MP 2 — Key–value Store

Created by Zhang, Xuewei, last modified on Mar 10, 2017

Due: Monday, April 10th, 11:59 pm

Demos will be conducted on the day after the MPs are due. Details will be given later.

You are encouraged to start this MP early since it involves considerable design, development, debugging and experimentation from your end. Whether you are doing the MP individually or as a team, the expectations and requirements are the same. We recommend not leaving the work for the days before the deadline. Please use Piazza for any clarifications.

In distributed systems, storing and sharing data across multiple machines is essential to many applications. In this MP, you will implement a scalable, failure-proof, in-memory key-value store. You can reuse your code from MP1.

I. Basic Functionality:

This distributed key-value store is made of many processes (one process per VM), each process should act as both client and server. The key-value store takes user commands from stdin, and prints output to stdout or save output to a local file. This key-value store supports an interface with following operations: ***SET***, ***GET***, ***OWNER***, ***LIST_LOCAL, BATCH***. A user should be able to write to a key in any VM, then read the key from any VM.

1. CLI Format

- Each line of input from stdin is one command.
- Parameters in one command will be separated by exactly one space.
- The content of a key is a string starting with an English alphabetic character, and consists of any printable ASCII characters except for whitespace.
- The content of a value is either a string that can contain any printable ASCII character except a newline

2. SET

SET will take 2. The first parameter is the key, the second parameter is the value It writes the key value pair to the key-value store. Note that the 2nd parameter may include whitespace; you should capture the remainder of the line after the key name (and separating space) as the value. If the key already exist in the key-value store, overwrite it; if the key doesn't exist, create it.

After every SET command, you should print the line "`SET OK`"

Usage:

```
SET course_name cs425

SET score 95

SET name Joe Student
```

3. GET

GET will take 1 parameter: the key to be read. It reads from the key-value store. The input parameters contain information about the key. You should print "Found: value" or "Not found"

Use case:

```
GET course_name
```

stdout: **`Found: cs425`**

`GET score`

stdout: **`Found: 95`**

`GET name`

stdout: **`Found: Joe Student`**

`GET quest`

stdout: **`Not found`**

4. OWNERS

OWNERS will take one parameter. The parameter is the key, and you should print out which machines are holding this key on stdout. You should print out the VM IDs of the machines, separated by exactly one space.

Use case:

`OWNERS course_name`

stdout: **`02 08 03`**

5. LIST_LOCAL

LIST_LOCAL will take zero parameter. When any machine get this command, it should print out all the keys that are saved locally on that machine. You should print out the keys, separated by a newline character. At the end, print the string "`END LIST`". The keys should be printed in a sorted order.

Use case:

LIST_LOCAL

stdout: **course_name**

stdout: **name**

stdout: **score**

stdout: **END LIST**

6. BATCH

BATCH will take two parameters and execute a list of commands. E.g., if file1 contains:

```
GET course_name
OWNERS course_name
SET new_key some new value
OWNERS new_key
```

`and one ran the command:`

`BATCH file1 file2`

Then the commands in file1 would be executed and file2 would contain the output:

```
Found: cs425
02 08 03
SET OK
01 05 04
```

There won't be recursive BATCH operations: in an input file, no BATCH command will be given.

II. Interface Requirements:

1. **When starting a process, you should type in at most one parameter: the vm id of the current vm.** All other information necessary to boot up a cluster must be saved on a configuration file or hard coded. Such as, the hostnames of all your vms, the port you will use, etc.

2. If the command from user is invalid/illegal, or an error occurs during the execution of the command, print those information through stderr, not stdout.

3. Any debugging/logging information should be printed through stderr. You are responsible of keeping stdout channel clean by only printing the information asked in each interface.

III. Scalability and Performance Requirements:

1. You must partition the key space onto each VMs, so that it's capacity is scalable. **In your design, if any VM keeps a list of all the keys, 0 point will be given for the entire MP.**

2. You will need to use some lookup algorithm to find the owners of a key. The lookup algorithm must have a time-complexity and bandwidth usage smaller or equal to O(logN), where N is the number of machines in your cluster. **In your design, if the lookup algorithm is O(N) in time-complexity or bandwidth usage, 0 point will be given for the entire MP.**

3. **K-V store need to be able to add machines while operating, and keys should be rebalanced** after new machine joins the cluster.

4. You must have one process per machine. **Within each VM, you can use multiple threads, but can not use multiple processes**.

5. You must use the **REUSEADDR/REUSEPORT option** in all your TCP servers. So when we reboot the processes, it doesn't take too long.

IV. Failure Model and Reliability Requirements:

1. We will **not** use tc to add network delay to your VMs during this MP's demo.

2. Assume crash-stop failure model for processes and unreliable network.

3. Your system should tolerate up to **two** simultaneous failures. After these failures, the system can be in a *recovery period* (no more than 1 minute per million keys stored) during which further failures cannot be tolerated. During this recovery period, you should still be able to respond to new GET/SET/OWNER/etc. requests. Once the recovery period is complete, new failures should be tolerated.

4. You can assume the cluster will always have at least 3 machines still alive.

5. Previously failed machines can be restarted and the system should then undergo a *rebalancing* period. The rebalancing period should take a reasonable time (no more than 1 minute per million keys stored). If a machine is added during a recovery period, you may delay rebalancing until recovery is complete.

6. Once rebalancing is complete, the new systems can be included for the purposes of satisfying requirement 4. E.g., the following sequence of events is possible:

    1. A B C are alive and store all the keys

2. D E F come up
3. Rebalancing to distribute keys to D E F
4. Rebalancing is complete
5. A B crash
6. Recovery period
7. Recovery complete
8. C crashes
9. Recovery period
10. Recovery complete

Through the entire sequence, the system should remain **available** and **not lose data**. On the other hand, if A crashed before step 4 or if C crashed before step 7, that exceeds the crash tolerance of the system.

7. A failure detector should detect any process failure within 20s, and has a false positive rate **very** close to 0. **Any false positive detection observed during the demo will be penalized.**

V. Consistency Requirements:

Eventual consistency.

VI. Implementation & Testing Tips:

1. You should start with modularizing the failure detector part in MP1, so that it's reusable.

2. You should test various failure cases, make sure you don't lose any data. Use the BATCH command and/or redirecting of stdin/stdout in your tests.

3. Do proper synchronization between threads to prevent race conditions.

Instructions

- You have an option of working with Java, Python, Go or C++ for your MPs. **But no third-party libraries unless approved by TA or Professor.**
- Please create a private git repository on Gitlab between you and your team mate. Please don't make your source code public till the end of the semester.
- The code submitted on Gitlab before the deadline would be counted as your submission.
- We will be running plagiarism detection on all your code.
- You are required to add a README.md file to your repository which has all the instructions about how to set up your code. The TAs might run your code again after the demo.
- Add all the course staff to your repository with **Reporter** access. The following are the net ids to be added:
  - nikita, jmehar2, ideshpa2, tzhu13, xzhan160
- Every team has been assigned a VM cluster, the details of which can be found on Piazza. It is your responsibility to ensure that your code is functional on the VMs that are assigned to you. If you face any issues with the VMs, please contact Engr-IT. It may take them a short while to respond, so make sure not to leave testing on VMs until the end.
- Add the design document to your repository and bring a print out of the same to the demo, the details of which are in the following section.

Design Document

In addition to the demo, you will be required to submit a design document of 2–3 pages, describing your design for the scalable, failure-proof, in-memory key-value store. You will need to describe the algorithm you chose and explain why you chose it over other potential alternatives, as well as how some of the implementation details and parameters were decided on. You must show theoretical estimation about the scalability: when you add more machines to the cluster, how does that effect your capacity / memory usage per node / bandwidth usage per lookup / latency per lookup / failure detection bandwidth usage / failure detection time....

You are required to plot suitable metrics with respect to number of nodes in the cluster to show that your system is scalable. Explain these graphs. Take multiple readings and plot averages as well as confidence intervals.

You should also describe your approach to testing your design.

Grading rubric

Demo	
GET/SET Functionality	10%
OWNERS/LIST_LOCAL Functionality	10%
Load testing of GET/SET with BATCH	10%
Recovery after a failure	10%
Rebalancing after node restart	10%
Surviving two simultaneous failures	10%
Surviving many failures / additions with recovery/rebalancing in between	10%
Design document	20%
Code quality	10%

The tests during demo has dependencies. Missing one functionality may make other tests very hard or impossible, which will result in 0 points in those fields. For example, without GET/SET working, we cannot test robustness of the system under failure; without OWNERS/LIST_LOCAL functionality, we cannot test the balancing after membership changes. So please make sure at least implement the basics.

Also be aware, there are many penalizing items. Please read the requirements (especially the **Bold** parts) carefully.

Academic Integrity

You are not allowed to post solutions, ideas or any code on Piazza. **Violations of academic integrity, including any sharing of code outside of your group, will be punished severely**; see Section 1-402 of the Student Code as well as the Computer Science Honor Code for more information. Minimum penalty: 0 grade on the MP. Maximum penalty: expulsion. All cases will be reported to CS, your college and the senate committee.

Start early and happy coding!

No labels