Programming Assignment 2

UIUC CS 425 / ECE 428, Spring 2016, MP2, due April 2nd, 2016 by 7:00 p.m.

# Key-value Store Consistency

In this assignment, you will implement a distributed key-value store with various options for its consistency model. You will implement a replica server program that will store the values and carry out the consistency logic, along with a client that will issue the put/get operations to a replica. Your replica program will need to implement the linearizability and eventual consistency models.

For eventual consistency, you should accept various values of 'W' and 'R'. W is the number of replicas who should accept the value written before the write operation is considered complete. R is how many replicas to retrieve values from when reading; the value that has the most recent timestamp should be chosen. The eventual consistency mode should implement "last writer wins" logic.

You should specify the model to be used, including W and R, with command line arguments (command line format is up to you).

## Replica Server

We will be reusing the config file of MP1 to set channel delays; take that file's location as a command line argument. You can (and should) reuse the simulated-delay code you wrote in MP1. The replica server does not need to read stdin. Your replicas should be able to communicate with clients (described later), and should log those client interactions to an output file (described now). Your system must support up to 9 replicas. When testing eventual consistency, a single replica might crash (we will just kill the replica's process), and your system should continue working correctly. Any clients connected to the crashed replica should then connect to the replica with the next higher ID.

## Output file format

Each replica should record the results of all of the put/get operations that clients ask it to perform, in a log file. You can name these files whatever you like, just make sure to number them by process id. For instance, process 3 might write to `output_log3.txt`. There is a tool that will let you visualize your system in the same format that has been used on the slides. To work with that tool, use the following format for your log files:

Every read or write operation gets two lines: one when it is initiated (`req`), and one when it finishes (`resp`). Each line has 7 fields, separated by commas:

1. (Unused field; pick any number and use it in every line)
2. Integer ID of the client who made the request.
3. Either `get` or `put`
4. Variable name.
5. Timestamp.
6. Either `req` or `resp`
7. Value (empty for the `req` phase of a `get`)

Some examples:

```
555,2,get,y,1456940000000,req,
555,2,get,y,1456940000400,resp,4
555,0,put,x,1456940006000,req,7
555,0,put,x,1456940006250,resp,7
```

The timestamp should be in milliseconds since the Unix epoch. You only have to handle variable names a-z (i.e. a single, lowercase letter), and the variables can only take (integer) values of 0-9.

You must follow this format exactly: comma to delimit the fields, no spaces around the commas.

# Client

In addition to the replica server program, which implements the consitent key-value store, you'll be writing a program that makes use of that key-value store. This client should take instructions on stdin, and communicate with the server over TCP. Your client should take on its command line the ID of the replica to connect to. Once connected, it should accept on stdin the following commands:

- `put var_name value`
- `get var_name`
- `delay milliseconds`
- `dump`

`put` tells the replica to write the value to the variable. `get` request the variable's value from the replica, and should print the result to stdout. `delay` should pause the client's execution for the specified duration. `delay` is useful when you want to give the client a batch of commands in a textfile on stdin. `dump` should instruct the replica to dump its current state for all variables (**not** a consistent read operation, just its own data) to the replica's stdout. Please follow this format exactly! Here is an example sequence of commands:

```
get y
get x
put x 2
delay 400
get x
dump
```

The client is retrieving the value of y, then the value of x, then it is storing 2 into x, then it waits 400ms, then retrieves the value of x. Once it has that value it tells its replica to dump.

**Client-replica protocol:** We are also asking you to follow a specific format for communication between the client and the replica (which should be over TCP).

- The client's `put` requests are 3 bytes: ASCII 'p', followed by the variable name, followed by the ASCII of the digit to be stored. For instance, inputing `put x 2` to the client's console should result in the ASCII string "px2" being sent.
  - The replica's response (after the write has finished) should be ASCII 'A', for ACK.

- The client's `get` requests are 2 bytes: ASCII 'g', followed by the variable name. For instance, `get x` should result in the ASCII string "gx" being sent.
  - The replica's response should be the value read, again as an ASCII digit.

- `dump` requests are just 'd'. The response is 'A' for ACK, once the dump is complete.

- `delay` is just an instruction to the client; there is no communication with the replica.

## Administrative details

It should be possible to add multiple clients, with multiple clients connected to each server.

**This project should be performed by a team consisting of two students**. Any exceptions to that team size require the instructor's approval.

You may use C/C++, Java, or Python. Upload a compressed archive of your code to Compass. Include in your submission a PDF report with an overview of how you implemented the various consistency models, and what part of the project each partner did.

## Grading rubric

- 55% for correctness of the consistency models.
- 10% report
- 10% replicas' logs feed into the visualization tool correctly
- 10% client-replica protocol is followed as specified
- 5% basic code cleanliness (good comments, reasonable organization and design).
- 10% questions during demo.