Final Project: Distributed KV Store

Dexter Volz, Travis Brooks, Chris Audretsch, Chun Fung Lai
https://github.com/travBrook/P435_Final_Project

Introduction

We implemented a distributed key-value store system in Python. This system includes the following consistency schemes: sequential consistency, causal consistency, eventual consistency, and linear consistency. In this report, we will describe our design, including the lower level key-value store we used, the implementation of each consistency model, and our client's API. We will also describe the main challenges we faced in this project.

Key-Value Store

We implemented our own key-value store. Each replica contains a standard Python map from keys to values. This solution is simple and effective for the constraints of this project. We preferred using our own implementation because most low-level key-value store software is supported on Linux operating systems and some of our members use Windows machines.

Program Overview

- The execution begins with the driver.py file. Driver creates an instance of the master class and calls run() on it. Finally, after pausing execution for 2 seconds, driver instantiates client node processes.
- In the master's run() method, three replica nodes are instantiated and the master creates its own socket, binds and listens on it.
- The replicas begin by setting up their own socket. Before listening on the event loop, they send a message to the master as a boot acknowledgement.
- Client nodes create a socket same as other nodes, however the client checks parameters for an input file. If there is a valid file, the client parses the inputs and sends the requests to the master. If the parameters fit the raw request naming convention, the requests will be passed to the master. After the requests have been made, the client waits patiently for a response.
- Once the master receives the request, it assigns a request ID (rID), stores the message and timestamp information, and forwards the request onto a random replica (In eventual, sequential, and linear consistency).
- The replica receives the request and calls the appropriate consistency method based on the message information.

Consistency Models

Eventual Consistency

- **Get Request:** The replica that receives the request does a lookup in its own database and if it exists it will return the value to the master. If it doesn't, it will return 'get failure' to master. In both cases, the master logs time information and forwards the message onto the original client (master handles messages from replica this way in all requests and consistencies)..
- **Set Request:** The replica that receives the request performs the set operation on its own database. It then broadcasts a message to the other replicas and sends an acknowledgement back to the master. The replicas that receive the message, will perform the operation immediately.

Linear Consistency

- Gets and sets are handled very similar in our linear consistency implementation. Once the request is received from the master, it is logged in a requests database (rID = key, message priority queue = value) and in a request priority queue. All priority levels are based on lamport clocks. After the message is logged the replica performs a totally ordered broadcast, and will not perform the get/set operation until it receives acknowledgements from the other replicas.
- When a replica receives a message from another replica, it does a lookup in the request database to determine whether the request exists or not. If it doesn't exist, we must add it to the request database and the request priority queue. If it exists, we only log it in the request database. After the logging is complete, the replica does a series of checks to determine if it needs to broadcast acknowledgements or perform the operation. If a database operation is to be performed, then we must remove the item from the request priority queue, move the request from the request database to the processed_request database, perform the operation, and check the next item in the request queue to determine if additional work is needed. If the replica is the original request receiver, it will send a message back to the master with the requested information.

Sequential Consistency

- **Set Request:** Handled in the same way as in linear consistency sets.
- **Get Request:** A get request for the sequential consistency follows the same flow as a set request but does not need to be broadcast to other replicas. Since it is inserted in the request priority queue, the replica will be certain that the get requests follow the replica's writes -- making all of its operations fall into sequential order. When it is time to perform the request, the replica will send its own value it has stored in the database, or a failure message if it does not contain the key.

Causal Consistency

- **Set Request:** The master sends all write requests only to a special replica, known as the 'primary' (in our implementation, the master itself functions as the primary). The primary thus has perfect knowledge of the temporal ordering of write requests by all clients. After receiving a write request, it *lazily propagates* the request. That is, the primary sends a write request (along with the accurate timestamp) to each other replica. Replicas keep track of the highest write timestamp they have seen. This allows for causal consistency in reads, which we will now see.
- **Get Request:** Clients keep track of additional information -- their highest read timestamp and a map from their writes to their timestamps. On sending a read request, they include a *minimum acceptable timestamp*, which is equal to the max of the highest read timestamp and the write timestamp for that particular key. The replica that receives this request checks the minimum acceptable timestamp against the highest write timestamp that they have seen. If this highest write timestamp is greater, causal consistency is guaranteed and the replica returns the value for that key. If this highest write timestamp isn't greater, causal consistency isn't guaranteed; the replica sends a message explaining this. As soon as the most recent propagation of the write request hits that replica, it will be ready to respond to that read request.

API

There are a few different ways to have a client send requests to the master controller. From the command line, the client needs its own IP, the master controller's IP, and a role for itself. Optional arguments are requests. These requests are composed of "the consistency [1-4], the request [1-2], the data of the get or set". For the consistency, 1 is linearized, 2 is sequential, 3 is causal, and 4 is eventual. For the request, 1 is set, 2 is get. For the data, sets are delimited by three colons ":::", while gets are just the key of interest. You can see examples in the input directory's files. Here are the different ways to send a message.

- **Command line (explicit message)** : When running python client.py, you can add request parameters in the format specified above. The example below shows a set request with linearized consistency. The client IP is '127.0.0.1', and the master IP is '127.0.0.10'. The role is 'Client1'.
    - -> python client.py '127.0.0.1' '127.0.0.10' 'Client1' '1, 1, key ::: value'
- **Command line (input file)** : You can create a file in the input directory, and specify the file in the command line. Here is an example that uses the linear1 file to get requests.
    - -> python client.py '127.0.0.1' '127.0.0.10' 'Client1' 'linear1'
- **Add to static variable** : If you would like every client to send a specific request, you can add a message to the messages variable in client.py at the top. There is an example next to this variable that is commented out.

Challenges The main challenge with this project came with troubleshooting the totally ordered broadcast. We kept two different data structures just to help facilitate its functionality. It was a little cumbersome to maintain upkeep as the replicas could be facing many different situations. In some scenarios this was causing too many messages to be sent, or none at all.

Example We run the driver program, which initializes a variety of clients, each with its own form of consistency.

```
PS C:\Users\audch\OneDrive\Desktop\Distributed_Systems\P435_Final_Project-master> & C:/Users/audch/Anaconda3/python.exe c:/Users/audch/OneDrive/Desktop/Distributed_Systems/P435_Final_Project-master/driver.py
```

```python
def run():

    #Spawn controller and replicas
    proc_id = subprocess.Popen([sys.executable, './master.py', config.MASTER_IP, 'Master Control'])

    time.sleep(2)
    print("\nMoving to clientel\n")

    #Spawn Client
    proc_id = subprocess.Popen([sys.executable, './causalClient.py', config.CLIENT1_IP, config.MASTER_IP,'Client1', 'causal1'])
    proc_id = subprocess.Popen([sys.executable, './client.py', config.CLIENT2_IP, config.MASTER_IP,'Client2',  'linear1'])
    proc_id = subprocess.Popen([sys.executable, './client.py', config.CLIENT3_IP, config.MASTER_IP,'Client3',  'seq2'])
    proc_id = subprocess.Popen([sys.executable, './client.py', config.CLIENT4_IP, config.MASTER_IP,'Client4',  'linear2'])
    proc_id = subprocess.Popen([sys.executable, './client.py', config.CLIENT5_IP, config.MASTER_IP,'Client5',  'event1'])
    proc_id = subprocess.Popen([sys.executable, './causalClient.py', config.CLIENT6_IP, config.MASTER_IP,'Client6',  'causal2'])
    proc_id = subprocess.Popen([sys.executable, './client.py', config.CLIENT7_IP, config.MASTER_IP,'Client7',  'event2'])
    proc_id = subprocess.Popen([sys.executable, './client.py', config.CLIENT8_IP, config.MASTER_IP,'Client8',  'causal2'])
```

The clients' actions are specified in text files, which look like this:

```
4, 1, key1 ::: first
4, 1, key2 ::: second
4, 1, key3 ::: third
4, 1, key1 ::: firstOW
4, 1, key1 ::: firstOOW
4, 2, key1
4, 2, key3
4, 2, key2
4, 2, key1
```

Clients act asynchronously:

```
Client3 sent a request!
Client2 sent a request!
Client2 sent a request!
Client4 sent a request!
Client3 sent a request!
Client2 sent a request!
Client4 sent a request!
```

And hear back from the master:

```
Client1 received the follow:  ip: "127.0.0.11"
consis: CAUSAL
request: SET
ack: OK
data: "key2 ::: second"
l_Clock: 70.11
rID: 23
```

```
Client5 received the follow:  ip: "127.0.0.11"
consis: EVENTUAL
request: GET
ack: OK
data: "second"
l_Clock: 196.11
rID: 45
```

```
Client2 received the follow:  ip: "127.0.0.11"
consis: LINEARIZED
request: GET
data: "REQUEST FAILURE"
l_Clock: 289.11
rID: 11
```

Supplementary Information
Process logs and statistics can be found in the ‘log’ folder of the repository.