Implementing Causal Consistency

- Abhiram Sarma (956337515) & Rahul Shevade (934974674)

High-Level System Description

This system implements Causal Consistency when messages need to be replicated across multiple servers and there is a causal relation between the messages. Each message has a dependency associated with it depending on the causal order of messages and represented by vector clocks present on the clients and the server.

This is implemented by a key value store structure, where each client interacts with a particular key and can either perform a read or a write operation. Each client has its own primary server and performs the operation on it directly. It is the primary's job to replicate the operations onto other existing servers. Another client may read/write on key and expects causal consistency.

Protocol

A client increments its vector clock by 1 whenever it generates a message write on the server. It sends the updated value and its own vector clock to its server. Once the server makes sure that this write has its dependencies resolved, it can update the key/value store. And then replicate this message to other servers. Server checks for dependency by the following algorithm:

1) Check if this write from the client has timestamp just 1 apart from the index in its own vector clock
2) Every other client's write requests occurring before this write have already been implemented (every other index in the message vector clock is same as its own).

This ensures that messages are delivered in a causal order. If the conditions aren't met, the server's thread waits until some other thread receives this message.

Program Structure

The program is divided into client, server and util files. Common functionalities like socket programming code, vector clock handling, dependency checking are done in the util file. Client exposes write() and read() APIs. Server handles the incoming network request and writes/reads its own data base and may replicate the message. The server supports multithreading, so it doesn't block a client. Bytes over the socket are deserialized to requests and responses.

Working Parts

All basic requirements of the assignment work. All the prescribed APIs are present in the server/client along with appropriate responses and replies. The test cases show both the standard case and a case where message is intentionally delayed by using sleep() so that it receives messages in a non-causal order but can deliver it in a causally consistent manner.

Demo

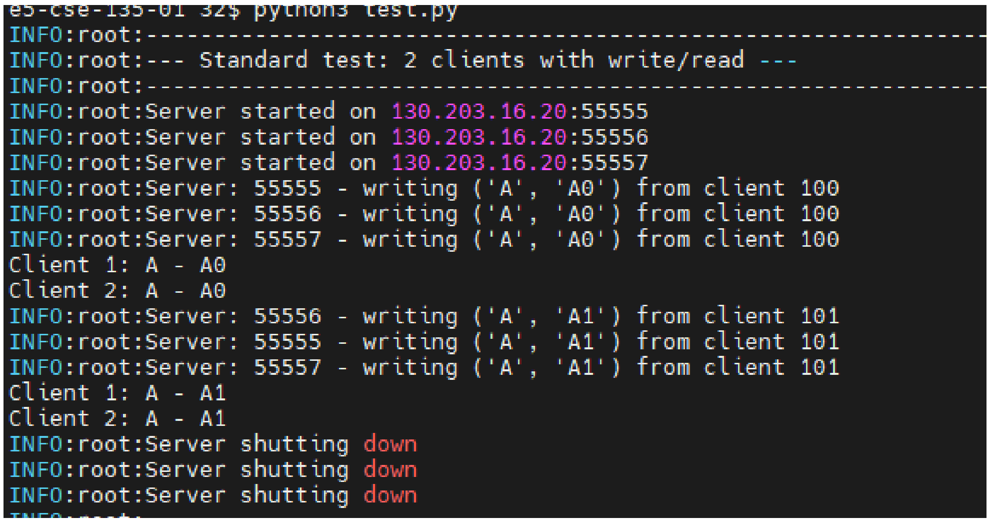
For the demo, we just change directory to test, and run the python file, test.py

```
cd ./test

python test.py
```

Outputs

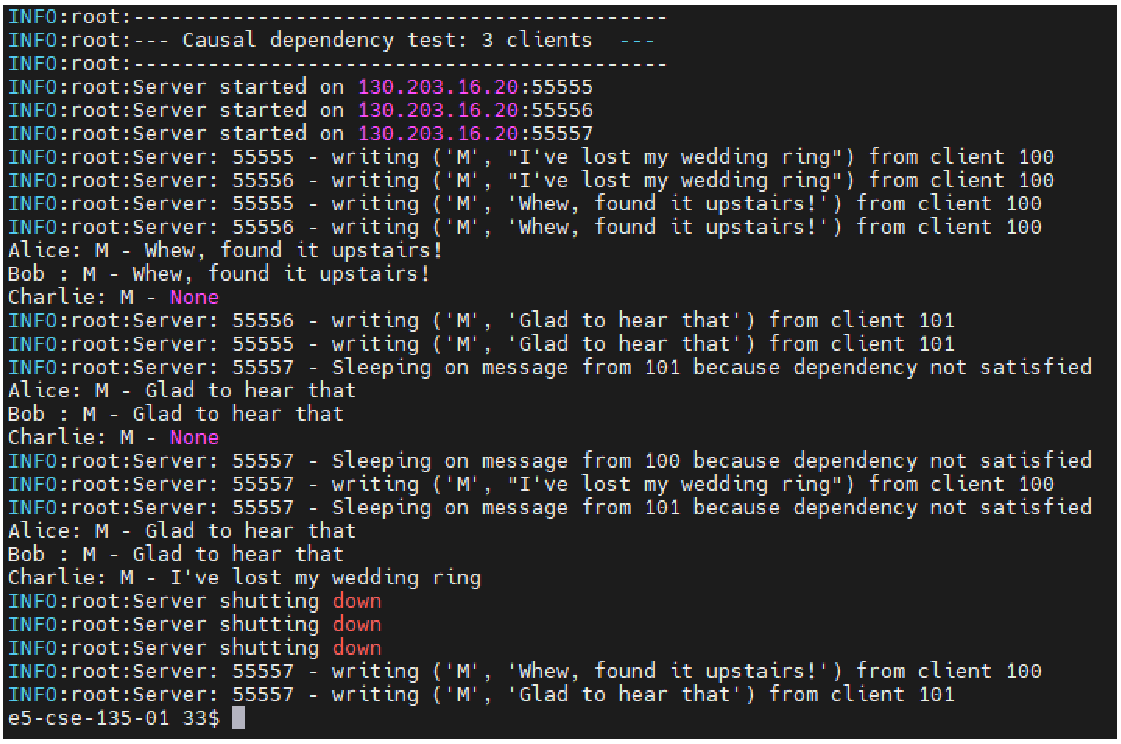
Below are the sample outputs for our demo test cases.



```
e5-cse-135-01 32$ python3 test.py
INFO:root:----------------------------------------------------------------
INFO:root:--- Standard test: 2 clients with write/read ---
INFO:root:----------------------------------------------------------------
INFO:root:Server started on 130.203.16.20:55555
INFO:root:Server started on 130.203.16.20:55556
INFO:root:Server started on 130.203.16.20:55557
INFO:root:Server: 55555 - writing ('A', 'A0') from client 100
INFO:root:Server: 55556 - writing ('A', 'A0') from client 100
INFO:root:Server: 55557 - writing ('A', 'A0') from client 100
Client 1: A - A0
Client 2: A - A0
INFO:root:Server: 55556 - writing ('A', 'A1') from client 101
INFO:root:Server: 55555 - writing ('A', 'A1') from client 101
INFO:root:Server: 55557 - writing ('A', 'A1') from client 101
Client 1: A - A1
Client 2: A - A1
INFO:root:Server shutting down
INFO:root:Server shutting down
INFO:root:Server shutting down
INFO:root:
```

a) Standard Test

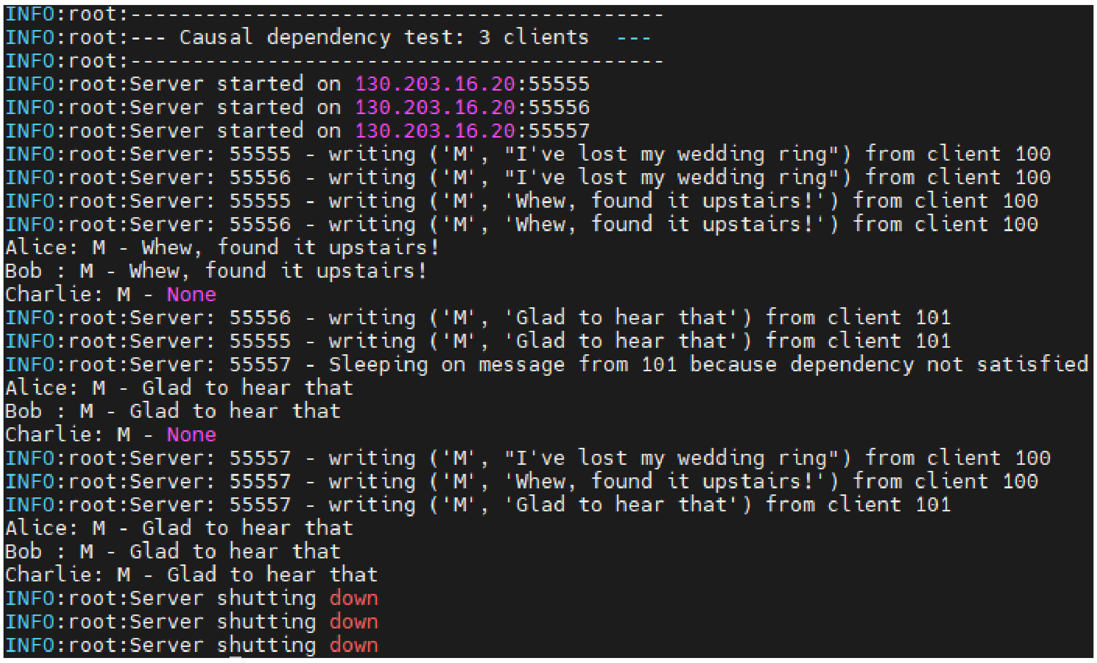
In the first example, it is a standard test where client 100 writes value ‘A0’ to key ‘A’. Both clients perform reads to get ‘A0’. Next, client 101 writes value ‘A1’. Again, both clients can read A1.



```
INFO:root:-----------------------------------------
INFO:root:--- Causal dependency test: 3 clients  ---
INFO:root:-----------------------------------------
INFO:root:Server started on 130.203.16.20:55555
INFO:root:Server started on 130.203.16.20:55556
INFO:root:Server started on 130.203.16.20:55557
INFO:root:Server: 55555 - writing ('M', "I've lost my wedding ring") from client 100
INFO:root:Server: 55556 - writing ('M', "I've lost my wedding ring") from client 100
INFO:root:Server: 55555 - writing ('M', 'Whew, found it upstairs!') from client 100
INFO:root:Server: 55556 - writing ('M', 'Whew, found it upstairs!') from client 100
Alice: M - Whew, found it upstairs!
Bob : M - Whew, found it upstairs!
Charlie: M - None
INFO:root:Server: 55556 - writing ('M', 'Glad to hear that') from client 101
INFO:root:Server: 55555 - writing ('M', 'Glad to hear that') from client 101
INFO:root:Server: 55557 - Sleeping on message from 101 because dependency not satisfied
Alice: M - Glad to hear that
Bob : M - Glad to hear that
Charlie: M - None
INFO:root:Server: 55557 - Sleeping on message from 100 because dependency not satisfied
INFO:root:Server: 55557 - writing ('M', "I've lost my wedding ring") from client 100
INFO:root:Server: 55557 - Sleeping on message from 101 because dependency not satisfied
Alice: M - Glad to hear that
Bob : M - Glad to hear that
Charlie: M - I've lost my wedding ring
INFO:root:Server shutting down
INFO:root:Server shutting down
INFO:root:Server shutting down
INFO:root:Server: 55557 - writing ('M', 'Whew, found it upstairs!') from client 100
INFO:root:Server: 55557 - writing ('M', 'Glad to hear that') from client 101
e5-cse-135-01 33$
```

b) Causal ordering test

In the second example, there are 3 clients Alice, Bob and Charlie having their own primary servers (55,56,57 respectively). Alice and Bob interact as per the wedding ring example, however, Charlie’s primary server recieves Alice’s second replicated message and Bob’s message before Alice’s first message. Alice first writes ‘ring lost’, then writes ‘found it’ which is successfully replicated in Alice and Bob's primary servers. However, Charlie hasn't received Alice's first message still and we see a 'None' on message read from his server and see the server logs buffering Alice's second message and Bob's message. Then after a while Charlie finally recieves Alice's first message as indicated by the server 57 log and Charlie's final output. Finally, the buffered messages are delivered causally as seen in the server logs to get the same state in all 3 at the end.



```
INFO:root:-------------------------------------------
INFO:root:--- Causal dependency test: 3 clients  ---
INFO:root:-------------------------------------------
INFO:root:Server started on 130.203.16.20:55555
INFO:root:Server started on 130.203.16.20:55556
INFO:root:Server started on 130.203.16.20:55557
INFO:root:Server: 55555 - writing ('M', "I've lost my wedding ring") from client 100
INFO:root:Server: 55556 - writing ('M', "I've lost my wedding ring") from client 100
INFO:root:Server: 55555 - writing ('M', 'Whew, found it upstairs!') from client 100
INFO:root:Server: 55556 - writing ('M', 'Whew, found it upstairs!') from client 100
Alice: M - Whew, found it upstairs!
Bob : M - Whew, found it upstairs!
Charlie: M - None
INFO:root:Server: 55556 - writing ('M', 'Glad to hear that') from client 101
INFO:root:Server: 55555 - writing ('M', 'Glad to hear that') from client 101
INFO:root:Server: 55557 - Sleeping on message from 101 because dependency not satisfied
Alice: M - Glad to hear that
Bob : M - Glad to hear that
Charlie: M - None
INFO:root:Server: 55557 - writing ('M', "I've lost my wedding ring") from client 100
INFO:root:Server: 55557 - writing ('M', 'Whew, found it upstairs!') from client 100
INFO:root:Server: 55557 - writing ('M', 'Glad to hear that') from client 101
Alice: M - Glad to hear that
Bob : M - Glad to hear that
Charlie: M - Glad to hear that
INFO:root:Server shutting down
INFO:root:Server shutting down
INFO:root:Server shutting down
```

c) Causal ordering test

This is another execution sequence where Charlie is able to display the output. The variation is due to using the sleep timer.