Programming Assignment for EM-012

Atul Adya

Feb 2021

[**1. Introduction**](#_heading=h.30j0zll) **1**

[**2. Deliverables**](#_heading=h.1fob9te) **2**

[**3. Client-Server Protocol**](#_heading=h.3znysh7) **2**

[**4. Server Implementation**](#_heading=h.2et92p0) **3**

[**5. Client implementation**](#_heading=h.tyjcwt) **4**

[**6: Dealing with Timeouts**](#_heading=h.3dy6vkm) **5**

[**7. Two Sharded Servers**](#_heading=h.1t3h5sf) **6**

[**Appendix: gRPC brief instructions**](#_heading=h.4d34og8) **7**

**Warning: I highly recommend starting on the assignment early. There are a bunch of “moving parts” and starting late may cause you to panic/miss the deadlines.**

# 1. Introduction

This programming assignment will give you some experience with gRPC and the kinds of problems one has to tackle in a distributed system. In this assignment, you will build a simple **R**emote **in**-memory **k**ey-value store (or a RINK). This remote store exposes a simple hash-table-like API (like a dictionary in a Python program) with operations to insert or update the value of a key, delete a key and list all the keys and values in the table. A client can access this API by sending RPC calls (see below). There could be one server or more servers. For example, if there are two servers, as shown below, the client splits the key space (“shards” the key space) by sending, say, odd keys to Server 1 and even keys to Server 2.



This programming assignment is divided into two parts. In the first part, the client will communicate with a single server. In the second part, it will communicate with two servers such each holds half of the keys approximately.

**The folder for all the files of this assignment are in** [**this Google Drive folder.**](https://drive.google.com/drive/folders/1Ccj1JjRC8doHZk98m2r5zi8AoMRpvHeq?usp=sharing)

# 2. Deliverables

Your assignment is to write the server and different clients as instructed. As instructed below, you will have to deliver a python file for the server and 2 client files. Please note that code will be evaluated using a driver program and diffing of output files. Please make sure that your code runs with the supplied driver program and the output is identical to the given file (unless otherwise stated). We will run the same driver program and diff the output against the same output files to check your assignment.

* The first part of the assignment is due on **Tuesday February 9 11:59 pm IST**
* The second part of the assignment is due on **Thursday Feb 18 11:59 pm IST**

The files should be named in the following manner: keyval\_server\_groupXX.py, keyval\_client1\_groupXX.py, keyval\_client2\_groupXX.py where XX is your group number (e.g., 01, 03, 12, etc). **Please make sure that the code is well-documented and commented since just running the code is not sufficient. Furthermore, if your code is buggy or not fully done, the code documentation will help your case.**

Part 1 consists of everything specified in Sections 3, 4 and 5. Part 2 consists of the remaining part of the document.

Please do ask questions on the mailing list if you are confused about any aspect of the assignment.

# 3. Client-Server Protocol

The protocol between the client and the server is provided to you in **keyval.proto.** The specs in the file describe each proto in detail. The interface exposed by the server is that of a hash table/dictionary. This dictionary in the remote server stores a value for each key along with a version number for the current value. The **version number** is an integer that starts at 1 and is incremented every time the value is updated via the Write call. This version number is used to perform some checks before a Write or a Delete operation can succeed. Here is a brief description of the operations in the protocol (the proto file has the description for the fields):

* **Read:** For a given key, return the corresponding value and version stored in the dictionary. Return an error if the key does not exist in the table. It is also an error to not specify the key in the Read request.
* **Write:** Create a new key/value entry or update the existing value for the given key. The behavior is determined by the version number specified in the Write call and the one stored in the dictionary for the key:
  + If the Write operation’s version number matches the one in the dictionary (‘version check”), the write operation is allowed to proceed and the stored version number is incremented by 1.
  + If the Write operation’s version number < 0, it is treated as a **blind write** and the version check is not performed. The value overwrites the existing value and sets the version number to 1.
  + Errors:
    - A version number of 0 in the Write request is considered invalid.
    - A missing key or value in the request is considered invalid
    - As stated above, an error is returned if the version check fails
* **Delete:** Deletes the value stored for a given existing key. The version check is the same as in the Write call and the whole discussion applies (instead of a blind write, the server performs a blind delete instead). Errors are:
  + A version number of 0 in the Delete request is considered invalid.
  + A missing key in the request is considered invalid
  + As stated above, an error is returned if the version check fails
* **List:** Lists all the keys, values and version numbers stored currently in the dictionary.

# 4. Server Implementation

For the server code, you will have to build a gRPC server - the best way would be to use the RouteGuide [example](https://grpc.io/docs/languages/python/basics/) to get the boilerplate code for starting the server. For each RPC method, make sure that you validate the incoming proto, i.e., if the proto is invalid, return an error to the caller. For example, if the ReadRequest does not specify the key, it is an error.

The server takes the following flags:

* --server\_id <id>: The id of the server (e.g., 1 or 2), that is returned in every Status proto.
* --write\_delay <delay>: The delay in seconds that is added to the Write RPC handling in the server, i.e., before the Write RPC is handled on the server side.
* --port <port>: The port on which the server listens. Default: 50050

Note that adding the delay can simply be done using a sleep call in the Write RPC handler on the server side. For the first part of the assignment, make sure to set the server\_id to be 1 for the server and the port to be 50050.

The errors need to have the following text so that your file has the same text as the one in **client\_single\_server\_results.txt.** In the text below, %s, %d refer to substitution of real values in the error text.

* Writes:
  + When there is a version mismatch
    - *Write aborted. Record version mismatch. Expected = %d, Actual = %d*
  + When try to write a value but the value does not exist:
    - *Write aborted. Record missing but Write expected value to exist at version %d*
* Reads:
  + When the key is not present:
    - *Read aborted. Key not present %s*
* Deletes:
  + When the key is not present:
    - *Delete aborted. Key not present %s*
  + When there is a version mismatch
    - *Delete aborted. Record version mismatch. Expected = %d, Actual = %d*
* **Invalid protos:** See the output in **client\_single\_server\_results.txt** and match the output there.

To verify the server side code, just run **keyval\_driver.py** against your running server and check that it does not complain/crash or throw an error - compare the output with **keyval\_driver\_results.txt**. The file for this code has to be **keyval\_server\_groupXX.py,** e.g., keyval\_server\_group01.py, keyval\_server\_group02.py.

# 5. Client implementation

[The file for the the operations below **keyval\_client1\_groupXX.py]**

The client side needs to send a few RPCs to the server and print the output from the server to a file. Again, I would recommend using the gRPC [RouteGuide](https://grpc.io/docs/languages/python/basics/) example. The operations to execute in your client in the given order are:

1. **Blind write:** Write *Key1, Value1* with no version check
2. **Normal write:** Write *Key1, Value1* expecting the current version to be 1
3. **Version check failure:** Write *Key1, Value3* expecting the current version to be 1
4. **Version failure with key missing:** Write *Key2, Value3, 1*
5. **Normal read:** Read *Key1*
6. **Non-existing key read:** Read *Key2*
7. **Get full state with List:** List
8. **Add new element as a blind write:** Write *Key3, Value3*
9. **Get full state with List:** List
10. **Delete with version check failure**: Delete *Key1* with current\_version stated as 1
11. **Normal delete**: Delete *Key1* with current\_version stated as 2
12. **Delete of a non-existent key:** Delete *Key1* with current\_version as 2
13. **Delete last element:** Delete *Key3* with current\_version as 1
14. **Get full state with List:** List

The results of the above operations should be output to a file (using shell redirection, i.e. “>| /tmp/foo”). We will run your program and look and diff it with the given file **client\_single\_server\_results.txt.** Note that the file just contains the protos returned by the servers along with some extra lines for demarcating tests (in case this is confusing, please do talk to me).

# 6: Dealing with Timeouts

**[There are no files to be submitted for this section - just experiments with results]**

In this assignment, you will deal with timeouts. Recall that you have added the ability to add a delay to the server-side processing of the Write RPC above. The goal is to artificially introduce delays on the RPC path so that we can experiment with timeouts. In this case, write a client file that has a command line parameter **--write\_timeout=<timeout>** that is the timeout specified in the gRPC Write RPC message to the server.

**Experiment 1**

For the following, send blind write RPCs (Write *Key<i>* for i from 0 to 4). Start the server with **--write\_delay=1** (i.e., a 1 second delay on the Write RPC handler).

1. Start a client and on the client side
   * Send 5 RPCs to the server with a timeout of 0.5 second
   * Sleep for 5 seconds
   * Send a List RPC
   * Send 5 RPCs at the end to blindly delete all 5 keys
2. Start a client and on the client side
   * Send 5 RPCs to the server with a timeout of 1.5 second
   * Send a List RPC
   * Send 5 RPCs at the end to blindly delete all 5 keys

For the above RPCs, you will probably need to handle a gRPC error, e.g.,

try:

# Send Write RPCs

except grpc.RpcError as exception:

# Handle the exception, e.g., by printing the exception

**In general, for any of the experiments below, your client code should not crash, i.e., do handle the gRPC exception.**

**Report the following:**

1. For each of the Write RPC sets above, count the number of RPCs that were successful and the number that were unsuccessful.
2. If the numbers are different for the 2 scenarios, can you explain why? If the numbers are the same for the 2 scenarios, can you explain why?
3. What did List return in both cases? Why?

**Experiment 2**

Run the following experiments with a restarted server:

1. Run the server with a write\_delay of 1 second. Then run the client with a timeout = 0.5 and the following behavior:
   * Write *Key1, Value1, current\_version = -1*
   * Delete *Key1, current\_version = -1*
   * *Sleep 1 second*
   * *List*
2. Run the server with a write\_delay of 1 second. Then run the client with a timeout = 1.5 and the following behavior:
   * Write *Key1, Value1, current\_version = -1*
   * Delete *Key1, current\_version = -1*
   * *Sleep 1 second*
   * *List*

**Report the following:**

1. Which Write and Delete calls succeeded and which calls failed?
2. Did List return the same result in A and B? Why or why not?

# 7. Two Sharded Servers

[The client file for this scenario is **keyval\_client\_sharded\_groupXX.py**]

In this case, there will be one client and two servers. The goal is to send approximately half of the RPCs to one server and half to the other one. The keys to use for this scenario are going to be of the form *ShardKey<xx>, i.e., ShardKey01, ShardKey02, ...* In this case, the odd keys go to server 1 and even keys go to server 2 (recall the --server\_id flag that you added in part one of this assignment). Now do the following:

1. Bring up both servers and send a total of 10 RPCs from *ShardKey00 to ShardKey09* and then call List on each server. Make sure that the odd RPCs go to server 1 and even ones go to server 2. The output of the RPCs should be logged to a file and compared with **client\_sharded\_server\_results.txt** (if the results from List are in a different sorted order, that’s ok)**.** The exact calls are:
   1. Write RPCs with odd keys to server 1
   2. Write RPCs with even keys to server 2
   3. List from server1
   4. List from server2
2. Perform the following experiment and report the following:
   1. Kill server 1 and run the same 10 RPCs and report what you see?
   2. What could one do to avoid this problem that you observed in a real system?

# Appendix: gRPC brief instructions

Here are the relevant instructions for installing gRPC. For the gRPC command-line client, I have given instructions for Linux - that is needed to run my driver program. So in your group if someone has Linux, that would be the best way to test against the driver.

* There is a lot of good documentation at the main [gRPC web site](https://grpc.io/). It would be worth reading some of the concepts presented there.
* Install gRPC for python. [gRPC quick start for Python](https://grpc.io/docs/languages/python/quickstart/) has the installation instructions. Note that you may have python3 instead of python (as a symbolic link to python3) . In that case, substitute the commands for python with python3.
* **See the two emails that I have** [**sent**](https://groups.google.com/a/plaksha.org/g/em-012/c/d6JD4bN6QI4/m/MHgoBwsgAwAJ) **to install gRPC for your platform.**

## For the CLI tool

Install the grpc source and compile: See instructions [here](https://github.com/grpc/grpc/blob/master/BUILDING.md). Here’s what I had done but you may have to do more or less depending on the machine.   
  
Linux  
**Install cmake**

sudo apt-get install build-essential autoconf libtool pkg-config  
sudo apt-get install cmake

**Build grpc binaries**

git clone -b v1.34.1 <https://github.com/grpc/grpc> (you should have done this already when you installed the example. This step is here for completeness)  
  
cd grpc  
git submodule update --init

mkdir -p cmake/build

cd cmake/build/

cmake ../..

make

**Now build the grpc\_cli command**

cd cmake/build/

cmake -DgRPC\_BUILD\_TESTS=ON ../..

cd ../..

make grpc\_cli

### For Windows

You could build grpc\_cli from the sources as above using the instructions [here](https://github.com/grpc/grpc/blob/master/BUILDING.md). You could also use an already built executable, e.g. [here](https://github.com/dhtech/proto/blob/master/util/grpc_cli.exe). Please do verify that the executable is safe, e.g., by checking on VirusTotal

### For Mac

You can install cmake etc or try <https://github.com/grpc/homebrew-grpc>. I executed the following commands:

* /bin/bash -c "$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"
* brew install grpc
* To run the driver, you will have place a link to grpc\_cli in the directory where