Com S 454/554 Fall 2021 Assignment 7

Due: Sunday December 1

Total points: 150 (each member of a group) or 200 (individual)

In this assignment, we will develop a simplified Primary-based Remote-write protocol (refer to the lecture and lecture notes) to implement sequential consistency for distributed data store. You are suggested to use Java for the project; but you can also use python or C/C++. Specific requirements are as follows.

Note: You are allowed to work individually, or in a group of two students and submit only one version. In the case of working as a group, each one can earn up to 150 points, and you need to specify each one's work (i.e., who develop which functions) honestly in your submission. In the case of working individually, one can earn up to 200 points. Please be reminded that, do not copy efforts from others (including other groups in the class, or other resources other than general resources and you acknowledge).

How do the data store servers work?

You are required to develop a distributed data store replicated at one primary server and multiple backup servers. For simplicity, the data store maintains just one integer variable with initial value of 0; the servers will run on the same computer as different processes.

The primary server

When the primary server starts, it takes one integer argument: *PrimaryPort*. Then, it (1) sets up an integer variable (i.e., the primary replica of the data store), (2) creates a TCP server socket with port *PrimaryPort*, and (3) waits at the port for requests from clients or backup servers. It needs to handle the following types of requests coming to the port:

- **READ** request from client – When receiving this request, it should return the current value of its replica of the data store variable.
- **WRITE:newValue** request from client – When receiving this request, it should (1) update its own replica of the data store to **newValue**, (2) request each of the backup server to update its replica of the data store and get acknowledgement from it, and then (3) reply to the requesting client. Note that, if multiple WRITE requests are received, they should be executed sequentially for sequential consistency.
- **JOIN:backupPort** request from a backup server – When receiving this request, it should (1) record the requesting backup server's server socket port number (i.e., **backupPort**), and then (2) send acknowledgement to the requesting backup server.
- **UPDATE:newValue** request from a backup server -- When receiving this request, it should first act the same as receiving **WRITE:newValue** to have all the data store replicas to update to

newValue. Then, it acknowledges the requesting backup server of the completion of update. Note that, if multiple WRITE/UPDATE requests are received, they should be executed sequentially for sequential consistency.

Each backup server

When a backup server starts, it takes two integer argument: **BackupPort** and **PrimaryPort**. Then, it (1) sets up an integer variable (i.e., a backup replica of the data store), (2) sends a **JOIN:BackupPort** request to the primary server's server socket at port **PrimaryPort**, creates a TCP server socket at port **BackupPort**, and (3) waits at port **BackupPort** for requests from clients or the primary server. It needs to handle the following types of requests coming to the port:

- **READ** request from client – When receiving this request, it should return the current value of its replica of the data store variable.
- **WRITE:newValue** request from client – When receiving this request, it should send **UPDATE:newValue** request to the primary server, and waits for its acknowledgement. Then, it sends acknowledge to the requesting client. Note that, when multiple WRITE requests are received, they should be executed sequentially (one after another) for sequential consistency.
- **UPDATE:newValue** request from the primary server -- When receiving this request, it should update its replica of data store to **newValue** and then reply acknowledgement to the primary server.

The Client

A client program name Client.java is provided (find it at the attachment) to provide a command line interface to interact with the above servers. Once started, it accepts and passes commands you input to the servers accordingly. Examples:

- If you type command:

  5000 READ

  The client sends request **READ** to the server at port 5000, gets it executed, and prints the returned value.
- If you type command:

  5000 WRITE:10

  The client sends request **WRITE:10** to the server at port 5000, gets it executed, and prints the acknowledgement from the server.

Note that, you can run multiple instances of the program to act as multiple clients sending commands concurrently.

Other requirements

You are required to implement the server program(s) in Java and submit the source Java code well-documented. Though it is okay to submit primary server program and backup server program developed separately, it is more efficient to develop just one program that can run primary server (when only one argument is provided) or backup server (when two arguments are provided).

Make sure your program compile and run correctly on pyrite.cs.iastate.edu.

Score allocation:

1. Your programs can be compiled on pyrite.cs.iastate.edu – 20%; if cannot be compiled, receive 0.
2. Implementation of Primary Server – 40%
    a. Setup – 10%
    b. READ – 5%
    c. WRITE – 10%
    d. JOIN – 7.5%
    e. UPDATE – 7.5%
3. Implementation of Backup Server – 30%
    a. Setup – 10%
    b. READ – 5%
    c. WRITE – 7.5%
    d. UPDATE – 7.5%
4. You have reasonably comment your code – 10%

An Example

The following example show how the programs are expected to run:

Suppose you submit a program DataServer.java (and maybe together with some helping java files), which is compiled to DataServer.class.

First, start the primary server at a window:

> java DataServer 5000

```
I am the primary!
Data Server is listening on port 5000
```

Second, start two backup servers in two separated windows:

>java DataServer 5001 5000

```
I am a backup with port: 5001
Primary port: 5000
Just send out: JOIN:5001 to port: 5000
Got response: COMPLETE_JOIN
Data Server is listening on port 5001
```

>java DataServer 5002 5000

```
I am a backup with port: 5002
Primary port: 5000
Just send out: JOIN:5002 to port: 5000
```

```
Got response: COMPLETE_JOIN
Data Server is listening on port 5002
```

Third, run the client to interact with the servers in another window as follows:

>java Client

```
Enter your command:
5001 READ
port: 5001  command: READ
Just send out: READ to port: 5001
Got response: COMPLETE_READ:0
Enter your command:
5002 WRITE:11
port: 5002  command: WRITE:11
Just send out: WRITE:11 to port: 5002
Got response: COMPLET_WRITE
Enter your command:
5001 READ
port: 5001  command: READ
Just send out: READ to port: 5001
Got response: COMPLETE_READ:11
Enter your command:
5001 WRITE:22
port: 5001  command: WRITE:22
Just send out: WRITE:22 to port: 5001
Got response: COMPLET_WRITE
Enter your command:
5002 READ
port: 5002  command: READ
Just send out: READ to port: 5002
Got response: COMPLETE_READ:22
Enter your command:
```

(Note: when you run your programs on pyrite, try to use port numbers different from the above to avoid interference with your classmates.)

A programming tip:

- To ensure WRITE/UPDATE operations are not interleaving at a server, you can make the method that conduct such operations as “synchronized”.