# cs425-mp2

C++ Key/Value store implementation. Supports linearizability and eventual consistency models.

## Directory Purposes

`src` — Holds all of the `.cpp` files that we wrote

`include` — Holds all of the `.h` files that we wrote

`build` — Holds all of the `.o` files that are used when compiling

`bin` — Holds all of the `.exe` files that are used to execute the search program

## How to Use

The following command will start a process with supplied `pid` using the `port` and `ip` listed in `multicast.config`. The second argument chooses the consistency model (linearizable or eventual). R and W specify how many processes to read from for a get request and how many processes to write to for a put request.

```
./bin/runner.exe <pid> <consistency_model=[linearizable,eventual]> <R> <W>
```

The linearizable model requires the user to start up a sequencer along with the n processes as follows:

```
./bin/runner.exe <pid> linearizable s
```

To PUT a value to K/V Store:

```
put <key> <value>
```

To GET a value from K/V Store:

```
get <key>
```

To pause the client's execution for *n* milliseconds:

```
delay <ms>
```

To print local values of all keys for client:

```
dump
```

## Implementation Details

### MP 1 Implementation

Network delay simulation was accomplished by spawning n-1 threads (one for each process receiving the message) and sleeping them for a random duration of time bounded by `min_delay` and `max_delay` before sending. Note that sending to yourself doesn't incur any simulated delay. For causal ordering, we implemented the vector timestamp based algorithm outlined in the textbook on page 657. This method uses a hold-back queue to avoid delivering a message until it has delivered all messages causally preceding it. To multicast a message to group g, process p adds 1 to its entry in the timestamp and multicasts the message with the timestamp to group g. To meet the causal constraints, $p_i$ will wait until it has delivered any earlier message sent by $p_j$ and it has delivered any message that $p_j$ had delivered at the time it multicast the message. Total ordering was implemented with the help of a sequencer. A process wishing to multicast a message will append a unique identifier (id) and pass the message along to the sequencer as well as members of g. The sequencer is implemented as its own process, and it is responsible for maintaining a group specific sequence number and assigning increasing and consecutive sequence numbers to messages it delivers. The sequencer announces messages by multicasting the ordered messages to group g.

### MP 2 Implementation

The key/value store was built on the existing multicast implementation from mp1. In order to achieve linearizability, the total ordering scheme was used with a sequencer to ensure that all messages are received in the same order; this implies linearizability by preserving the global order of non-overlapping operations. Each process stores its own copy of the data in a hashtable mapping the key (all chars) to a tuple containing the value and timestamp. To achieve eventual consistency, the "last writer wins" approach was used with writing processes providing timestamps to discern order. With R+W > #nodes, this guarantees that the most updated value will always be read due to the overlap between reading and writing proceesses. This implementation multicasts to W-1 or R-1 other processes as well as itself when reading or writing. The timestamps are recorded using system time from the start of the lifecycle of each process.

## Partner Work

We worked on most parts of the project together. The framework/format for sending key/values was created by Jon as well as the implementation of the linearizable consistency model. Vish implemented logging and the modifications to support eventual consistency models.