CSE 486/586 Distributed Systems Programming Assignment 4

Replicated Key-Value Storage

Introduction

At this point, most of you are probably ready to understand and implement a Dynamo-style key-value storage; this assignment is about implementing a simplified version of Dynamo. (And you might argue that it's not Dynamo any more ;-) There are three main pieces you need to implement: 1) Partitioning, 2) Replication, and 3) Failure handling.

The main goal is to provide both availability and linearizability at the same time. In other words, your implementation should always perform read and write operations successfully even under failures. At the same time, a read operation should always return the most recent value. To accomplish this goal, this document gives you a guideline of the implementation. However, you have freedom to come up with your own design as long as you provide availability and linearizability at the same time (that is, to the extent that the tester can test). The exception is partitioning and replication, which should be done exactly the way Dynamo does.

This document assumes that you are already familiar with Dynamo. If you are not, that is your first step. There are many similarities between this assignment and the previous assignment for the most basic functionalities, and you are free to reuse your code from the previous assignment.

References

Before we discuss the requirements of this assignment, here are two references for the Dynamo design:

1. Lecture slides
2. Dynamo paper

The lecture slides give an overview, but do not discuss Dynamo in detail, so it should be a good reference to get an overall idea. The paper presents the detail, so it should be a good reference for actual implementation.

Step 0: Importing the project template

Just like the previous assignment, we have a project template you can import to Android Studio.

1. Download the project template zip file to a directory.
2. Extract the template zip file and copy it to your Android Studio projects directory.
    a. Make sure that you copy the correct directory. After unzipping, the directory name should be "SimpleDynamo", and right underneath, it should contain a number of directories and files such as "app", "build", "gradle", "build.gradle", "gradlew", etc.

3. After copying, delete the downloaded template zip file and unzipped directories and files. This is to make sure that you do not submit the template you just downloaded. (There were many people who did this before.)
4. Open the project template you just copied in Android Studio.
5. Use the project template for implementing all the components for this assignment.
6. The template has the package name of “edu.buffalo.cse.cse486586.simpledynamo“. Please do not change this.
7. The template also defines a content provider authority and class. Please use it to implement your Dynamo functionalities.
8. We will use SHA-1 as our hash function to generate keys just as last time.
9. The template is very minimal for this assignment. However, you can reuse any code from your previous submissions.
10. You can add more to the main Activity in order to test your code. However, this is entirely optional and there is no grading component for your Activity.

Step 1: Writing the Content Provider

Just like the previous assignment, the content provider should implement all storage functionalities. For example, it should create server and client threads (if this is what you decide to implement), open sockets, and respond to incoming requests. When writing your system, you can make the following assumptions:

1. Just like the previous assignment, you need to support insert/query/delete operations. Also, you need to support @ and * queries.
2. There are always 5 nodes in the system. There is no need to implement adding/removing nodes from the system.
3. However, there can be *at most 1 node failure at any given time*. We will emulate a failure only by force closing an app instance. We will ***not*** emulate a failure by killing an entire emulator instance.
4. *All failures are temporary;* you can assume that a failed node will recover soon, i.e., it will not be permanently unavailable during a run.
5. *When a node recovers, it should copy all the object writes it missed during the failure.* This can be done by asking the right nodes and copy from them.
6. *Please focus on correctness rather than performance.* Once you handle failures correctly, if you still have time, you can improve your performance.
7. Your content provider should support *concurrent read/write operations*.
8. Your content provider should handle *a failure happening at the same time with read/write operations*.
9. *Replication should be done exactly the same way as Dynamo does.* In other words, a (key, value) pair should be replicated over three consecutive partitions, starting from the partition that the key belongs to.
10. Unlike Dynamo, there are two things you do not need to implement.
    a. Virtual nodes: Your implementation should use physical nodes rather than virtual nodes, i.e., all partitions are static and fixed.

    b. Hinted handoff: Your implementation do not need to implement hinted handoff. This means that when there is a failure, it is OK to replicate on only two nodes.
11. All replicas should store the same value for each key. This is “per-key” consistency. There is no consistency guarantee you need to provide across keys. More formally, you need to implement *per-key linearizability*.
12. Each content provider instance should have a node id derived from its emulator port. This node id should be obtained by applying the above hash function (i.e., genHash()) to the emulator port. For example, the node id of the content provider instance running on emulator-5554 should be, *node_id = genHash(“5554”)*. This is necessary to find the correct position of each node in the Dynamo ring.
13. Your content provider’s URI should be “content://edu.buffalo.cse.cse486586.simpledynamo.provider”, which means that any app should be able to access your content provider using that URI. This is already defined in the template, so please don’t change this. Your content provider does not need to match/support any other URI pattern.
14. We have fixed the ports & sockets.
    a. Your app should open one server socket that listens on 10000.
    b. You need to use run_avd.py and set_redir.py to set up the testing environment.
    c. The grading will use 5 AVDs. The redirection ports are 11108, 11112, 11116, 11120, and 11124.
    d. You should just hard-code the above 5 ports and use them to set up connections.
    e. Please use the code snippet provided in PA1 on how to determine your local AVD.
        i. emulator-5554: “5554”
        ii. emulator-5556: “5556”
        iii. emulator-5558: “5558”
        iv. emulator-5560: “5560”
        v. emulator-5562: “5562”
15. Any app (not just your app) should be able to access (read and write) your content provider. As with the previous assignment, please do not include any permission to access your content provider.
16. Please read the notes at the end of this document. You might run into certain problems, and the notes might give you some ideas about a couple of potential problems.

The following is a guideline for your content provider based on the design of Amazon Dynamo:

1. ***Membership***
    a. *Just as the original Dynamo, every node can know every other node.* This means that each node knows all other nodes in the system and also knows exactly which partition belongs to which node; any node can forward a request to the correct node without using a ring-based routing.
2. ***Request routing***
    a. Unlike Chord, each Dynamo node knows all other nodes in the system and also knows exactly which partition belongs to which node.

   b. Under no failures, a request for a key is directly forwarded to the coordinator (i.e., the successor of the key), and the coordinator should be in charge of serving read/write operations.

3. ***Quorum replication***
   a. For linearizability, you can implement a quorum-based replication used by Dynamo.
   b. Note that the original design does not provide linearizability. You need to adapt the design.
   c. *The replication degree N should be 3.* This means that given a key, the key's coordinator as well as the 2 successor nodes in the Dynamo ring should store the key.
   d. *Both the reader quorum size R and the writer quorum size W should be 2.*
   e. The coordinator for a get/put request should *always contact other two nodes* and get a vote from each (i.e., an acknowledgement for a write, or a value for a read).
   f. For write operations, all objects can be versioned in order to distinguish stale copies from the most recent copy.
   g. For read operations, if the readers in the reader quorum have different versions of the same object, the coordinator should pick the most recent version and return it.

4. ***Chain replication***
   a. Another replication strategy you can implement is chain replication, which provides linearizability.
   b. If you are interested in more details, please take a look at the following paper: http://www.cs.cornell.edu/home/rvr/papers/osdi04.pdf
   c. In chain replication, a write operation always comes to the first partition; then it propagates to the next two partitions in sequence. The last partition returns the result of the write.
   d. A read operation always comes to the last partition and reads the value from the last partition.

5. ***Failure handling***
   a. Handling failures should be done very carefully because there can be many corner cases to consider and cover.
   b. Just as the original Dynamo, each request can be used to detect a node failure.
   c. *For this purpose, you can use a timeout for a socket read;* you can pick a reasonable timeout value, e.g., 100 ms, and if a node does not respond within the timeout, you can consider it a failure.
   d. ***Do not rely on socket creation or connect status to determine if a node has failed.*** Due to the Android emulator networking setup, it is **not** safe to rely on socket creation or connect status to judge node failures. Please use an explicit method to test whether an app instance is running or not, e.g., using a socket read timeout as described above.
   e. When a coordinator for a request fails and it does not respond to the request, *its successor can be contacted next for the request.*

Testing

We have testing programs to help you see how your code does with our grading criteria. There are 6 phases in testing. The first three phases will not take much time, so it'll be better to finish them as quickly as possible. You will then be able to spend most of your time for the last three phases.

1. Testing basic ops
    a. This phase will test basic operations, i.e., insert, query, delete, @, and *. This will test if everything is correctly replicated. There is no concurrency in operations and there is no failure either.
2. Testing concurrent ops with different keys
    a. This phase will test if your implementation can handle concurrent operations under no failure.
    b. The tester will use independent (key, value) pairs inserted/queried concurrently on all the nodes.
3. Testing concurrent ops with same keys
    a. This phase will test if your implementation can handle concurrent operations with same keys under no failure.
    b. The tester will use the same set of (key, value) pairs inserted/queried concurrently on all the nodes.
4. Testing one failure
    a. This phase will test one failure with every operation.
    b. One node will crash before operations start. After all the operations are done, the node will recover.
    c. This will be repeated for each and every operation.
5. Testing concurrent operations with one failure
    a. This phase will execute operations concurrently and crash one node in the middle of the execution. After some time, the failed node will also recover in the middle of the execution.
6. Testing concurrent operations with one consistent failure
    a. This phase will crash one node at a time consistently, i.e., one node will crash then recover, and another node will crash and recover, etc.
    b. There will be a brief period of time in between the crash-recover sequence.

Each testing phase is quite intensive (i.e., it will take some time for each phase to finish), so the tester allows you to specify which testing phase you want to test. You won't have to wait until everything is finished every time. However, you still need to make sure that you run the tester in its entirety before you submit. *We will not test individual testing phases separately in our grading.*

- You can specify which testing phase you want to test by providing '-p' or '--phase' argument to the tester.
- Note: If you run an individual phase with "-p", it will always be a fresh install. However if you run all phases (without "-p"), it will not always be a fresh install; the grader will do a

fresh-install before phase 1, and do another fresh-install before phase 2. Afterwards, there will be no install. This means that all data from previous phases will remain intact.
- '-h' argument will show you what options are available.
- The grader uses multiple threads to test your code and each thread will independently print out its own log messages. This means that an error message might appear in the middle of the combined log messages from all threads, rather than at the end.
- Download a testing program for your platform. If your platform does not run it, please report it on Piazza.
    - Windows: We've tested it on 32- and 64-bit Windows 8.
    - Linux: We've tested it on 32- and 64-bit Ubuntu 12.04.
    - OS X: We've tested it on 32- and 64-bit OS X 10.9 Mavericks.
- Before you run the program, please make sure that you are running five AVDs. `python run_avd.py 5` will do it.
- Run the testing program from the command line.
- On your terminal, it will give you your partial and final score, and in some cases, problems that the testing program finds.

Submission

We use the CSE submit script. You need to use either "submit_cse486" or "submit_cse586", depending on your registration status. If you haven't used it, the instructions on how to use it is here: https://wiki.cse.buffalo.edu/services/content/submit-script

You need to submit one file described below. Once again, you must follow everything below exactly. Otherwise, you will get no point on this assignment.

- Your entire Android Studio project source code tree zipped up in .zip: The name should be SimpleDynamo.zip.
    a. Never create your zip file from inside "SimpleDynamo" directory.
    b. Instead, make sure to zip "SimpleDynamo" directory itself. This means that you need to go to the directory that contains "SimpleDynamo" directory and zip it from there.
    c. Please do not use any other compression tool other than zip, i.e., no 7-Zip, no RAR, etc.

Deadline: **5/6/2016 (Friday) 11:59am**

The deadline is firm; if your timestamp is 12pm, it is a late submission.

Grading

This assignment is 20% of your final grade. Also there is extra credit if you pass all 6 phases.

- Phase 1: 3%
- Phase 2: 4%
- Phase 3: 3%
- Phase 4: 5%

- Phase 5: 5%
- Phase 6: 3%

Notes

- Please do not use a separate timer to handle failures. This will make debugging very difficult. Use socket timeouts and handle all possible exceptions that get thrown when there is a failure. They are:
    - SocketTimeoutException, StreamCorruptedException, IOException, FileException, and EOFException.
- Please use full duplex TCP for both sending and receiving. This means that there is no need to create a new connection every time you send a message. If you're sending and receiving multiple messages from a remote AVD, then you can keep using the same socket. This makes it easier.
- Please do not use Java object serialization (i.e., implementing Serializable). It will create large objects that need to be sent and received. The message size overhead is unnecessarily large if you implement Serializable.
- Please do not assume that there is a fixed number of messages (e.g., 25 messages) sent in your system. Your implementation should not hardcode the number of messages in any way.
- There is a cap on the number of AsyncTasks that can run at the same time, even when you use THREAD_POOL_EXECUTOR. The limit is "roughly" 5. Thus, if you need to create more than 5 AsyncTasks (roughly, once again), then you will have to use something else like Thread. However, I really do not think that it is necessary to create that many AsyncTasks for the PAs in this course. Thus, if your code doesn't work because you hit the AsyncTask limit, then please think hard why you're creating that many threads in the first place.

  This document gives you more details on the limit and you might (or might not, depending on your background) understand why I say it's "roughly" 5.

  http://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html
  (Read "Core and maximum pool sizes.")
- For Windows users: In the past, it was discovered that sometimes you cannot run a grader and Android Studio at the same time. As far as I know, this happens rarely, but there is no guarantee that you will not encounter this issue. Thus, if you think that a grader is not running properly and you don't know why, first try closing Android Studio and run the grader.