Distributed Systems: Assignment 1
Client-Server Database

Petru Eles, Sergiu Rafiliu, and Ivan Ukhov
{firstname.surname}@liu.se

January 14, 2013

1 Introduction

Recall that, in the previous assignment, you built a non-distributed system composed of a number of standalone machines, each of which comprised both the server and client parts of our fortune database. You observed that the private copies of the data, which the machines possessed, could easily become inconsistent as there were no attempts of interaction. The goal of the current assignment is to mitigate this problem by utilizing the client-server model [1] of distributed systems. Specifically, we shall designate the server role to a separate machine rendering the other ones as pure clients. This scenario is depicted in Figure 1; compare it with the one for the zeroth assignment.

2 Your Task

2.1 Preparation

Download [2] and extract the source code for the assignment. In the archive, you will find the following files:

- `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 Assignment 0;
- `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 detailed in Section 2.3.
- `modules/Server/database.py` — the same as for Assignment 0 (overwrite with your implementation);

Carefully read the source code and understand what it does. Pay your attention to the missing parts marked with comments saying "Your code here". In order to run the code, open two terminal windows and change the current folder to

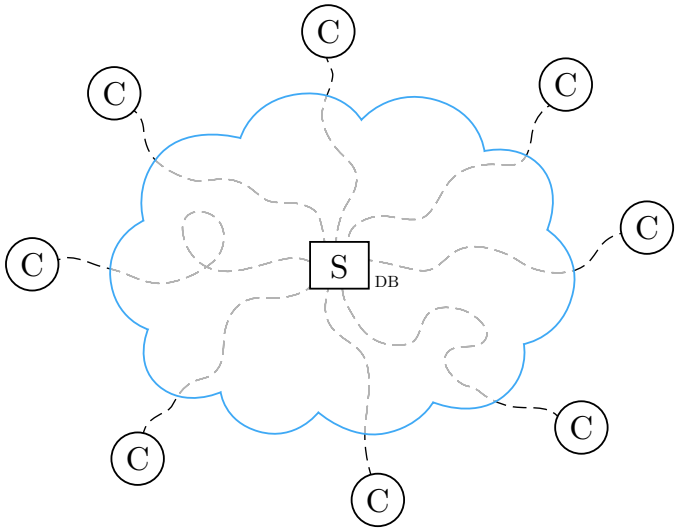

Figure 1: A distributed database with a single server (S), maintaining a database (DB), and a number of clients (C).

`lab1`. In one of the terminals, issue the following command in order to run the server (recall that the server is unique in this assignment):

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

The code randomly chooses a port to listen to and outputs this port along with the host name of the server machine; in this case, the host name is `c-204-13.eduroam.liu.se` and the port is 45725. As you saw in `server.py`, you can also choose a port yourself by providing 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:

```
$ python 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 here you should 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.

To automate the testing procedure of this and the following assignments, we have implemented a set of shell scripts—all are stored in `labN/test.sh` where `N` stands for the assignment number—that open the needed terminals and run the corresponding commands. Have a look inside the first of them, `lab1/test.sh`; you might find it helpful. You can also modify these scripts as you wish.

2.2 Communication

Using `Python`'s sockets [3], complete the implementation of `server.py` and `client.py`. The following are the requirements to your code:

- The server should process each client in a separate thread such that it does not stop serving other incoming connections. For now, leave synchronization issues aside as they will be address separately in Section 2.3.
- Both sides, `Server` and `Client`, should implement the communication protocol described in the code and outlined below.
- The server should be robust enough to resist the errors that might arise (network problems, violations of the protocol, invoking non-existing functions, invoking methods with wrong arguments, *etc.*).

The communication protocol is as follows. All messages are encoded in the JSON format [4, 5]. Whenever a client wants to invoke a function on the server side (*e.g.*, in order to add a new fortune), it sends a message to the server using the following specification:

```
{    'method': called_method,
     'params': called_methods_params
}
```

where `method` is the name of the method that the client wants to call, and `params` is a list of arguments for this function (*e.g.*, the text of a new fortune). If the client request is processed successfully, the server sends a message to the client using the following format:

```
{    'result': called_methods_result    }
```

where `result` is the output of the executed function encoded in JSON. On the other hand, if an error occurs while serving the request, the server replies with a message using the following scheme:

```
{    'error': {    'name': error_class_name,
                   'args': error_args          }
}
```

where `error` is the occurred error encoded in such a way that it can be restored and raised on the client side. As a consequence, `error` should contain the class name of the error, stored in `name`, and a list of arguments, stored in `args`, to create an instance of this class. Make sure that, in spite of all possibly occurred errors, the server keeps on working until it is stopped explicitly by the user.

2.3 Synchronization

Since there can be several threads within the server 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. 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 impose the least overhead.

3 Conclusion

In this assignment, you have implemented a rather naïve distributed system: there is only one server and the clients are required to know the current server' address and port. 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 leads to inconvenience for clients as the host name and/or port of the server might easily change, and all the clients should get to know about this change somehow. This inconvenience will become especially bothering when you allow your distributed system to have several servers; therefore, in the next assignment, you will address this problem first.

References

[1] http://www.ida.liu.se/~TDDD25/lecture-notes/lect2-3.frm.pdf.

[2] http://www.ida.liu.se/~TDDD25/labs/assignment1.zip.

[3] http://docs.python.org/2.7/library/ipc.html.

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

[5] http://docs.python.org/2.7/library/json.html.