- The submitted key must be smaller than or equal to 256 B (unless the request is get).
- The submitted value must be smaller than or equal to 128 KB.
- Must send a response KVMessage after the server has processed a request.
- Throw KVExceptions appropriately.

Declarations:

<u>In SocketServer:</u>

Socket clientsocket;

<u>In KeyServer:</u>

```
    private Lock t;
```

Descriptions:

<u>In KeyServer.java:</u>

```
public boolean put(K key, V value) throws KVException {
        create a ByteArrayOutputStream called stream;
        create an ObjectOutputstream called objectstream from stream;
        write the key into the objectstream;
        convert the stream into a bytes array called bytes;
        create a ByteArrayOutputStream called valuestream;
        create an ObjectOutputstream called valueobjectstream from valuestream;
        write the key into the valueobjectstream;
        convert the stream into a bytes array called valuebytes;
        boolean temp;
        if(bytes.length > 256 || valuebytes.length > 1024 *128)
                throw new KVException;
        else{   try {   put key value into Cache;
                        acquire the lock;
                        put key value into KVStore and set temp to the returned value;
                        release the lock;           }
catch(Exception e)      {       release the lock;
                                        throw new KVException;          }
        }
return temp;
}
public V get (K key) throws KVException {
        try {   set x to the value associated with key in Cache;
```

```
                if(x != null)
                        return x;
                acquire the lock;
set x to the value associated with key in KVstore;
        release the lock;
        dataCache.put(key,x);   }
catch(Exception e) {    release the lock;
                        throw new KVException(null); }
return x;
}
public void del(K key) throws KVException {
                create a ByteArrayOutputStream called stream;
                create an ObjectOutputstream called objectstream from stream;
                write the key into the objectstream;
                convert the stream into a bytes array called bytes;
                if(bytes.length > 256)
                        throw new KVException;
                else{   try {   delete the key from Cache;
                                acquire the lock;
                                delete the key from KVstore;
                                release the lock;       }
                        catch(Exception e) {    release the lock;
                                                throw new KVException(null);    }
                }
        return false;
}
```

<u>In KVClientHandler</u>:

```
public void handle(Socket client) throws IOException {
        try{    Runnable task = new ProcessSocket(client);
                threadpool.addToQueue(task);  }
        catch(IOexception){     throw new KVException;          }
}

private class ProcessSocket implements Runnable {
        private Socket socket;
        public ProcessSocket(Socket c)          {
                socket = c;     }
        public void run()       {
                try{
                        KVMessage t = new KVMessage(socket.getInputStream());
                        KVMessage output;
                        set msgtype variable to the msgtype in the KVMessage t;
```

```
                    set key variable to the key in the KVMessage t;
                    set value variable to the value in the KVMessage t;
                    boolean status;
                    if(msgtype is "getreq")
                        set a temp variable to keyserver.get(key);
                        set the string value to marshall(temp);
                        set output to new KVMessage("resp", key,value);
                    else if(msgtype is "putreq")
                        set status to keyserver.put(key,value);
                        set output to new KVMessage("resp", null, null, status, "Success");
                    else if(msgtype is "delreq")
                        set output to new KVMessage("resp", null,null, null, "Success");
                    else
                        set output to new KVMessage("resp", null,null,null, "Error Message");
                    create a PrintWriter called out to new PrintWriter(socket.getOutputStream(), true);
                    print the XML message from output to the PrintWriter out;
// this is the "resp" pckt }
    catch(Exception e) // if anything goes wrong
    {       set output to new KVMessage("resp", null,null,null, "Error Message");
            set out to new PrintWriter(socket.getOutputStream(), true);
            print the XML message from output to out;  // "resp" pckt
            throw new KVException;          }
}
```

<u>In SocketServer.java:</u>

```
public void connect() throws IOException {
            try { set server to new ServerSocket(port);         }
            catch (IOException e) {  throw new KVException;           }
}

public void run() throws IOException {
            while(true) {
            try {   set clientSocket to null;
                    try {   set clientSocket to server.accept(); }
                    catch (IOException e) { throw new KVException;}
                    handler.handle(clientSocket);
             } catch (IOException e) {  throw new KVException; }
  }

public void addHandler(NetworkHandler handler) {
            set this.handler to handler;
}
```

Testing Plan:

5. Start server and create 1 client. Test the functionality of put, del, and get by sending several

messages to the server. Confirm that the server returns the correct response messages upon finishing requests.

6. Create multiple clients. Test that the server processes multiple requests asynchronously by checking the number of working threads at a given time.
7. Check that the server implements write through policy correctly by checking that updates cache values are reflected in KVStore.
8. Check that the server does not process keys greater than 256B and values greater than 128 KB by passing large keys and values in KV messages.

[diagram isn't copying from my version. soz babiz]

Correctness Constraints:

1. Execution of tasks is asynchronous.
2. Maintains a queue of tasks.
3. Tasks are assigned to free threads as soon as they are available
4. Add to queue must be non-blocking.

Declarations:

<u>In ThreadPool.java</u>

```
Queue TaskQueue;
Lock taskQueueLock;
private Condition taskAdded;
private ReentrantLock conditionLock;
```

<u>In WorkerThread (subclass in ThreadPool.java)</u>

```
Queue ThreadPool;
```

Descriptions:

The ThreadPool class will have a queue of tasks and a queue of threads. The TaskQueue will contain all tasks waiting to be run. The ThreadQueue contains all threads instantiated for this ThreadPool. The ThreadQueue contains WorkerThreads that was spawned upon initialization. The WorkerThreads and the ThreadPool reference each other in a mutual reflexive scheme. Each WorkerThreads knows the ThreadPool it belongs to; Each ThreadPool knows all the WorkerThreads within it. This is a many:one relationship, meaning there are many WorkerThreads to a ThreadPool, one ThreadPool for many WorkerThreads. Each WorkerThread polls the TaskQueue within the ThreadPool continuously to acquire tasks and run them. When a WorkerThread finishes with one task, it immediately runs waiting tasks on the TaskQueue. If there are no more tasks, the WorkerThreads waits on the TaskQueue and sleeps. When a task is added to the TaskQueue, it will call wake on the WorkerThreads.

<u>In ThreadPool.java:</u>

```
create TaskQueue
conditionLock = new ReentrantLock();
taskAdded =  conditionLock.newCondition();

public ThreadPool(size)
create ThreadQueue of given size;
for (each element in ThreadQueue)
          create WorkerThread for each array element;
          workerthread.run();
```

```
public void addToQueue(task):
    enqueue task to TaskQueue
    lock the conditionLock;
    signal waiting threads on cond var taskAdded;
    unlock the conditionLock;
```

<u>In WorkerThread</u>

```
declare local ThreadPool reference
WorkerThread(ThreadPool o)
    this.ThreadPool = o

public void run()
    infiniteLoop:
        acquire TaskQueue lock
        task = get task from ThreadPool.TaskQueue
        release TaskQueue lock;
        if (task != null)
            task.run();
        lock the conditionLock;
ask thread to wait for new task to be added uninterruptibly on cond var taskAdded;
unlock conditionLock;
```

Testing Plan:

1. Create ThreadPool
2. Add tasks < number of threadPool threads
3. Add tasks == number of threadPool threads
4. Add tasks > number of threadPool threads
5. Check the idle/active status of threads for corresponding tasks.

Part IV. Implement LRU Cache

Overview:

Implement a fully-associative LRU cache and its methods: get, put, del, to access data and process requests more efficiently that are sent by threads in the threadpool. Synchronize access to the cache to keep data from being corrupted.

Correctness Constraints:

- The cache must be fully-associative.
- If the cache is full, remove the least recently used (LRU) entry from the cache.
- When you retrieve or add an entry to the cache, that entry must become the most recently used (MRU) entry in the cache.
- Deleting a key removes the entry associated with that key from the cache.
- The number of entries in the cache cannot exceed the maximum number of entries specified by *cacheSize*.

Declarations:

<u>In KVCache.java:</u>

- private final int cacheSize;

//modification: final; maintains maximum size of cache so as not to be corrupted

- private LinkedHashMap<K, V> cache;
- private HashMap<K, ReentrantLock> keyLocks; //maps locks to each key/entry in the cache
- private ReentrantLock Lock;

Descriptions:

In KVCache.java:

```
public KVCache(int cacheSize) throws KVException { //added throws modification
        try {   this.cacheSize = cacheSize passed in;
                cache = create new LinkedHashMap<K, V> with initial capacity of cacheSize, load factor
of 0.75, and ordering of the cache based on order of access (from LRU at the head to MRU at the tail);
} catch (IllegalArgumentException e) {   throw new KVException(null);   }
        keyLocks = new HashMap<K, ReentrantLock>();
        Lock = new ReentrantLock();
}
```

Get(key) retrieves an entry from the cache. That entry becomes MRU.

```
public V get(K key) {
V tmpValue = null;
        Lock.lock();
        try {   try {   lock(key); //lock the entry
                } catch (KVException e) {       e.getMsg();     }
        } finally {
                Lock.unlock();
        }
        //check if it is: 1) possible for cache to have keys, 2) cache has at least one entry, and 3) cache
contains key
        if (cacheSize > 0 && number of entries currently in the cache > 0 && cache contains key)
                tmpValue = get value stored at key in cache;
        unlock(key); //unlock the entry
        return tmpValue;
}
```

Put(key, value) adds a new entry if the key didn't already exist as an entry in the cache, or overwrites the previous value stored at key with the new value. This new entry becomes MRU entry. If cache has reached full capacity, then replaces the LRU entry with new entry. Returns true if the value at key was overwritten, false otherwise.

```
public boolean put(K key, V value) {
boolean overwritten = false;
 //if key already exists in cache, this call to put will overwrite previous value
        Lock.lock();
        try {   try {   lock(key);      }
catch (KVException e) { e.getMsg();     }
        } finally {
        Lock.unlock();
}
//if cache is at full capacity, remove the LRU entry to allow addition of a new entry
        while (cacheSize > 0 && reachedMaxCapacity()){
Map.Entry<K, V> eldest = get the LRU entry stored at the head of the cache;
del(eldest.getKey());
        }
        //double checks that the cache has room for more entries
        if (cacheSize > 0 && number of entries currently in cache < cacheSize) {
            if (cache contains key)
                overwritten = true; //key already exists in cache and is going to be overwritten
            put new entry into cache;
        }
        unlock(key);
        return overwritten;
}
```

Del(key) removes the entry with specified key from the cache.

```
public void del(K key) {
    Lock.lock();
    try {    try { lock(key);}
catch (KVException e) { e.getMsg();}
} finally { Lock.unlock(); }
if (cache contains key)
    remove key from cache;
unlock(key);
}
//reachedMaxCapacity: returns true if the cache has reached maximum capacity, else false
public boolean reachedMaxCapacity() {
    return (number of entries currently in cache >= cacheSize);
}
```

//lock: Locks only the entry that a request is trying to access by creating a *keyLock* for every entry that exists in the cache. If *keyLock* is being held by another thread, then wait on that *keyLock* (may be interrupted) and wait to be notified when the *keyLock* is unlocked. Else, lock the *keyLock* for specifed key.

```
private void lock(K key) throws KVException {
    ReentrantLock keyLock;
    if (keyLocks does not contain key)
        keyLock = create new ReentrantLock();
        put keyLock as value for key into keyLocks;
    else
        keyLock = get lock stored at key in keyLocks;
    while (keyLock is locked) {
        try { keyLock.wait();}
catch (InterruptedException e) {throw new KVException(null); }
}
lock keyLock;
}
```

```
//unlock: unlocks the keyLock for specified key only if the current thread trying to unlock it is actually holding the keyLock, else do nothing
private void unlock(K key) {
    ReentrantLock keyLock;
    if (keyLocks does contain key)
        keyLock = get lock stored at key in keyLocks;
        if (keyLock is held by the current thread)
            notify any waiting threads waiting for keyLock;
            unlock keyLock;
}
```

Testing Plan:

1. Create an instance of the cache. Create multiple threads to access the cache with a varied amount of different orderings of requests: get, put, del. Ensure correctness of data: correct retrievals, new additions, and removals. Make the cache full and check that the cache is correctly removing the LRU entry.
2. Intentionally try to break the code to check that exceptions are being properly handled.

Part V: Implementing KVServer

Overview:

Implement a Key-Value server which will handle requests by KVClients via a NetworkHandler. The NetworkHandler will create tasks which it will append to the Queue of tasks waiting on worker threads. As soon as a worker thread is available it will start the next queued runnable task, which will create a KVmessage from the socket input stream. The server processes the task and sends back a response to the Client that issued the request. Depending on whether the task was correctly completed the response may contain an error message; additionally, the keys and values are checked so that they do not exceed their respective upper bounds.

Correctness Constraints:

- Tasks should run asynchronously.
- Write through policy must be implemented. KVStore and KVCache should always be in agreement.
- The submitted key must be smaller than or equal to 256 B (unless the request is get).
- The submitted value must be smaller than or equal to 128 KB.
- Must send a response KVMessage after the server has processed a request.
- Throw KVExceptions appropriately.

Declarations:

In SocketServer:

```
Socket clientsocket;
```

In KeyServer:

```
    private Lock t;
```

Descriptions:

In KeyServer.java:

```
public boolean put(K key, V value) throws KVException {
    create a ByteArrayOutputStream called stream;
    create an ObjectOutputstream called objectstream from stream;
    write the key into the objectstream;
```

```
          convert the stream into a bytes array called bytes;
          create a ByteArrayOutputStream called valuestream;
          create an ObjectOutputstream called valueobjectstream from valuestream;
          write the key into the valueobjectstream;
          convert the stream into a bytes array called valuebytes;
          boolean temp;
          if(bytes.length > 256 || valuebytes.length > 1024 *128)
                    throw new KVException;
          else{     try {     put key value into Cache;
                              acquire the lock;
                              put key value into KVStore and set temp to the returned value;
                              release the lock;          }
catch(Exception e)            {         release the lock;
                                                  throw new KVException;          }
          }
return temp;
}
public V get (K key) throws KVException {
          try {     set x to the value associated with key in Cache;
                    if(x != null)
                              return x;
                    acquire the lock;
set x to the value associated with key in KVstore;
          release the lock;
          dataCache.put(key,x);   }
catch(Exception e) {          release the lock;
                              throw new KVException(null); }
return x;
}
public void del(K key) throws KVException {
                    create a ByteArrayOutputStream called stream;
                    create an ObjectOutputstream called objectstream from stream;
                    write the key into the objectstream;
                    convert the stream into a bytes array called bytes;
                    if(bytes.length > 256)
                              throw new KVException;
                    else{     try {     delete the key from Cache;
                                        acquire the lock;
                                        delete the key from KVstore;
                                        release the lock;          }
                              catch(Exception e) {     release the lock;
                                                       throw new KVException(null);     }
                    }
            return false;
    }
```
<u>In KVClientHandler</u>:
```
    public void handle(Socket client) throws IOException {
            try{        Runnable task = new ProcessSocket(client);
                        threadpool.addToQueue(task);   }
            catch(IOexception){         throw new KVException;              }
    }

    private class ProcessSocket implements Runnable {
            private Socket socket;
            public ProcessSocket(Socket c)              {
                    socket = c;         }
            public void run()               {
                    try{
                            KVMessage t = new KVMessage(socket.getInputStream());
                            KVMessage output;
                            set msgtype variable to the msgtype in the KVMessage t;
                            set key variable to the key in the KVMessage t;
                            set value variable to the value in the KVMessage t;
                            boolean status;
                            if(msgtype is "getreq")
                                    set a temp variable to keyserver.get(key);
                                    set the string value to marshall(temp);
                                    set output to new KVMessage("resp", key,value);
                            else if(msgtype is "putreq")
                                    set status to keyserver.put(key,value);
                                    set output to new KVMessage("resp", null, null, status, "Success");
                            else if(msgtype is "delreq")
                                    set output to new KVMessage("resp", null,null, null, "Success");
                            else
                                    set output to new KVMessage("resp", null,null,null, "Error Message");
                            create a PrintWriter called out to new PrintWriter(socket.getOutputStream(), true);
                            print the XML message from output to the PrintWriter out;
    // this is the “resp” pckt }
            catch(Exception e) // if anything goes wrong
            {       set output to new KVMessage("resp", null,null,null, "Error Message");
                    set out to new PrintWriter(socket.getOutputStream(), true);
                    print the XML message from output to out;  // “resp” pckt
                    throw new KVException;              }
    }
```
<u>In SocketServer.java:</u>
```
    public void connect() throws IOException {
            try { set server to new ServerSocket(port);       }
            catch (IOException e) {  throw new KVException;       }
}

public void run() throws IOException {
            while(true) {
            try {    set clientSocket to null;
                  try {    set clientSocket to server.accept(); }
                  catch (IOException e) { throw new KVException;}
                  handler.handle(clientSocket);
             } catch (IOException e) {  throw new KVException; }
  }

public void addHandler(NetworkHandler handler) {
            set this.handler to handler;
}
```

Testing Plan:

9. Start server and create 1 client. Test the functionality of put, del, and get by sending several messages to the server. Confirm that the server returns the correct response messages upon finishing requests.
10. Create multiple clients. Test that the server processes multiple requests asynchronously by checking the number of working threads at a given time.
11. Check that the server implements write through policy correctly by checking that updates cache values are reflected in KVStore.
12. Check that the server does not process keys greater than 256B and values greater than 128 KB by passing large keys and values in KV messages.