Assignment 3: Distributed Key-Value Service

Due: 11:59 p.m., Tuesday, December 6, 2016

Outline

1. Introduction

GOAL: To explore issues of fault tolerance in distributed applications through the development of a distributed key-value service.

Overview

Distributed key-value stores provide a scalable, high-performance alternative to relational SQL databases and are found behind many modern web applications.

A key-value store provides a simple and flexible data storage mechanism through two operations, GET and PUT. Data items (also called objects) in the store are identified by a key (typically a string representing the name of the object). The object itself is the "value" associated with the key and can be an arbitrarily sized sequence of bytes. A GET operation requests a key and receives the value corresponding to that key, while a PUT operation updates the value corresponding to a particular key.

A distributed key-value store consists of multiple key-value servers, each of which is responsible for a portion of the key space (i.e., the range of possible keys). Since keys may be arbitrary strings, they are first hashed to a unique fixed-length identifier which serves as the key in all operations within the key-value service. For example, MD5 can be used to generate a 128-bit hash of the key string. Finally, the request (GET or PUT) can be issued to the server responsible for the part of the key space that includes the identifier. Various strategies are used for mapping a key to the responsible server.

Many sophisticated reliable key-value stores have been described in the research literature. For the purposes of this assignment, we will make some heavy simplifying assumptions (not all of which are reasonable in real-world services).

Assumptions:

- **Failure model: fail-stop.** A failed key-value server is assumed to have crashed. It may only be restarted with the same parameters (same server id, same communication port number(s), etc.). A restarted server will know that it is rejoining an active service, rather than initializing a fresh service. *The service should tolerate a single failure (f=1).*
- **Timing model: partial synchrony.** We can set timeouts on message replies after which a server can be deemed to have failed.
- **Membership: static configuration.** All key-value servers are initialized when the service is started. There are no dynamic server departures or additions, other than the failure and recovery of an existing server.
- **Consistency:** GET requests should typically return the result of the most recent completed PUT on a given key. The key-value service does not provide concurrency control, or support operations on multiple keys. Thus, if two clients GET the value for a particular key, increment it locally, and PUT back the new value, the result of one of the updates will be overwritten by the other. This is the same as two threads reading a memory location into a register, incrementing the register, and writing it back to memory without any synchronization between the threads. Furthermore, if a client C1 issues a PUT operation and the update is carried out on server Si (as soon as the secondary replica gets updated), a GET operation issued by another client C2 may see the local update in Si, *before* the acknowledgment message reaches the client C1.
- **Reliability and Availability:** the service should continue to handle GET/PUT requests as long as no more than *f=1* server is failed. We will use in-memory replication to other servers in the service, rather than persistent storage, to provide these properties. You can assume that a server has enough memory to contain the specific data, and cannot run into out-of-memory issues. The storage and the network are assumed to be failure-free.
  Nevertheless, this does **not** mean that your code should blindly assume that nothing else can fail during recovery. You should always make sure to add error checks where necessary in your code and to gracefully shutdown rather than segfault.
- Additional assumptions may be added to this list, based on questions that arise during the assignment.

Your task in this assignment is to design and implement a distributed key-value service based on the assumptions listed above, as well as the details from the following sections.

2. Design details

2.1 Key-value system

Overview: The key-value store architecture consists of several KV servers (S1, S2,.., Sn) and one metadata server (M). The key-value servers serve PUT/GET requests from clients. The metadata server which keeps track of the servers in the system, detects failures (via heartbeat messages), and coordinates the recovery process. Each key is replicated twice: once on a *primary server* (the one that stores the *primary replica*) and once on a *secondary server* (the one that stores the *secondary replica*; more on this later in this section).

When issuing a PUT/GET operation on a key, a client will contact M to find out which one is the right primary server Si for that key, then contact Si.

A client request can timeout if Si is down, at which point the client must retry contacting M (you can assume M can never go down), until it gets a live server and completes its request. More on failures later.

2.1.1. Key-value server

A key-value server must service client requests, send heartbeat messages to M, and assist in the recovery process when a server goes down.

Client requests: A key-value server is waiting for GET/PUT requests from clients and acts accordingly:

- **GET requests**: Server sends back the value corresponding to the requested key. If the key is not found among its primary set of keys, the server responds with a special message indicating to the client that the key is invalid.
- **PUT requests**: Server updates the value corresponding to the specified key. If the key does not fall within its key range, the request is ignored and an "invalid request" message is sent to the client (this is to avoid clients bypassing M and trying to maliciously contact any server with random PUT requests). If the key does not exist (but falls within its key range since M directed the client here), then the key-value pair is created locally. If the key does exist, it gets updated with the new value. The server must propagate these changes to the secondary replica *before* responding to the client. You can assume that the server has the information on which server has the secondary replica. For simplicity, you can assume that each server's key range is replicated entirely on only one other server. In a real life system though, this is not necessarily a good assumption. You may also assume that there are at least 3

servers in the system, and that servers are never cross-replicated (there is no pair of servers Si and Sj such that Sj is primary for Si's data and vice-versa.)

Heartbeat messages: A key-value server sends periodic "heartbeat" messages to M, in order to indicate that the server is still alive.
Simplifying assumption: You can assume that the network does not lose the heartbeat messages, so the only way that these don't get delivered to M is if the corresponding server has actually gone down. In a real system, it is possible that a key-value server could get overloaded with requests from clients and fail to send a heartbeat to M within a reasonable timeframe. In such a situation, M could incorrectly consider a key-value server as having failed.

Recovery process: The server must assist in the recovery process, in order to bring back up a replacement for a failed server. More on this later, when we discuss the recovery flow.

2.1.2. Metadata server

The metadata server M contains information on the live key-value servers, their key-value ranges (we'll make the simplifying assumption that each server gets its own subrange of keys from the entire key range).

Client requests: When M receives a client PUT/GET request for a specific key, it informs the client which server Si holds the primary replica for that key. M is the sole authority for which servers own the primary and secondary replicas for each key. No other S server can modify M's mapping table.

Heartbeat messages: M receives periodic "heartbeat" messages from all servers. If a server S fails to send a heartbeat on time, then that server is considered failed, and M goes into *recovery mode*.

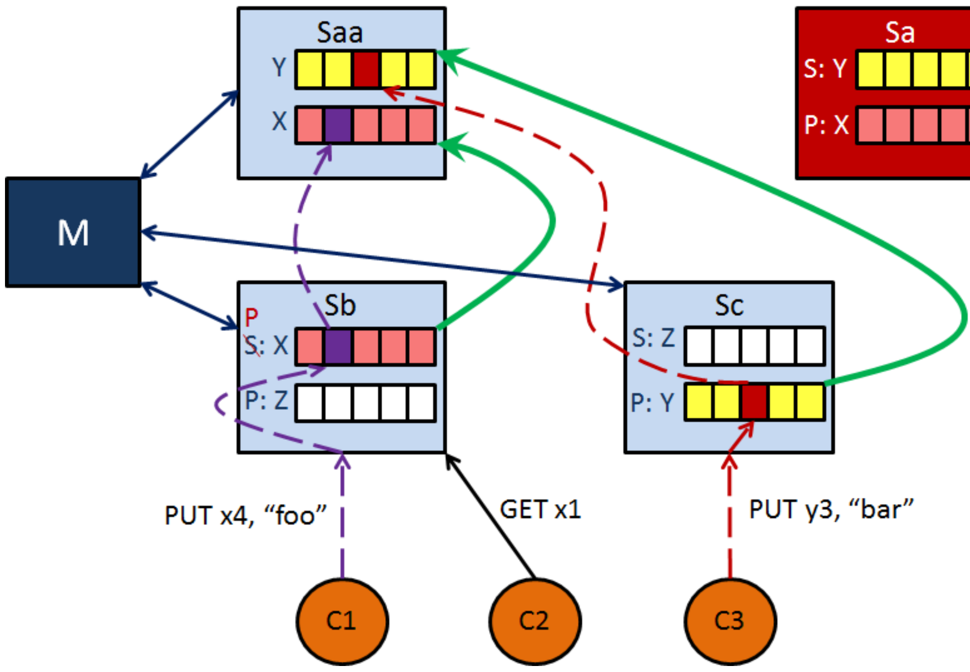
Recovery process: The recovery flow is described later on in more detail.

2.2 Recovery protocol

2.2.1. Recovering from a failed server

Once a server failure is detected, M goes in recovery mode. Let's consider that server Sa is the one that failed (remember: we assume no more than f=1 can fail). Let's assume that Sa's primary replicas are the set of keys X, and its secondary replicas are the set of keys Y. Let's assume that Sb has the secondary replicas for set X, and Sc acts as the primary fo set Y. For easier visualization, please refer to the diagram below.

At this point, M will spawn a new server Saa in *reconstruction mode*, to eventually replace Sa and take over as a primary for set X and as a secondary for set Y. M will contact Sb and instruct it to go in recovery mode as well, then send Sb the address of Saa. Sb will be responsible for sending Saa its X data asynchronously (more on this later). M will now mark Sb as the primary for set X, and direct future client requests to Sb for keys that fall into set X. M will also mark Saa as the secondary for X (only temporarily, until Saa is up to speed and can take over as a primary). Finally, M will instruct Sc to send to Saa set Y (as well as in-flight updates to keys in set Y).



Saa continues to receive the set of keys X from Sb, and the set of keys Y from Sc (Sb and Sc continue to receive client requests for X and Y, to maintain availability). Once Saa is up to date (including any other updates that Sb and Sc receive during the recovery process), M will modify its mappings once more, to mark Saa as the new primary for set X, and turn Sb back into a secondary replica server for X. M goes back to the normal operation mode, as do all the servers involved in the recovery (Saa, Sb and Sc). The recovery flow is described later on in more detail.

2.2.2. Recovery flow

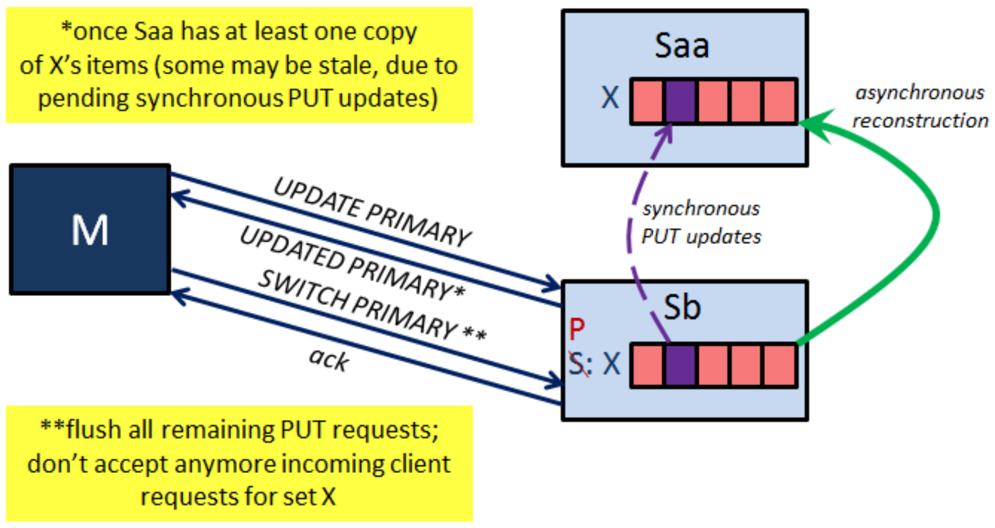
You can view the recovery flow as a sequence of steps. Each server will act as a state machine and transition between these states accordingly.

- 1. M detects failure, spawns a new server Saa to replace the failed server Sa.
- 2. M sends Sb a *UPDATE-PRIMARY* message containing information on Saa.
- 3. Sb receives the *UPDATE-PRIMARY* message. Then, Sb spawns a new thread to

asynchronously send its set X to Saa. Basically, this thread will send the set X one by one to Saa in the background, as new PUT requests keep coming in to Sb. Next, Sb sends a confirmation back to M, to indicate that it received the *UPDATE-PRIMARY* message.

- 4. M marks Sb as the primary for set X.
- 5. M sends Sc a *UPDATE-SECONDARY* message containing information on Saa.
- 6. Sc receives the *UPDATE-SECONDARY* message and spawns a new thread to **asynchronously** send its set Y to Saa. Sc confirms the *UPDATE-SECONDARY* message by sending a confirmation message to M.
- 7. At this point, any PUT requests received by Sb are to be sent **synchronously** to Saa, independently of the updater thread. The same goes for Sc. Any PUT operation will update the value "in place" locally on Sb/Sc, before being sent to Saa. You will need to use synchronization to avoid a race condition between the thread sending data asynchronously and the synchronous PUT operations in flight.
- 8. Sb's asynchronous updater is done sending set X to Saa and contacts M with an *UPDATED-PRIMARY* confirmation message. Sb may continue to receive PUT requests and propagate them to Saa at this point until it hears back from M with further instructions.
- 9. M receives Sb's *UPDATED-PRIMARY* message and awaits on confirmation from Sc as well. If it has already arrived, then it can now skip to step 12.
- 10. Sc's asynchronous updater is done sending a secondary copy of Y to Saa. Some synchronous PUT requests may also have been sent to Saa in the meantime (this should not affect consistency in any way, since the synchronized in-place updates guarantee that no ordering violation will occur - see diagram below). Sc sends an *UPDATED-SECONDARY* confirmation message.
- 11. M receives Sc's *UPDATED-SECONDARY* confirmation message.
- 12. M halts any further client requests for the set X until the swap (of Saa taking over as primary for X) is finalized. It can still service client requests for any other keys.
- 13. M sends Sb a *SWITCH PRIMARY* message, to indicate that it should flush any in-flight PUT requests and ignore any further PUT requests for set X.
- 14. Sb receives *SWITCH PRIMARY* message and flushes all the remaining updates to Saa, and responds to the corresponding clients. Any clients that issue PUT requests at this point will be ignored (these clients will timeout and have to contact the metadata server again). Since the time window to flush any remaining requests to Saa should be small, the client retries should not pose a big problem with respect to availability.
- 15. Sb sends a confirmation message to M, indicating that the switch primary message was handled.

- 16. M receives an acknowledgment message and marks Saa as the new primary for set X, then resumes responding to client requests for keys that fall into set X.
  The last few message exchanges, the asynchronous replica reconstruction, and the synchronous PUT update propagation are represented in the diagram below.


*once Saa has at least one copy of X's items (some may be stale, due to pending synchronous PUT updates)
Saa
X
asynchronous reconstruction
M
UPDATE PRIMARY
UPDATED PRIMARY*
SWITCH PRIMARY **
ack
synchronous PUT updates
Sb
P
S: X
**flush all remaining PUT requests; don't accept anymore incoming client requests for set X

2.2.3. Special cases

Server Si receives a client's PUT request, and needs to update the secondary replica, but the corresponding server Sj is down. The metadata server M has not yet detected that Sj is down.
In this case, Si cannot respond to the client yet, and has to wait for the failure detection and the recovery process to take its course. The simplest thing to do in this case is to just drop the client request and let the client timeout. The client will have to contact M again and eventually will get its request serviced once the failure is detected by M (even if it will retry a few times). If you set a reasonable frequency for the heartbeat messages, these retries should not be noticeable for the end-user under reasonable load. Alternatively (more complex approach): queue up the client requests if the secondary replica is down, and finalize them once the recovery is underway.

Server Si receives a client's PUT request, it relays it to Sj, which acknowledges the update. Before Si can respond to the client, Si fails. In this case, Sj will have a copy of the PUT request, but the client did not get a confirmation of this update. This is ok according to the consistency guarantees that we discussed, since the client will retry the update and eventually it will get an acknowledgment of the update. The tricky part is that, while in recovery mode, a GET request could come in to Sj, *before* the client gets its PUT request acknowledged by Sj. In this case, the GET request will see the PUT request even though the "issuer" of the PUT request did not get a confirmation for it. In real-life systems, such a case typically involves more complex commit protocols, depending on the consistency model. For this assignment, according to the consistency guarantees mentioned above, this is ok because we offer no guarantees on the order of updates with respect to concurrent GET requests.

2.3 Code Provided

We provide the client, an (incomplete) server and metadata server startup code, headers specifying the message formats, and the MD5 hashing code, an implementation of a hashtable (for in-memory storage), and helper functions for networking. You must complete the server and metadata server code. You must **not modify** the client code. The starter code can be found in /u/csc469h/fall/pub/a3-f2016/starter_code.tgz

2.4 Testing

In order to properly test such a system in a real setting, we will be providing a test cluster, in which each of the servers as well as the metadata server will run on their own machines. The clients will also be launched from separate machines.

Due to performance and protection concerns, your code must acquire the test cluster for each test run. As a result, you will not have direct access to the test cluster, but you will be able to submit your code for testing via MarkUs. A background task picks up the code in your repository (the source code files should be at the top level of your A3 directory!) and tests it on the cluster, then sends you the client and server logs by committing them in your svn repository, so that you can inspect the status of your current work.

In order to receive meaningful feedback from your own server implementation, your server implementation must dump all output to a local file called server_*.log (where * stands for the respective server ID). Similarly, the metadata server should dump all output to a file called mserver.log. Each client will similarly produce a client_*.log. Please make sure to stick to this naming convention (do not modify how the "-l" command line arguments are handled in all the programs), and please make sure to keep your output within reasonable means (if your log files are larger than a few MB each, they may not be sent back to your svn repository).
Additionally, the returned results will also include files that contain anything your code prints to stderr and stdout. Please note that some of the long-running tests will be run with only stderr output enabled (stdout and log redirected to /dev/null) in order to produce a reasonable amount of output, so please write only actual error messages to stderr. Finally, keep in mind that you will go through these results to see what went wrong, so meaningful yet succinct output might be more useful than going through hundreds or

thousands of lines of output).

Before you submit to the test cluster via MarkUs, please ensure first that your code:

- a) compiles locally on your own machine, and
- b) runs correctly and completely on your localhost.

Once again, **please make sure to test your code locally first! Test your server and metadata server on your own machine (or on the teaching labs, if you prefer) in the early stages of development**. You will have a lot more control over your server output and thus get much faster feedback than submitting your work to the test cluster and having to wait for the logs. Once your server and metadata server code works correctly on your machine (client requests get serviced properly, recovery completes correctly, etc.), you are ready to submit to the test cluster.

After you have safely debugged your code and ensured that it runs without any problems, please make sure to uncomment the `-O2 -DNDEBUG` from the Makefile, for better performance.

2.5 Stages of implementation (recommended)

1. Your first step should be to add replication to the starter code, and make sure that this works. Before even attempting to detect failures and handle them using the protocol described before, you must make sure that replication works properly in the absence of failures.
2. Next, you must ensure that failures are detected correctly. You should implement heartbeat messages, and make sure that M detects when a server has gone down.
3. The last step involves reacting to detected failures, by launching the recovery process. You must implement the protocol described above, where the replicas get reconstructed on the newly-spawned server instance.

2.6 Conceptual questions

This key-value store contains several simplifying assumptions, which do not hold in a real-world system. Implementing certain features adds complexity that is beyond the scope of this assignment. However, we want you to reason about a set of these limitations by writing a report which addresses the following questions:

1. What happens if a server could run out of storage space for its keys? Rejecting client requests due to being out-of-memory is not a practical way to handle this case (as the starter code does). How would a real-world key-value store handle

such a case? Explain your design decision, potential alternatives, and discuss their merits or drawbacks.

2. What happens if the items in a primary replica were striped across several servers, instead of duplicated on one server only? Explain how this would affect your design, in terms of performance, consistency, resiliance in the face of failures, and failure recovery approach.
3. What happens if failures could happen during recovery? Explain your rationale.
4. How many replicas would we need to tolerate N failures? Explain your rationale.
5. This key-value store does replication on the critical path (PUT requests are not acknowledged to the client until the update propagates to the secondary replica(s)). If you were to change this to an eventual consistency model, how would that work? How would you design the recovery mechanism, if replication was not done on the critical path?
6. Discuss your thoughts on relaxing the static membership assumption, and scaling the system "elastically". In other words, if nodes can be added dynamically in response to increasing load, or if nodes can be taken down when underloaded, what implications would this have on the regular operation and recovery?

Consider all the implications, and discuss solutions for mitigating these problems. Please explain your rationale in a clear and detailed manner. There's no need to write excessively, a couple of paragraphs for each problem will do, as long as the explanations and arguments make sense or expose the right tradeoffs.

2.7 Bonus feature: scalability/performance

While we do expect your code to be efficient, so efficiency will be considered when marking your submission, in terms of performance, there are some restrictions in the starter code, which can pose some scalability bottlenecks, and which you can attempt to address, for bonus marks. The main problem is that the server and metadata server are not necessarily going to scale well. For example, the server could handle all the client requests in parallel, by using multithreading. The metadata server could dedicate a separate dedicated thread for handling heartbeat messages asynchronously from incoming client requests, etc.
Essentially, you must rework the server and/or metadata server to handle requests in a more scalable way. You must consider and handle all the implications of how requests get handled in your approach, such that the parallel server and metadata server **correctly** and **efficiently** handle the requests under both normal and recovery conditions.

3. Literature Resources

- "Dynamo: Amazon�s Highly Available Key-value Store" , SOSP 2007
- Memcached
- Redis

4. What (and how) to hand in

4.1 What

- A report describing your key-value service briefly, alternatives considered or tried, and your answers to the conceptual questions. Remember to include a bibliography, should you cite any resources (e.g., research papers, white papers on existing systems).
- Your source code files and Makefile, such that we can run your code correctly. The source code files should be at the top level of your A3 directory! Please do not submit executables, logs and such, these will be generated automatically during marking.

4.2 How

All your files must be committed in your MarkUs repository for A3. Please **do not create an A3 directory, one should be created for you automatically** (if not, please contact me). If you create your own A3 directory, we will not have access to this and you may not received the marks you deserve. Make sure that you've added and committed all the right files! Double-check by doing a checkout in a different location and making sure that everything is there.

5. Marking scheme

1. Replication (25%)
2. Failure detection (15%)
3. Recovery (35%)
4. Report (15%)
5. Coding style/design (10%)
6. Bonus (scalability) (up to 20%)