Project #4 Distributed Key-Value Store

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

Section 106, TA: Kevin Klues

Task 1: Two Phase Commit Messages (KVMessage) and KVClient.ignoreNext

Correctness Constraints

- KVMessage class should be modified, and ignoreNext() method in KVClient class should be implemented so that TPC messages are processed properly.

Declarations

There are no declarations necessary for this section.

Description

One of the first modifications we will have to make is to edit our *validMsgType()* helper function which we'll have to add the new message types of "ready", "abort", "commit" and "ack" as well as "register". For testing purposes we will also have to make "ignoreNext" a valid type.

After that modification most of our constructors will remain the same except the Socket constructor. The Socket constructor will also have to read our the TPCOpId field from the XML into the message. Also, we need to create a constructor which takes a socket and can timeout, for this one we just call Socket.*setSoTimeout()* with the timeout before we try to read the socket. Then we read the information from the socket and in the event of a timeout we catch the SocketTimeoutException and rethrow it as a KVException. The reading of the information should be identical to that of the other socket constructor so that logic can be extracted to a helper method, *readFromSocket* or something similar.

In our *toXML()* method, we can also reuse our method from part 3, but we will need to add a TPCOpId field to put and del messages in the case that it is non-null (it can be null if the request is coming from a client). We also need to add special handlers for our other TPC operations. This means an "if" case for "ack", "ready", and "commit" where we only add a TCPOpId tag, and an "if" case for "abort" where we include an error message if the message field is present and the TCPOpId. For registration we also need to support the "register" tag, which will include the "Message" tag. Finally we also need to have the case for the "ignoreNext" where we do not do anything but we do not throw an exception either.

We also need to implement KVClient.ignoreNext() which just sends an ignore message to the specified server. This is rather simple and can be done as follows:

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

Testing Plan

Our tests for the previous project should still hold true to this part. This includes testing incorrectly formatted messages to ensure exceptions, testing correctly formatted messages of various types, and testing messages with insufficient information to ensure exceptions. On top of this we can do the following:

- Create a correctly formatted message such as "putreq" without a TPCOpId and ensure that the XML doesn't include the corresponding tag.
- Also create a correctly formatted message with the op ID and ensure that the XML does include the tag.

Task 2: TPCLog

Correctness Constraints

- The interruptedTPCOperation field must be set on a rebuild if the server was in the middle of a Phase 1 TPC operation when it died.
- The KVServer must be build correctly if we are restoring from a failed node log.

Declarations

There are no necessary declarations for this section.

Description

We must implement the *appendAndFlush()* and *rebuildKeyServer()* in the TPCLog class.

For append and flush, the important thing to note is that this is used for rebuilding the KVServer after a failure which means that the only things that we should be saving to it are the put and del operations and any other 2PC operations (ie. only the ones that can actually change the state of the KVServer). Thus our method looks like this:

```
appendAndFlush(KVMessage entry):
    if(entry.getMsgType() is "putreq" "delreq" or "ready" or "abort" or "commit")
        entries.add(entry);
        flushToDisk();
    else
        // Do nothing
```

Now when it comes to rebuilding the server we want to read in all the entries from disk and if the last entry was a Phase 1 message, we need to set that as the interrupted TPC operation. Otherwise we don't need to set it because the master will continue resending messages until it gets a response. Our code looks like the following:

```
rebuildKeyServer();
    loadFromDisk();
    prev = null
    for(KVMessage entry : entries)
        if entry.type is "commit" and prev != null
            if prev.type is "putreq"
                kvServer.put(prev.key, prev.value)
            else if prev.type is "delreq"
                kvServer.del(prev.key)
        else if entry.type is "putreq" or "delreq"
            prev = entry
    interruptedTpcOp = entries.get(entries.size() - 1)
```

Testing Plan

In order to test our TPCLog, what we did is created a TPCHandler attached to a given server, and then a TPCLog attached to a different server, and set the TPCLog for the handler to be the one we just created. Then what we did was performed a complete two phase commit on our handler, putting and commiting a value into the KVServer. We also performed a half complete one, only submitting the put request. After this we called rebuildKeyServer on the log, which in turn performed the TPC operations on our second KVServer. We then asserted that only the completed operations made it into the key server, in this case the first put, and that the other putreq was available as an interrupted operation.

Task 3: Registration Logic (Registration Handler), SlaveInfo, findFirstReplica and findSuccessor

Correctness Constraints

- If a server dies it will register with the same node ID and might register under a new port, this case should be handled.
- After the slave is registered you should send a confirmation message noting the success to the slave.

Declarations

We declare the following:

- ArrayList¡SlaveInfo¿ *keySpace* - a list which contains all the nodes which divide our keyspace.

Description

For this section we must implement the RegistrationHandler *handle()* and *run()* methods as well as the SlaveInfo constructor, *connectHost()* and *closeHost()* methods. We also need to implement the *findFirstReplica()* and *findSuccessor()* methods in TPCMaster.

The TPCRegistrationHandler *handle()* method is not too complicated, all we need to do to handler requests for registration is create a new RegistrationHandler runnable passing the socket provided to the handle method into its constructor and then add it to the threadpool in a similar fashion as done in KVClientHandler in the previous project. On any exceptions, we send them to the client.

On the other hand, the RegistrationHandler *run()* method is a little more complicated. We want to read in the registration information from the socket in the form of a KVMessage, then create a new SlaveInfo object, add it to our map of registered slaves and then return a success message to the user.

```
run():
    try:
        KVMessage regMsg = new KVMessage(client);
        if("register".equals(regMsg.getMsgType())):
            SlaveInfo info = new SlaveInfo(regMsg.getMessage());
            // Search through our list to ensure it isn't there.
            for(int i = 0; i < keySpace.size(); i++)
                if(keySpace.get(i).getSlaveID() == info.getSlaveID()):
                    keySpace.set(i, info);
                    new KVMessage("resp", "Success").sendMessage(client);
                    return;
            // If not, add a new node into the space where it belongs
            for(int i = 0; i < keySpace.size(); i++)
                if(isLessThanUnsigned(info.getSlaveID(), keySpace.get(i).getSlaveID()))
                    keySpace.add(i, info);
                    return;
```

```
            // If we didn't add it, add it to the end
            keySpace.add(info);
            new KVMessage("resp", "Success").sendMessage(client);
        else:
            new KVMessage("resp", "Error invalid message to register server").sendMessage(client);
    catch (KVException e)
        e.getMsg().sendMessage(client);
```

One important note is that we will have to protect accesses to the *keySpace* array to ensure that it keeps correct state since we will be accessing during registrations and operations.

The SlaveInfo class is a little easier. In the constructor we just want to parse the registration string which can be done simply by spliting at the '@' and ':' characters and then forming our SlaveInfo from there. In the *connectHost()* and *closeHost()* methods we implement them just as we did with KVClient where the connectHost method creates a new Socket with the Slave's host and port and returns it and closeHost takes a provided Socket, generated from the connectHost() and safely closes it. We will also make sure that in our *connectHost()* method we add a timeout to our socket so that connections to our slaves will timeout if they fail and we do not indefinitely wait.

Now that our slave servers have been registered and are sorted in order of highest to lowest 64 bit unsigned ID finding the first server is rather simple. Our *findFirstReplica()* method looks as follows:

```
findFirstReplica(key):
    // 64-bit hash of the key
    long hashedKey = hashTo64bit(key.toString());
    for(SlaveInfo info : keySpace)
        // Return the first slave that has a higher ID than the hashed key
        if(isLessThanEqualUnsigned(hashedKey, info.getSlaveID()))
            return info;
    // If it's greater than the last element we give it the first one.
    return keySpace.get(0);
```

Then, finding the successor of a given slave is rather simple, we just find the index of the primary server provided to us and then return the next SlaveInfo object in the list or alternatively, if we are provided with the last element in our list, the successor is the first element.

Testing Plan

In order to test this section we can perform a series of registrations in order to make sure that our servers get registered to our TPCMaster. We can then perform requests from our master server and ensure that they are routed to the slave servers and if they are we know that our code works.

Task 4: TPCMaster and KVClientHandler

Correctness Constraints

1. Updates to the KVStore should follow the 2 phase commit protocol.
2. Gets should be parallel across caches.

Declarations

No additional declarations are necessary for this section.

Description

In TPCMaster we must implement *run()*, *performTPCOperation()*, and *handleGet()*.

In the *run()* method we must start a registration server in a new thread and have it listen for connections, this can be done rather trivially by calling *connect()* on the registration SocketServer, then creating a new Thread object, and overriding the *Thread.run()* method to call start on our registration SocketServer.

The *performTPCOperation()* is a slightly more complex method and I'll lay out the pseudo-code below:

```
performTPCOperation(KVMessage msg, boolean isPutReq):
    key = msg.getKey();
    value = msg.getValue();
    opId = msg.getTpcOpId();
    synchronized(masterCache.getWriteLock(key)):
        KVMessage req = new KVMessage(isPutReq ? "putreq" : "delreq");
        req.setKey(key);
        if(isPutReq):
            req.setValue(value);
        req.setTpcOpId(opId);

        SlaveInfo primary = findFirstReplica(key);
        SlaveInfo secondary = findSuccessor(primary);

        Socket pCon = primary.connectHost();
        Socket sCon = secondary.connectHost();
        boolean abort = false;
        // Send the initial message
        try:
            req.sendMessage(pCon);
        catch (KVException e):
            abort = true;
        try:
            req.sendMessage(sCon);
        catch (KVException e):
            abort = true;

        // Get the first response
        KVMessage pResp = null;
        KVMessage sResp = null;
        try:
            pResp = new KVMessage(pCon, TIMEOUT_MILLISECONDS);
            sResp = new KVMessage(sCon, TIMEOUT_MILLISECONDS);
        catch (KVException e):
            abort = true;

        if(abort || (!"ready".equals(pResp.getMsgType()) || !"ready".equals(sResp.getMsgType()))):
            // Send abort if we don't recieve anything
            sendDecisionUntilAck(primary, secondary, "abort", opId);
            KVMessage errorMsg = mergeErrorMessages(primary, pResp, secondary, sResp);
            throw new KVException(errorMsg);
        else:
            // Send commit if we've recieved everything
            sendDecisionUntilAck(primary, secondary, "commit", opId);
            // We need to update the cache after our operation completes.
            if(masterCache.get(key) != null):
                if(isPutReq):
```

```
                masterCache.put(key, value);
            else:
                masterCache.del(key);
```

In two cases above we use a method called *sendDecisionUntilAck()*, basically what that does is sends the decision to the slaves until it recieves an acknowledgement. It has the following logic:

```
sendDecisionUntilAck(SlaveInfo primary, SlaveInfo secondary, String decision, String opId):
    boolean pAcked = false;
    boolean sAcked = false;
    Socket pCon = null;
    Socket sCon = null;

    // Build decision
    KVMessage commitMsg = new KVMessage(decision);
    commitMsg.setTpcOpId(opId);

    while(!(pAcked && sAcked)):
        if(!pAcked):
            try:
                // Send decision
                pCon = primary.connectHost();
                commitMsg.sendMessage(pCon);
                // Wait for ack
                KVMessage pAck = new KVMessage(pCon, TIMEOUT_MILLISECONDS);
                if("ack".equals(pAck.getMsgType())):
                    pAcked = true;
            catch (KVException e):
                pAcked = false;
                e.printStackTrace();
        if(!sAcked):
            try:
                // Send decision
                sCon = secondary.connectHost();
                commitMsg.sendMessage(sCon);
                // Wait for ack
                KVMessage sAck = new KVMessage(sCon, TIMEOUT_MILLISECONDS);
                if("ack".equals(sAck.getMsgType())):
                    sAcked = true;
            catch (KVException e):
                sAcked = false;
```

Finally, the *handleGet* method is a little more simple luckily:

```
handleGet(KVMessage msg):
    String key = msg.getKey();
    synchronized(masterCache.getWriteLock(key))
        String value = masterCache.get(key)
        if(value == null)
            SlaveInfo primary = findFirstReplica(key);
            SlaveInfo secondary = findSuccessor(primary);
            KVMessage getReq = new KVMessage("getreq");
            getReq.setKey(key);
            Socket pCon = primary.connectHost();
            getReq.sendMessage(pCon);
            KVMessage pResp = new KVMessage(pCon);
```

```
            if(pResp.getValue() != null)
                value = pResp.getValue();
            else
                Socket sCon = secondary.connectHost();
                getReq.sendMessage(sCon);
                KVMessage sResp = new KVMessage(sCon);
                if(sResp.getValue() != null)
                    value = sResp.getValue();
                else
                    Merge error messages and throw new KVMessage
        // Update the cache
        cache.put(key, value)
        return value
```

In KVClientHandler we must implement *ClientHandler.run()* once again to work with the new TPCMaster. It would operate under the following logic:

```
run():
    try:
        KVMessage msg = new KVMessage(client);
        String type = msg.getType();
        if(type is "getreq")
            String key = msg.getKey();
            String value = tpcMaster.handleGet(msg);
            KVMessage resp = new KVMessage("resp");
            resp.setKey(key);
            resp.setValue(value);
            resp.sendMessage(client);
        if(type is "putreq" or type is "delreq")
            tpcMaster.performTPCOperation(msg, type is "putreq")
            KVMessage resp = new KVMessage("resp");
            resp.setMessage("Success");
            resp.sendMessage(client);
        else
            throw KVException due to unhandleable client request.
    catch (KVException e):
        e.getMsg().sendMessage(client);
```

Testing Plan

In order to test this part we can implement similar tests for TCPMaster and KVClientHandler as we did before. For TCPMaster we can test it in a similar manner to KVServer, but also take advantage of the multi-node capabilities as follows:

- testPutGet: run a simple put and then a get request on the TPCMaster and ensure we get the results we want.
- testPutDel: run a simple put and del and ensure that a get fails.
- testDel: run a del call to the master and ensure that it fails, then perform a put and a del and ensure it works properly, ie. that it actually puts the key into the server and then deletes, causing a get call to fail.

Task 5: TPCMasterHandler and KVServer.hasKey

This class handles two-phase commit logic on the slaves.

Correctness Constraints

- Retrieval operations (i.e. GET) of different keys must be concurrent unless restricted by an ongoing update operation on the same set.
- A slave will send vote-abort to the master if the key doesn't exist for DEL or invalid key/value size (if we don't implement it client/master side)
- Individual slave servers must log necessary information to survive from failures, and at any moment at most one slave server is down.
- Upon revival, slave servers always come back with the same ID.
- We must handle the slave server failing at any point in the 2PC transaction, handle the ignoreNext messages, and handle pathological inputs to the system.
- On failure, rebuild the slave using the log that it has been updating. Don't contact the server or other slaves.
- If a slave receives a duplicate decision from the master because the slave finished performing the operation in phase-2 but fails before sending an ack, the slave hsould immediately send an ack.
- The total number of slave servers is fixed
- Each key for hashing will be stored using 2PC in two slave servers
- The Each slave server will store the first copies of keys with hash values greater than the ID of its immediate predecessor up to its own ID.
- The IDs and hashes will be compared as unsigned 64-bit longs
- Upon receiving the ignoreNext message, the slave server will ignore the next 2PC operation (either a PUT or a DEL) and respond back with a failure message during the phase-1.

Declarations

The helper function, abort(), makes use of repeated code when aborting because a key does not exist or ignoreNext is true and the message type is delreq or putreq.

```
abort(String opID, KVMessage msg)
    KVMessage toAbort = new KVMessage("abort");
    toAbort.setMessage(reason);
    toAbort.setTpcOpId(opId);
    try:
        toAbort.sendMessage(client);
        if (tpcLog != null) {
            tpcLog.appendAndFlush(toAbort);
    catch (KVException e)
        // If this happens we try to send an error message:
        e.getMsg().sendMessage(client);
```

The helper function, respond(), makes use of repeated code when creating a response KVMessage and setting the key and the TpcOpId, and sending the message.

```
respond(String key, String msg, String opID)
    KVMessage response = new KVMessage(msg);
    response.setTpcOpId(opID);

    if (key != null && val != null):
        // Used in the handleGet() method only
```

```
        response.setKey(key);
        response.setValue(val);

    try:
        response.sendMessage(client);
        if (tpcLog != null) {
            tpcLog.appendAndFlush(response);
    catch (KVException e):
        e.getMsg().sendMessage(client);
```

Description

Firstly, the KVServer.hasKey() method is simple:

```
hasKey(String key) throws KVException
    return (dataStore.get(key) != null);
```

Now I will describe the run() method. Basically handle the ignoreNext case, by setting ignoreNext to true, calling tcpLog.appendAndFlush(msg), and creating and sending a new success response KVMessage. Also, create and send a KVMessage of type "abort" if the msgType is "delreq" or "putreq" and ignoreNext is set, or if the request is to delete a key which does not exist in kvServer. Otherwise, run() handles each situation and closes the connection at the end. The pseudocode is given below:

```
run():
    KVMessage msg = new KVMessage(client);

    String key = msg.getKey();
    String msgType = msg.getMsgType();

    if ((msgType.equals("delreq") || msgType.equals("putreq")) && ignoreNext) {
        ignoreNext = false;
        // Do nothing in the case of ignore
        try:
            new KVMessage("resp", "IgnoreNext Error: SlaveServer SlaveServerID has ignored this 2PC req
        catch (KVException e):
                e.getMsg().sendMessage(client);
    else if (msgType.equals("getreq")):
        handleGet(msg, key);
    else if (msgType.equals("putreq")):
        handlePut(msg, key);
    else if (msgType.equals("delreq")):
        handleDel(msg, key);
    else if (msgType.equals("ignoreNext")):
        // Set ignoreNext to true.
        ignoreNext = true;
        // Send back an acknowledgment
        new KVMessage("resp", "Success").sendMessage(client);
    else if (msgType.equals("commit") || msgType.equals("abort")):
        if (ignoreNext)
            ignoreNext = false;
            return;
        // Check in TPCLog for the case when SlaveServer is restarted
        if (tpcLog.hasInterruptedTpcOperation())
            originalMessage = tpcLog.getInterruptedTpcOperation();
        // originalMessage is either the passed in argument or
```

```
        // tpcLog.getInterruptedTpcOperation
        handleMasterResponse(msg, originalMessage, aborted);

        originalMessage = null;
        aborted = false;
    else:
        try:
            new KVMessage("resp", "Error: message type unknown").sendMessage(client);
        catch (KVException e):
            // We can fall through in this case.
    // Finally, close the connection
    closeConn();
```

The method handleGet() calls appendAndFlush(msg) and creates and sends a KVMessage of type "resp" if there is a value for the associated key (i.e. keyserver.get(key) != null).
Otherwise, it calls the abort() helper function defined above.

```
handleGet(KVMessage msg, String key):
    try:
        String val = keyserver.get(key);
        String opID = msg.getTpcOpId();
        respond("resp", opID, key, val);
    catch (KVException e):
        abort(msg.getTpcOpId(), e.getMsg().getMessage());
```

The handlePut() method calls appendAndFlush(msg) and sets aborted to false. It then creates and sends a "ready" KVmessage by calling the respond() helper function defined above.

```
handlePut(KVMessage msg, String key):
    // Store for use in the second phase
    originalMessage = new KVMessage(msg);
    aborted = false;

    tpcLog.appendAndFlush(msg);
    respond("ready", msg.getTpcOpId(), null, null);
```

The handleDel() function is very similar to handleGet() except it does not need to set the key value and it does need to reset this.aborted to false.
The pseudocode:

```
public void handleDel(KVMessage msg, String key) {
    // Store for use in the second phase
    originalMessage = new KVMessage(msg);
    try:
        kvServer.hasKey(key); // Can't delete it if it doesn't exist
        aborted = false;
        tpcLog.appendAndFlush(msg);
        respond("ready", msg.getTpcOpId(), null, null);
    catch (KVException e):
        aborted = true;
        abort(msg.getTpcOpId(), e.getMsg().getMessage());
```

The handleMasterResponse() method takes in the global decision taken by the master, i.e. the masterResp KVMessage, as well as the message from the client, KVMessage origMsg, as well as a boolean origAborted which tells is this slave server aborted it in the first phase. The method calls handleDel() with origMsg if the

type of masterResp is "commit" and the type of origMsg is "delreq." It calls handlePut() with origMsg if the type of masterResp is "commit" and the type of origMsg is "putreq." Otherwise, if the type of masterResp is "abort," it creates a new KVMessage of type "ack" and sends it.

```
public void handleMasterResponse(KVMessage masterResp, KVMessage origMsg, boolean origAborted):
    if(!origAborted):
        if(masterResp.getMsgType().equals("abort")):
            // do nothing
        else if (masterResp.getMsgType().equals("commit")):
            if (origMsg.getMsgType().equals("delreq")):
                kvServer.del(origMsg.getKey());
            else if (origMsg.getMsgType().equals("putreq")):
                kvServer.put(origMsg.getKey(), origMsg.getValue());

    // Send our response
    try:
        KVMessage ack = new KVMessage("ack");
        ack.setTpcOpId(origMsg.getTpcOpId());
        ack.sendMessage(client);
        if (tpcLog != null)
            tpcLog.appendAndFlush(masterResp);
    catch (KVException e):
        try:
            e.getMsg().sendMessage(client);
        catch (KVException e1):
            throw new RuntimeException(e1);
```

Testing Plan

The testing for this part is combined with the testing for the previous part as well as the testing to TPCLog. In our TPCHandlerLogTest, we created a TPCHandler and paired it with a log and performed a series of operations in which we expected the handler to handle our messages correctly: First submitting a putreq, a commit and then another putreq, we were able to ensure that our handler handler the TPC protocol correctly.

Also, as a sanity check, in our EndToEndTests, we perform 2 tests which test the interactions between our TPCMaster and our TPCMasterHandler by submitting a combination of put, get and del requests to ensure that the 2 phase commit protocol works correctly and produces the correct output from an end user's point of view.