Design a Server-side File-Processing Program

Zichen Nie
znie@usc.edu

Part One: Basic Assumptions

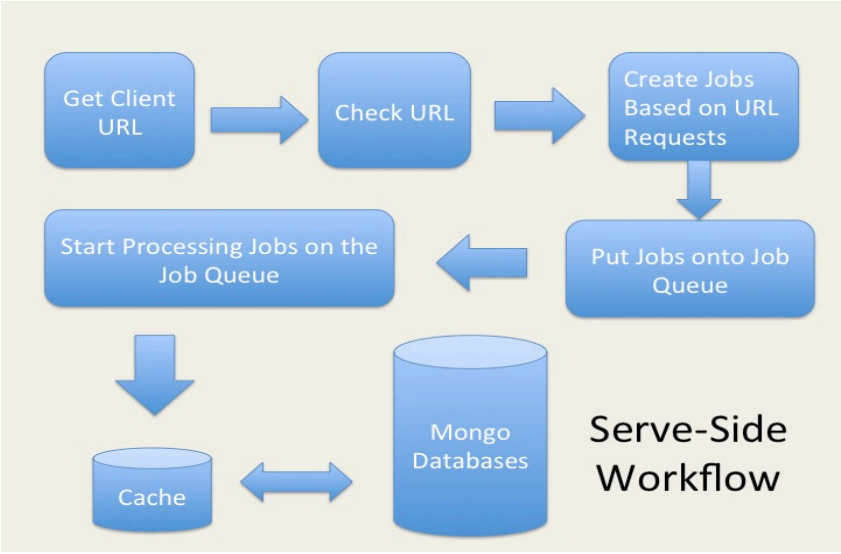
The problem instruction only specified the main function of the server-side program and the file format it should operates on. Based on these two points, I made the assumptions below to make it more detailed and simplified to some extent:

1. The JSON format files (objects) are stored in NoSQL databases on the server, like a Mongo Database, that could handle queries based on <key, value> pairs in the JSON objects.
2. The clients send file operation requests to server by constructing URLs that could specify all the information the server need to go for next step.
3. Clients have different permissions to read, update or delete files on the server.
4. Each request only contains one operation, it could be on one or multiple files but the operations should be the same to all of the files.

Part Two: Infrastructure Design

The whole file processing system follows Client-Server Architecture. The server could provide a RESTFUL service that receive the client's URL, processed it, and return client the JSON object to visualize the successful operation.

Below is a workflow for processing a single request from client on the server:



Let me briefly explain the workflow showed above.

1. Get client URL. The client sent URL to server to request operations on certain files or parts of files on the server.
2. Check URL. This process checks if the client exists, if the client has the permission to operate on the request file and if the file exists on the server. If any of the three criteria cannot be satisfied, the program should return a warning message to the client.
3. Each request is regarded as a Job in the program. Each Job contains a set of features that can uniquely identify the Job, such as clientID, referenced Files, Operation Type and Progress Flag. I will explain each of the features later in the implementation plan.
4. Jobs are stored as a Queue on server, in a multi-thread environment, there might be many server threads working at the same time. The Job Queue is a resource that shared by all the threads, therefore we should prevent the conflicts and deadlock.
5. Server start operating on the jobs, with three kinds of operations: read, update and delete. One file cannot be updated or deleted by more than two threads at the same time, but multiple threads can read the same file simultaneously.
6. Ideally, in case of large amounts of time spending on fetching JSON objects from Databases, we should create a cache for fast access to the most frequently or most recently fetched files to improve proficiency.
7. If requested file object cannot be found in cache, get to database to find it.

Part Two: Implementation Plan

There are three kinds of objects on the server:

1. URL . It provides four fields:
   - userID: the identity of the client who send the request
   - resourceID: the identity of the files the clients need to operates on
   - <keys, values>: the field of the files the clients need to operates on

   A sample URL is below, notice that the key and value should be in pairs and in order.

   url='http://localhost:8000/?userID=54635&operator=read&resourceID=1002,1004,1007&key=name,author&value=Truth,Chris'

2. Job Object
   - id: identify the file to operate
   - tag: indicate the file is in editing or not, if is in updating, no other thread can

touch the file. This is used in a muti-threaded environment.

- operator: specify the operations, read, update or delete
- keysArray: the fields in the JSON object needed to be read, updated or deleted.
- valuesArray: the value of the keys in keysArray need to be read, updated or deleted.
- userId: the client who request the job

3. Server Object
   - dataBase: the database object fetched from back-end Mongo Database
   - cachedFiles: the most recent used files stored here
   - fileQueue: the job stored on this fileQueue is waiting to be operated on
   - getURL(self, urlString): get the URL from client and format the url into an object that is easy to operate on.
   - startOperation(self, resArray, keysArray, valuesArray): start processing jobs on the fileQueue
   - checkPriority(self,userId, operation, resources, obj): check request from client is valid and has the authority to the operation or not, if yes, create a new job on the fileQueue
   - read(self,job, keysArray, valuesArray): perform read operation on job
   - update(self,job, keysArray, valuesArray):perform update operation on job
   - delete(self,job, keysArray, valuesArray): perform delete operation on job

4. Cache Object
   I did not have the time to implement this part, so I will explain this in detail in Part Four.

Part Three: Test Cases and Demos

In order to deliver the demo, I created a database in MongoDb call DolbyDB with two collections: Usertable and Filetable, the simplest version of a file database. I also created two simple Json files(User.json and files.json) and imported them into the DolbyDB, which correspondes to the Usertable and Filetable. In MongoDB, all documents are stored as JSON Objects so it is relatively easy to operate on.
Sample User.json:

```
[
{"userID" : 54635, "name" : "Ada","permission" : 111},
{"userID" : 54636,"name" : "Alice","permission" : 100},
{"userID" : 54637,"name" : "Ben","permission" : 0},
{"userID" : 54638,"name" : "Den","permission" : 110},
```

{"userID" : 54639,"name" : "Zoe","permission" : 111}]

Sample files.json

[

{"fileID" :1002,"name" : "Gone With the Wind","author" : "AdaLee","price":"30

{"fileID" : 1003,"name" : "Greedy Algorithm","author" : "CLS","price" : "27"}

{"fileID" : 1004,"name" : "Truth","author" : "Chris","price" : "28"},

{"fileID" : 1005,"name": "Computer System","author" : "Pros Den.","price" :"59"},

{"fileID":1006,"name":"Alice in the Wonder Land","author" : "Alice","price" :"10"},

{"fileID" : 1007,"name" : "Truth","author" : "Chris","price" : "33"},]

Test Case 1:

Read file 1002, 1004, 1007 with book name = Truth, author = Chris

url='http://localhost:8000/?userID=54635&operator=read&resourceID=1002,1004,1007&key=name,author&value=Truth,Chris'

```
I'm reading 1002
The query condition is ...
{'author': 'Chris', 'name': 'Truth', 'fileID': 1002}
[]
I'm reading 1004
The query condition is ...
{'author': 'Chris', 'name': 'Truth', 'fileID': 1004}
[{u'price': u'28', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467784'), u'name': u'Truth', u'fileID': 1004}]
I'm reading 1007
The query condition is ...
{'author': 'Chris', 'name': 'Truth', 'fileID': 1007}
[{u'price': u'33', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467787'), u'name': u'Truth', u'fileID': 1007}]
```

As you can see, the first and second queries returned null because there is no file match the query.

Test Case 2:

Read all the file with book name = Truth, author = Chris

url='http://localhost:8000/?userID=54635&operator=read&resourceID=0&key=name,author&price=Truth,Chris'

```
I'm reading 0
The query condition is ...
{'name': 'Truth', 'author': 'Chris'}
[{u'price': u'28', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467784'), u'name': u'Truth', u'fileID': 1004}, {u'price': u'33', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467787'), u'name': u'Truth', u'fileID': 1007}]
```

As you can see, there are two files match the condition.

Test Case 3:

Update file 1004, 1007, change their book name to Truth , book price to 35

url='http://localhost:8000/?userID=54635&operator=update&resourceID=1004,1007&key=name,price&value=Truth,35'

```
I'm updating 1004
Before update...
[{u'price': u'28', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467784'), u'name': u'Truth', u'fileID': 1004}]
After update...
[{u'price': u'35', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467784'), u'name': u'Truth', u'fileID': 1004}]
I'm updating 1007
Before update...
[{u'price': u'33', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467787'), u'name': u'Truth', u'fileID': 1007}]
After update...
[{u'price': u'35', u'author': u'Chris', u'_id': ObjectId('54e9ba9d653589bdc0467787'), u'name': u'Truth', u'fileID': 1007}]
```

Test Case 4:

Delete file 1007.

url='http://localhost:8000/?userID=54635&operator=delete&resourceID=1007'

```
I'm deleting1007
The query condition is ...
{'price': '33', 'name': 'Truth', 'fileID': 1007}
[]
```

After delete, the Mongo DB console shows nothing deleted because noting satisfies the condition.

Test Case 5:

Delete all the files with author name = Truth, book price of 35.

urlString_5='http://localhost:8000/?userID=54635&operator=delete&resourceID=0&key=name,price&value=Truth,35'

```
I'm deleting0
The query condition is ...
{'price': '35', 'name': 'Truth'}
[]
```

After delete, the Mongo DB console shows two files deleted because there are two files match the condition, after we have updated the price at TestCase 3.

Part Four: Further Thinking and Improvements

- To make the operations of read, update and delete more extendable and maintainable, we can use Strategy Design Pattern, add a interface for the set of operations.
- Cache design: main structure of the cache is a priority queue, apply the Least Recent Used strategy, the priority is associated with userId.
- Multi-thread design: I have no time to implement the multi-thread version, but it's important to prevent dead lock and conflicts on Job Queue. And in real world application, multi-threading is inevitably add both efficiency and complexity to the design.
- To speed up response to client, there might be distributed servers across internet to serve the clients all over the world. Choose a good routing technique is critical to save time and space, which means to save money.