Operating Systems – Spring 2015 Project 3

Single-Server In-Memory Key–Value Server

Requirements

In Project 3 and Project 4, you will be building a distributed in-memory key-value storage system. The goal of this project is to provide you chance to learn how to write concurrent program, as well as how to start a distributed systems program from scratch (This time we are not going to provide you with code skeletons!). Also the code you will write will be an important part for your project 4.

In this project, you will build an in-memory key-value store with a simple backup scheme. The system contains two servers, one primary server, and one data backup. The specification of the system is as follows:

1) (Data Format) both keys and values are unparsed Strings of variable lengths. You can assume the total amount of data we will store on your servers is of reasonable size (smaller than usable memory on the servers). Each of the keys and values are also of reasonable length. The key-value store behaves like a dictionary data type, with unique keys.

2) (Read/Write/Delete/Update) Only the primary server can handle operations (including reading and writing), from the client. The server use an HTTP interface to take client request. The following table provides the details of required server API. After receiving new key-value pair, the server will only store it in its memory without writing it to disks.

If the client request is unsuccessful, the output should set the `success` to `false`, and contain an optional `error_message` filed for your own debugging purpose. The server needs to support concurrent operation for any number of concurrent client connections (upto the hardware performance limit, of course). You server should NOT crash/hang or record inconsistent data with concurrent client connections.

URL	**HTTP Method**	**Description**	**Input (IN HTTP FORM encoding)**	**Output (IN JSON)**
`http://<IP>:<PORT>/kv/insert`	POST	Insert a new pair	`key=k&value=v`	`{"success":"<true or false>"}`
`http://<IP>:<PORT>/kv/delete`	POST	Delete a pair	`key=k`	`{"success":"<true or false>", “value”:”<value deleted>”}`
`http://<IP>:<PORT>/kv/get`	GET	Query for a value of a key	`?key=k`	`{"success":"<true or false>","vaule":"<value>"}`
`http://<IP>:<PORT>/kv/update`	POST	Update the value of a existing key	`key=k&value=v`	`{"success":"<true or false>"}`

As a side note, this API design follows a popular de facto standard called RESTful API, where you should use the standard HTTP GET and POST encoding for parameters, and the return data format is called JSON. Remember to use the correct encoding (e.g. URL_ENCODING, or JSON library) to treat the key and values, just in case some keys or values contains special characters like spaces, quotes, & or =. Although it is not required by the project, you can find more information about what these abbreviations mean on Wikipedia.

You should check common errors from users and return success field accordingly. Specifically (i.e. we will test these operations), you should check for the following errors.

A) On INSERT op – if the key exists, you should return FALSE in success field;

B) On DELETE op – you should return success =FALSE if the key does not exist;

C) On a GET op – you should return success=FALSE if the key does not exist;

D) On an UPDATE op – you should return FALSE if the key does not exist.

If the server fails during the operation, you should set the success field to false, and optionally set an error message describing the cause of the failure. In any case, when the success field is set to FALSE, we do not care the value returned in the message, and you can set it arbitrarily.

3) (Reliability) To improve the durability of data, the primary makes a backup of the data to the client, synchronously (i.e. the primary makes sure that the backup have received the data, before finishing the client write). To simplify the project, we can allow the case when backup is down, the primary can let the client request to fail. If the primary fails / or returns an unsuccessful message the client can just give up and expose the error to the user.

4) (Backup Server operation) The backup server only provide data backup. It will never become the primary, even in the case of a primary failure. If the primary fail, the backup server will wait until the primary comes back, transfer the entire content to the primary, and continue.

5) (Configuration) Both primary and backup servers should take the same configuration file "conf/settings.conf". The configuration file is in JSON format, {"primary":"<primary ip>", "backup":"<backup ip>", "port":"<port>"}. Your configuration file must be in this format as we will use it to configure your tests on the grading servers.

6) (Server management and testing tools) It is important to have an management interface for any distributed systems, and it is essential for us to test your system too. At least you will need to provide the following programs or scripts:

compile.sh in your project's root directory should compile everything and put everything in place.

/bin directory should contain all the scripts and programs we need to test your program. Specifically it should contain

bin/start_server –p starts the primary

bin/start_server –b starts the backup

bin/stop_server –p stops the primary

bin/stop_server –b stops the backup

Further more, the both the primary and the back server should offer an addition set of rest APIs to report its status to the system administrators (and the auto grading scripts). For simplicity, the server can use the same port to handle the management API calls. Also, the management operations do not need to guarantee consistency. i.e. you don't have to consider the case that these APIs are called at the same time with the client APIs.

URL	**HTTP Method**	**Description**	**Input**	**Output**
`http://<IP>:<PORT>/kvman/countkey`	GET	Return a count of number of keys stored	None	`{"result":"<number of keys>"}`
`http://<IP>:<PORT>/kvman/dump`	GET	Returns a list of all key,value pairs stored	None	`[["<key>", "<value>"], ...]`
`http://<IP>:<PORT>/kvman/shutdown`	GET	Shutdown the server	None	None

7) Your servers should run on Linux machines (a modern version of Ubuntu or CentOS).

8) (Testing requirements) In addition to the server, you will be required to write a test program (or test script) to test your server (and other people's servers too). The test program should be a concurrent program that is able to send a large amount of concurrent requests over the API to the servers, and check the results. The test program should perform a black-box test, where the client should only test the APIs in the specification above and the required scripts, without knowing about the implementation details of the servers. Your client should be able to run against other groups' servers too. Try to find for corner cases that break other peoples servers, and include them in your test. Given that you handled the corner case in your servers yourself, and the corner case is within the specification above, you will get extra credit!

Your test program should be "bin/test", and we can run it without any flag to do the test. Please contain a "readme" in the root directory that contains the information of what you are testing and where the source files of your test program are. This information is very important since we need to make sure your test program do the right things.

Your test program should not output any information while running if we run it without any flag. After running the test program, it should output some performance metrics to standard output in the following format.

Result: <success or fail>

Insertion: <number of insertions succeeded>/<total number of keys inserted>

Average latency: <key insert>/<key get>

Percentile latency: <20th of insert>/<20th of get>, <50th of insert>/<50th of get>, <70th of insert>/<70th of get>, <90th of insert>/<90th of get>

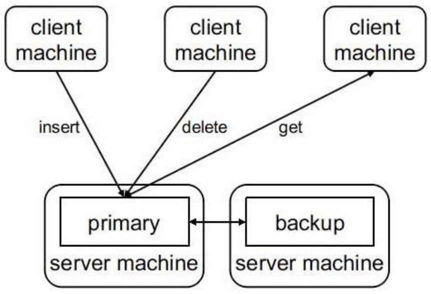

Figure: A single-node key-value store with three clients making simultaneous requests.

Source Code and Testing

For this project, you do not need to submit any document. Please contain the scripts that mentioned in requirement 6. We will test your program in a 64bit Linux environment.

1. To get the basic score (60%), your server should pass our naive test program that will test your server in the following order:
    1. Insert 10 pair, read it back – 5%
    2. Restart backup, on successful restart – 5%
    3. Delete 2 pair – without error return 5%
    4. Update 2 pair, read back the results 5%
    5. Restart primary, on successful restart – 5%
    6. Dump all key-values, and check with desired results – 35%
2. The remaining 40% is assigned as follows:
    1. If your server cannot pass your own testing program or your expected result is wrong, your group will get 0 in this part.

    2. We will use other groups' tests to test your program. There are 10 groups in this class. If you can pass at least 8 tests of the 10 groups, you get 40%. Otherwise failing each test costs you 5%.

3. Extra Credits Possible:

   If your program passes your own test, and if your test successfully catches bugs in > 4 groups, we will offer you 5 points of extra credits per group you catch (given that your tests follows the specification – the TAs will determine if your test is fair). For example, if your test cause 6 groups to fail, you get (6 - 4) * 5 = 15 points, on top of everything else. So, keep what you are testing secret to get points from others!

Submission

Submit your code (all the files of the skeleton code and your own library) to learn.tsinghua.edu.cn website.