Project 3: Two-Phase Commit

Due: December 1st, 2016 23:59:59pm

In this project, you will extend the file service you implemented in Project 1 to support two-phase commit over replicated file servers. Unlike previous projects, you need to define your own RPC interface for this project using Apache Thrift. Feel free to modify the `fileservice.thrift` file you used in Project 1 as needed. This project is worth **10%** of your total score. You must use **C/C++, Java, or Python** to implement this project.

You are **strongly encouraged** to work in a group on this project. Every group can have **at most 2 students**. It is okay for a group to have one graduate student and one undergraduate student.

It is also okay if you prefer to work individually on this project. However, you will not receive any extra credit for working individually.

If you choose to work in a group, at least one member in the group **must** send the instructor and cc the TA an email listing the names and email addresses of the both members of your group by **November 17, 2016** by the end of day. If we do not receive your email by this deadline, we will assume that you prefer to work individually on this project. **No exception will be made**.

The project consists of four parts, which are described below.

1 Durable Remote File Service

In the first part, you will implement a durable remote file service. Your file service should support two methods:

writeFile given the file name and contents, the corresponding file should be written to the server.

readFile if a file with a given name exists on the server, return its contents. Otherwise, an exception should be thrown.

Unlike Project 1, you do not need to worry about the file's owner or other metadata. In other words, this is a simple key-value store with the filename being the key and the file content being the value.

Your remote file service must be durable. That is, if your process exits or is killed, when it restarts, any files that were previously successfully stored using a `writeFile` operation must still be retrievable using a `readFile` operation.

2 Two-Phase Commit

In the second part, you will implement a replicated remote file service, building on Part 1. The architecture of your replicated file service is very simple: it consists of a single "coordinator" process and multiple "participant" processes:

coordinator the "coordinator" process should expose an RPC interface to clients that contains two methods: `writeFile` and `readFile`. When the coordinator process receives a state-changing operation, i.e., `writeFile`, it uses two-phase commit to commit that state-changing operation to all participants. When the coordinator receives a `readFile` operation, it selects a participant at random to issue the request against.

The coordinator should be **concurrent**. It should allow multiple clients to connect to it and should be able to process multiple operations concurrently.

participant each participant implements a durable remote file service you built in Part 1. It exposes an RPC interface to the coordinator to participate in two-phase commit. It is up to you what other methods the participants should implement to perform two-phase commit.

The participants should also be concurrent: it should be able to process multiple commands concurrently. Thus, it needs to implement some form of concurrency control. For this project, the concurrency control scheme is very simple: if two concurrent operations manipulate different files, they can safely proceed concurrently. If two concurrent operations manipulate the same file (e.g., two `writeFile` operations to the same file show up at the same time), the first to arrive should be able to proceed while the second's two-phase commit should abort.

To implement two-phase commit, both the coordinator and the participants will need to do some form of logging to keep durable state associated with the two-phase commit. You will need to figure out how to implement logging, what to log, and how to replay your log on recovery. Your implementation needs to be fault-tolerant. If either the coordinator or one or more participants fail, your two-phase commit implementation needs to do the right thing. Similarly, after a failed component recovers, your two-phase commit implementation needs to recover and proceed as appropriate.

To let the coordinator and participants learn about the system configuration, i.e., which processes exist and where they are running, you can follow the example in Project 2. In Project 2, we used a local file `branch.txt` to store the names, IP addresses, and port numbers of all branches. You can create a similar file for this project.

3 Clients

In the third part, you should implement a client program that connects to the coordinator and issues a stream of `writeFile` and `readFile` requests. You should be able to launch multiple clients in order to drive the coordinator concurrently – i.e., each separate client should be issuing its own stream of `writeFile` and `readFile` operations.

4 Test

You should write your own test program to test your implementation of two-phase commit. We will use your test program for grading. Your test program output should be **succinct and informative**. You need to find a way to test as many corner cases as you can, such as the coordinator failing during the period of uncertainty. At a minimum, your test program should check the following cases:

1. **Durability.** After a successful commit of `writeFile`, all participants crashed. When they are restarted, a `readFile` request returns the correct file content.

2. **Concurrency Control.** If two concurrent `writeFile` requests to the same file arrive at the same participant at the same time, the later one should be aborted. Note here that concurrency control is only done at the participants, not at the coordinator.

3. **Coordinator Failure Case 1.** The coordinator failed before voting started, i.e., no voting messages were sent out. All participants time out and abort the writeFile request. When the coordinator recovers, it restarts voting and finds that the all participants have aborted.

4. **Coordinator Failure Case 2.** The coordinator failed after voting has started, and at least one participant has replied Yes. When the coordinator recovers, it first checks if it has logged the final decision. If yes, it sends out the final decision, and all participants apply appropriate changes based on the final decision. If not, the coordinator queries all participants for their decisions and make a final decision based on these decisions.

5. **Participant Failure.** One participant failed after replying Yes to commit but before receiving the coordinator's final decision. When it recovers, it asks the coordinator for the final decision. If the final decision is to commit, the participant commits. Otherwise, aborts the operation.

5 How to submit

To submit the project, you should first create a directory whose name is "your BU email ID"`-project3`. For example, if your email ID is `jdoe@binghamton.edu`, you should create a directory called `jdoe-project3`. If you are working in a group, you should include the email IDs of both group members, e.g., `jdoe-jsmith-project3`.

You should put the following files into this directory:

1. Your source code.
2. A `Makefile` to compile your source code.
3. A `Readme` file describing how to compile and run your program.
4. A brief `report` that describes:
   (a) the structure of your code, including any major interfaces that you implemented,
   (b) the RPC interface your participants expose to the coordinator,
   (c) how failures are detected, and if failures do occur, how they are reflected to clients via the RPC interface that the coordinator exposes,
   (d) test cases you have explored, how to repeat these test cases, and sample input/output.

Compress the directory (e.g., `tar czvf jdoe-project3.tgz jdoe-project3`) and submit the tarball (in `tar.gz` or `tgz` format) to Blackboard. Again, if your email ID is `jdoe@binghamton.edu`, you should name your submission: `jdoe-project3.tar.gz` or `jdoe-project3.tgz`. If you are working in a group, you should name your submission: `jdoe-jsmith-project3.tar.gz` or `jdoe-jsmith-project3.tgz`. Every group only need to submit one project to Blackboard. Failure to use the right naming scheme will result in a 5% point deduction.

Your project will be graded on the CS Department `REMOTE` computers `remote.cs.binghamton.edu`. If you use external libraries that do not exist on the `REMOTE` computers in your code, you should also include them in your directory and correctly set the environment variables using absolute path in the Makefile. If your code does not compile on or cannot correctly run on the CS computers, you will receive zero points.

Your project must be your original work. We will use MOSS[1] to detect plagiarism in the projects.

[1] `https://theory.stanford.edu/~aiken/moss/`