Distributed Systems: Lab 1
Client-Server Database

Petru Eles and Ivan Ukhov
petru.eles@liu.se and ivan.ukhov@liu.se

January 24, 2016

1 Introduction

In the previous lab, you built a non-distributed system composed of a number of standalone machines, and each such machine was responsible for both the server and client components of your fortune database. You observed that the private copies of the data maintained by the machines could easily become inconsistent as there were no attempts of synchronization. The goal of the current lab is to mitigate this problem by utilizing the client-server model [1] of distributed systems. Specifically, we shall designate the server role to one separate machine and turn the rest into pure clients. This scenario is depicted in Figure 1; compare it with the one from Lab 0 [2].

2 Communication Protocol

In order for the server and clients to be able to interact with each other, they need to agree upon a communication protocol. In this programming project, the communication protocol has the following specification.

Each message is encoded in the JSON format [3, 4] and sent as a single line, which ends with a new-line character. Whenever a client wants to invoke a function on the server side, the client sends a message to the server using the following scheme:

```
{
    "method": method_name,
    "args": method_arguments
}
```

The `method` field contains a string with the name of the function that the client wants to call, and the `args` field contains an array of arguments that that function takes. For example, in order to add a new fortune into the database of the server, a client might send the following message:

```
{
    "method": "write",
    "args": ["Take it easy"]
}
```

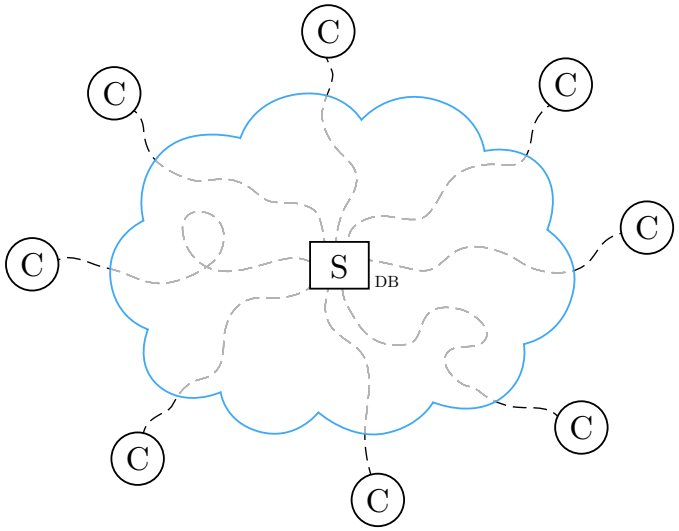

Figure 1: A distributed database with a single server (S), maintaining a database (DB), and a number of clients (C).

meaning that the method to invoke is called "write," and "Take it easy" should be passed as the first and the only argument of that method. As noted earlier, it is important to send each message as a single line terminated by a new-line character; make sure you follow this convention when writing into the socket.

If a method invocation is successful, the server sends an acknowledgment to the corresponding client using the following scheme:

```
{
    "result": method_result
}
```

where the `result` field contains the output of the executed function encoded in the JSON format; the whole message is a single line. Note that the presence of this field is sufficient to conclude that the call was successful, and the value of the field can be `null` if the corresponding function returns nothing.

If a method invocation fails due to an error, the server replies with a message having the following structure:

```
{
    "error": {
        "name": error_class_name,
        "args": error_arguments
    }
}
```

The occurred error is encoded in `error` in such a way that it can be instantiated and raised on the client side; in fact, it is one of the requirements to your implementation. At this point, it is worth recalling that an exception in `Python` is an instance of some exception class. In order to create such an instance, one has to know the name of the exception class and the arguments that the class'

constructor requires. With this in mind, the `name` field delivers the class name of the occurred exception while `args` delivers an array of the parameters to be passed to the constructor of the class.

3 Your Task

3.1 Preparation

Continue working with the same code base that you have been working with so far, including all the changes that you have made. The files relevant to this lab are listed below. You should read and understand them.

- `lab1/client.py` — the `Client` application (should be modified);
- `lab1/server.py` — the `Server` application (should be modified);
- `lab1/test.sh` — a shell script that you can use for testing;
- `lab1/dbs/fortune.db` — the same as for Lab 0 [2];
- `modules/Common/wrap.sh` — an auxiliary script needed for `test.sh`;
- `modules/Server/lock/readWriteLock.py` — the class utilized by `Server` for the purpose of synchronization as motivated in Section 3.2 (no changes are needed);
- `modules/Server/database.py` — the same as for Lab 0 [2] (should contain your previous changes).

Carefully read the source code and understand what it is doing. Pay your attention to the missing parts marked with "Your code here." In order to run the code, open two terminal windows and change the current folder to `lab1`. In one of the terminals, issue the following command in order to run the `Server` application (recall that the server is unique in this lab):

```
$ ./server.py
Listening to: c-204-13.eduroam.liu.se:45725
Press Ctrl-C to stop the server...
```

The code randomly chooses a port number to listen to and prints it out along with the hostname of the server machine. In the example given above, the hostname is `c-204-13.eduroam.liu.se`, and the port number is `45725`. As you have already discovered by studying the code in `server.py`, you can choose a port number yourself by specifying the corresponding argument in the command line. Now, let us create an instance of the `Client` application by running the following command in the second terminal:

```
$ ./client.py -i c-204-13.eduroam.liu.se:45725
Choose one of the following commands:
    r            ::  read a random fortune from the database,
    w <FORTUNE>  ::  write a new fortune into the database,
    h            ::  print this menu,
    q            ::  exit the application.
```

```
Command> w Keep it simple.
Command> r
None
Command>
```

Note that you have to specify the address that your server is using, and it can be a different address each time you restart the server. Regardless of the correctness of the address, the system does not work at this stage since neither `Server` nor `Client` has the corresponding communication part implemented.

In order to ease the testing procedure of this and the future labs, we have implemented a set of shell scripts that open the needed terminals and run the corresponding commands for you. These scripts are called `test.sh` and can be found in the main folder of each lab. For instance, the test script for the current lab is `lab1/test.sh`. It is important to note, however, that you should be able to set up everything yourself without the help of these scripts. So, have a look at the content of the scripts and make sure you understand what they do. You can also modify these scripts as you wish.

3.2 Communication

Using `Python`'s sockets [5], complete the implementation of `server.py` and `client.py`. Your code should satisfy the following requirements:

- Both `Server` and `Client` should implement and follow the communication protocol described in Section 2.
- The server should process each client in a separate thread such that it does not stop serving other incoming connections. Your implementation should be thread safe as motivated below.
- The server should be robust: it should catch and adequately respond to exceptions including those due to potential network problems, violations of the protocol, invocations of non-existing functions, invocations of proper functions but with wrong arguments, etc.
- Whenever a client receives an error from the server, it should instantiate and raise it as if the error has occurred locally. The user of the `Client` application should not feel any difference between the non-distributed and distributed versions of the system.

Since the server can have several threads serving client requests in parallel, the operations on the database issued by the clients can interfere with each other and leave the database in a corrupted state. In order to prevent this behavior, you should protect the critical parts of your code with a read/write lock provided to you in `readWriteLock.py`. Think where the locking should take place and how to make it efficient in terms of the waiting time.

Lastly, make sure that, in spite of all possibly occurred exceptions, the server keeps on working until it is explicitly stopped by the user.

4 Conclusion

In this lab, you have implemented a rather naïve distributed system: there is only one server and each client is required to know the current address of the

server. The former issue makes our distributed system both unreliable (e.g., your carefully written code will not protect the server against a blackout) and inefficient (e.g., if there are many clients, the workload of a single server machine can be substantially high making clients wait for a response for a long time). The second issue makes it inconvenient for the clients to connect to the server as the hostname and/or port of the server might easily change, and the clients need to keep track of such changes. This inconvenience will become especially bothering when you allow your distributed system to have several servers; therefore, in the next lab, you will address this problem first.

References

[1] http://www.ida.liu.se/~TDDD25/lectures/lect2.frm.pdf.

[2] https://gitlab.ida.liu.se/tddd25/labs/raw/master/doc/lab0.pdf.

[3] http://en.wikipedia.org/wiki/JSON.

[4] http://docs.python.org/3/library/json.html.

[5] http://docs.python.org/3/library/ipc.html.