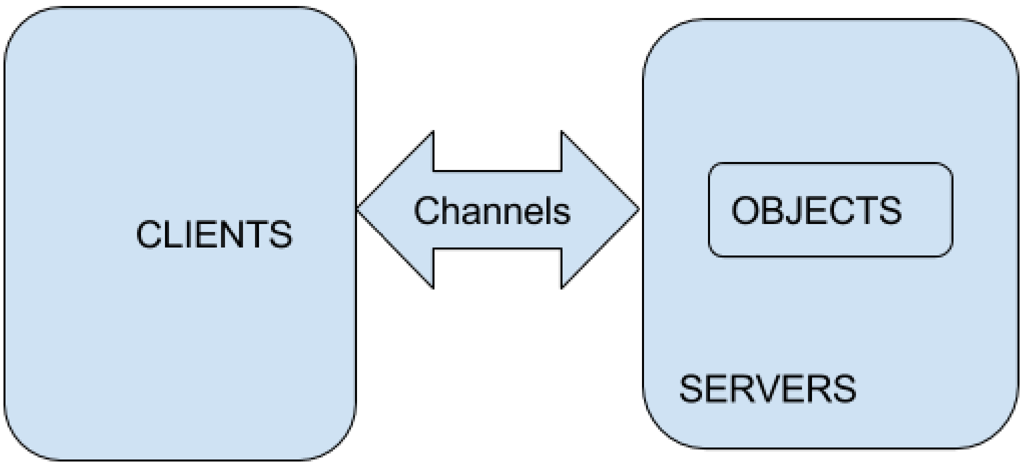
MODEL

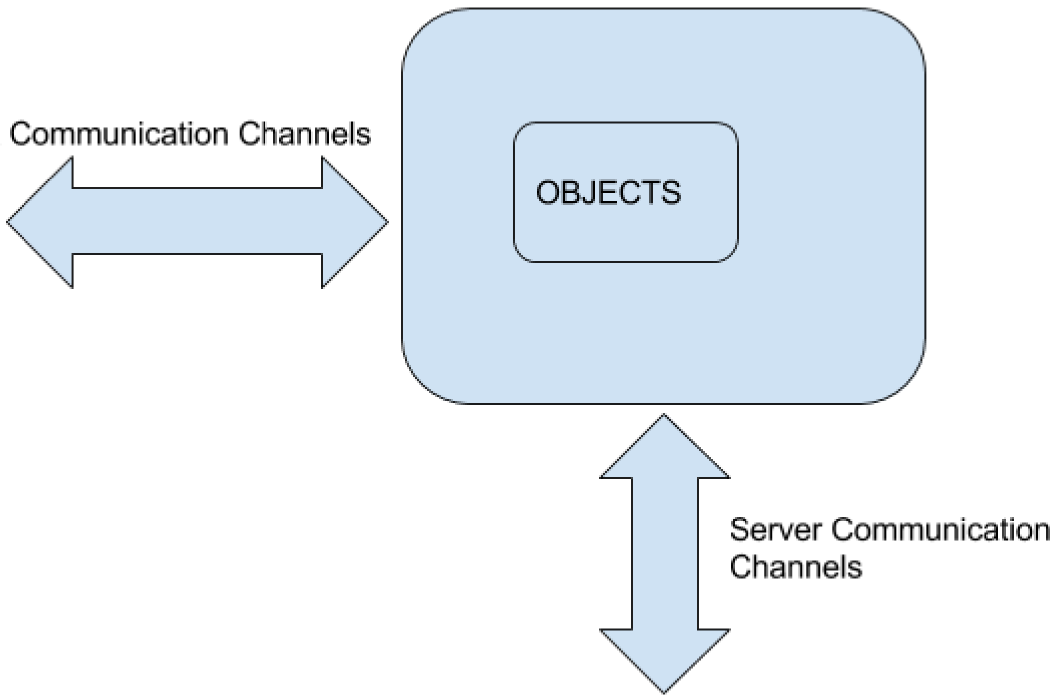
5 Clients interacting with 7 Servers over reliable FIFO channels.
Each client is connected to all 7 servers using TCP communication channels.
All servers are connected to each other via TCP communication channels.



CONSTRAINTS AND ASSUMPTIONS:

1. Servers are initialized before clients.
2. Clients connect to servers sequentially based on configuration provided.
3. All servers are connected to each other initially.
4. When channel goes down due to DISCONNECT, channel is not expected to be up again.

SERVERS:

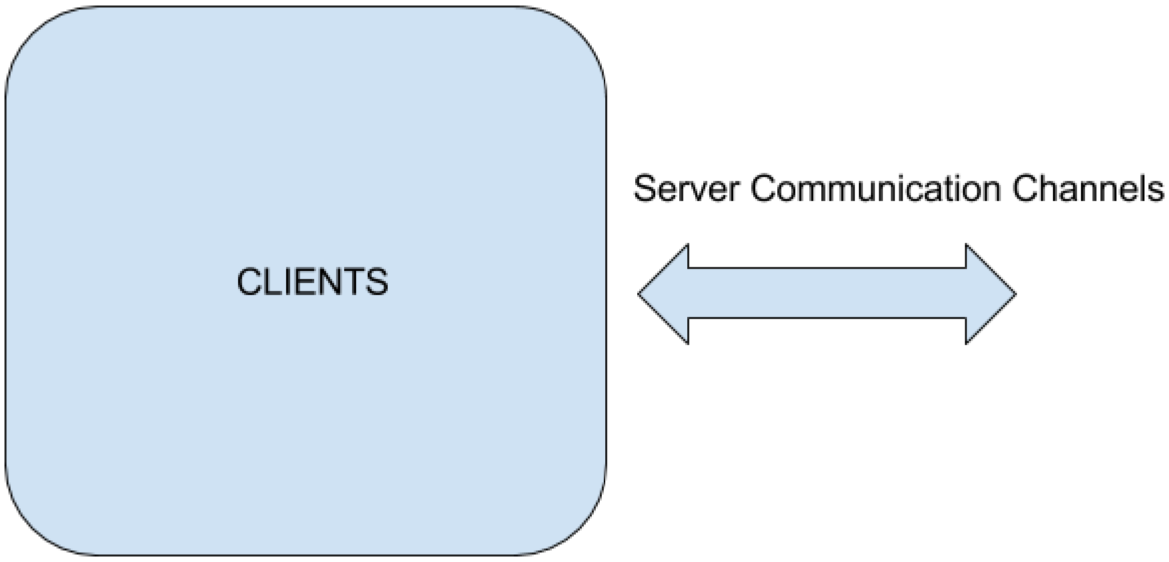


Operations on Objects:
1. WRITE
2. UPDATE
3. READ

Operations on Channels:
1. CONNECT
2. DISCONNECT
3. SUSPEND
4. RESUME

Each object has an ID which is 32 bit integer and used to identify the server via HASHING function.
Servers follow simple voting protocol during write operations. Servers vote and the value with maximum frequency in the votes is updated to object value.

CLIENTS:



Operations on Objects:
1. WRITE
2. UPDATE
3. READ

Operations on Channels:
1. CONNECT
2. DISCONNECT
3. SUSPEND
4. RESUME

Clients use Maekawa's Mutual Exclusion Algorithm to WRITE/UPDATE objects in servers. Servers are selected based on Robert Jenkin's Hash function with only 32 bit object IDS.

MESSAGE STRUCTURE:

COMMAND ID	OBJECT ID	OBJECT VALUE	TIMESTAMP	SEQ. NO

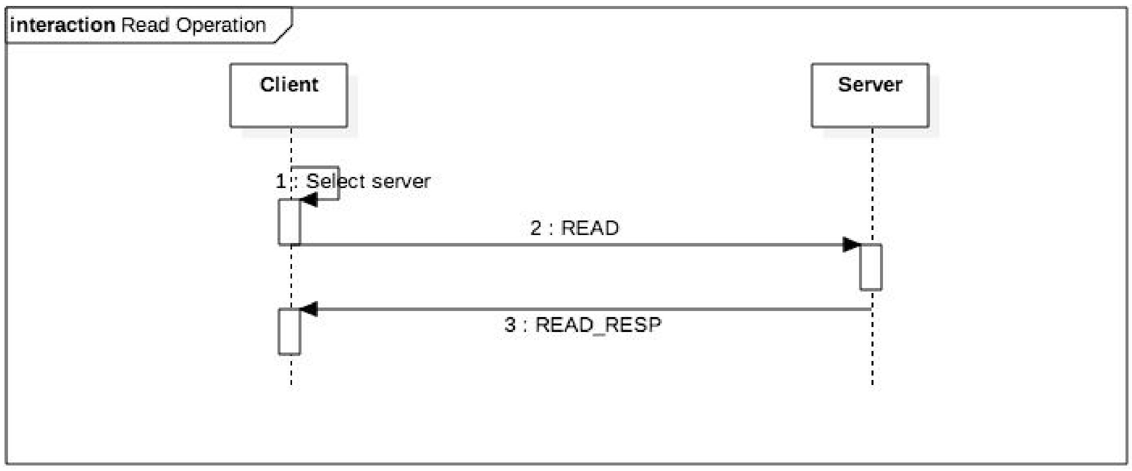
COMMANDS:

```
enum {
    MESSAGE_LOCK,
    MESSAGE_UNLOCK,
    MESSAGE_LOCK_RESP,
    MESSAGE_UNLOCK_RESP,
        MESSAGE_QUERY,
        MESSAGE_QUERY_RESP,
        MESSAGE_WRITE,
        MESSAGE_WRITE_RESP,
        MESSAGE_UPDATE,
        MESSAGE_UPDATE_RESP,
        MESSAGE_READ,
        MESSAGE_READ_RESP,
        MESSAGE_SUSPEND,
        MESSAGE_SUSPEND_RESP,
        MESSAGE_RESUME,
        MESSAGE_RESUME_RESP,
        MESSAGE_AGREE,
        MESSAGE_TYPE_MAX,
}MESSAGE_TYPES;
```

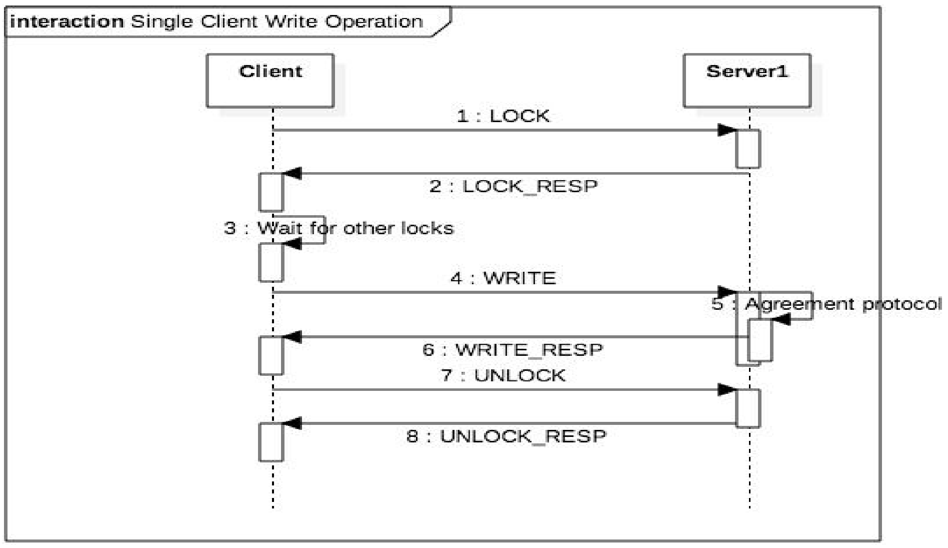
Operations:

1. READ operation: READ <OBJECT_ID>
2. WRITE operation: WRITE <OBJECT_ID> <OBJECT_VALUE>
3. UPDATE operation: UPDATE <OBJECT_ID> <OBJECT_VALUE>
4. Lock operation: LOCK <OBJECT_ID>
5. Unlock operation: UNLOCK <OBJECT_ID>

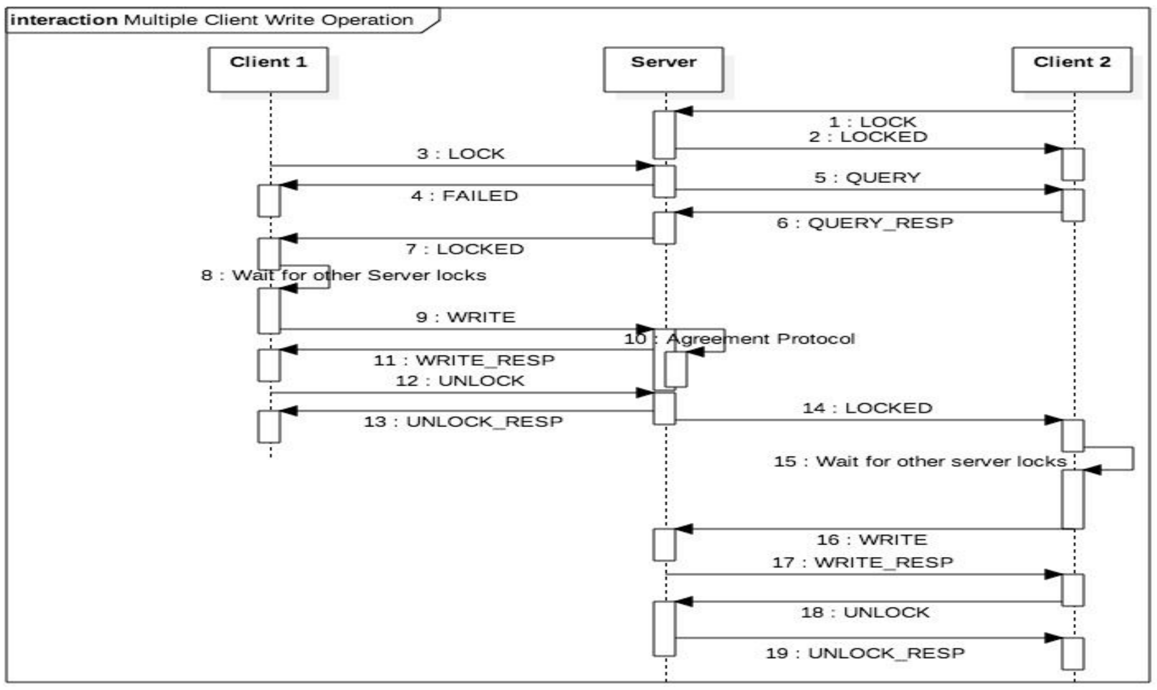
READ OPERATION:



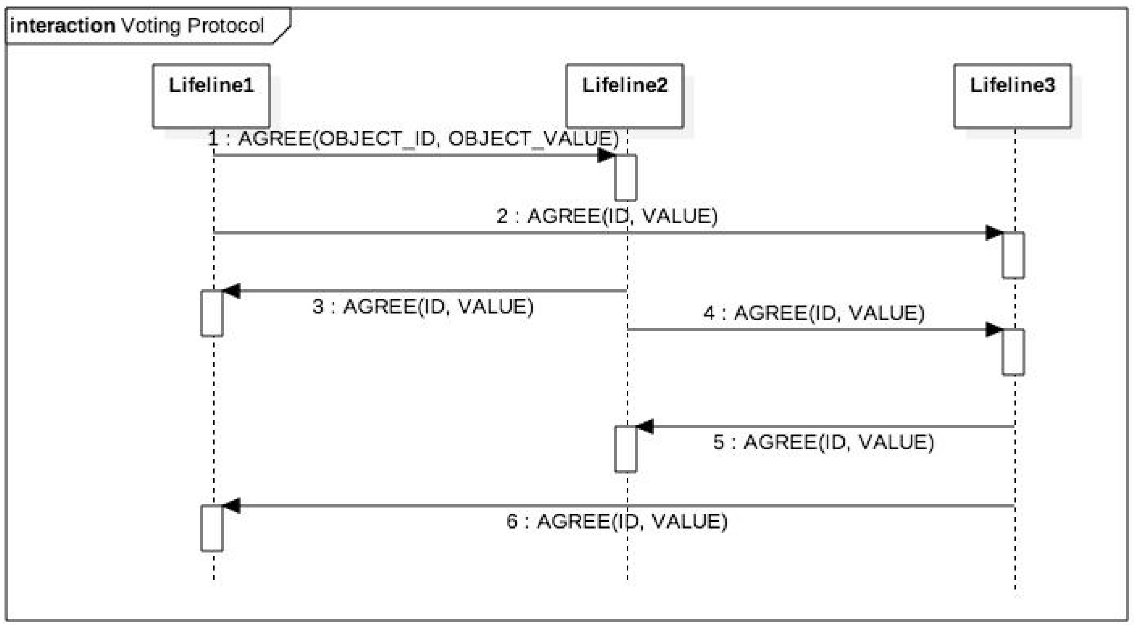
SINGLE CLIENT WRITE OPERATION:



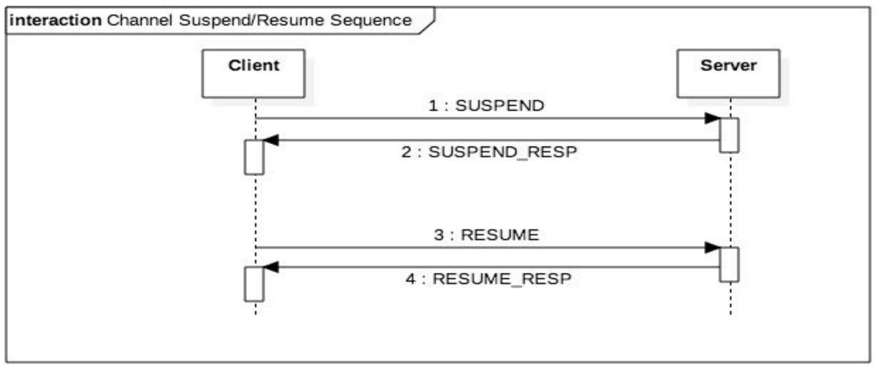
MULTIPLE CLIENT WRITE OPERATION:



VOTING PROTOCOL:



CHANNEL OPERATIONS:



HASH Function:

Knuth's Multiplication Method based Hash Function was initially tested. However, for every even ID'ed object, server 0 was chosen and for every odd ID'ed object, server 1 was chosen. This would not distribute the object storage across servers.

Hash function based on Robert Jenkin's 32 bit Mix Functions yielded better distribution for object storage. Hence this function was implemented and tested.

```
key = ~key + (key << 15); // key = (key << 15) - key - 1;
key = key ^ (key >>> 12);
key = key + (key << 2);
key = key ^ (key >>> 4);
key = key * 2057; // key = (key + (key << 3)) + (key << 11);
key = key ^ (key >>> 16);
```

PROTOCOL FOR MUTUAL EXCLUSION:

Requirement of write/update operation suggests 3 replicas in servers with IDs

1. H(object ID)%7
2. (H(object ID)+1)%7
3. (H(object ID)+2)%7

Update and writes have to be successful if atleast two servers are locked. Voting protocol is needed for this. To write/update client needs to lock atleast 2 servers. So, subset of each object ID is

<H(object ID), (H(object ID)+1)%7, (H(object ID)+2)%7>

If atleast two servers are not locked, client doesn't write or update the objects in servers.
Choice of mutual exclusion here was based on Maekawa's Mutual Exchange Algorithm.
Safety: Once a server is locked by a client, write operation by any other client is not permitted. Only the client which locked the server is given permission to write/update the server.
Liveness: To prevent deadlocks, QUERY and QUERY_RESP messages are exchanged among server and client. Client will respond as SUCCESS to QUERY if client is not able to lock all servers.

Client will respond FAILURE to QUERY if client is able to lock all servers. If server receives SUCCESS as QUERY_RESP, server will store the client request in request list and send LOCK message to pending client. If server receives FAILURE as QUERY_RESP, server waits for UNLOCK message from performing client and LOCK message is sent to pending client.

RESULTS:

1. Read from Server when no object with ID exists:

```
READ 0
Buffer = READ 0
Object id:0 server id: 1
TRACE(5): state(4) channelstate (3)
TRACE(25): recv command(11) fd(4)
NO SUCH OBJECT EXISTS ERROR FROM SERVER
```

2. Read from server when ID exists:

```
READ 0
Buffer = READ 0
Object id:0 server id: 1
TRACE(170): state(4) channelstate (3)
TRACE(190): recv command(11) fd(4)
OBJECT ID: 0 VALUE: 100
READ 0
Buffer = READ 0
Object id:0 server id: 0
TRACE(195): state(4) channelstate (3)
TRACE(215): recv command(11) fd(3)
OBJECT ID: 0 VALUE: 100
READ 0
Buffer = READ 0
Object id:0 server id: 1
TRACE(220): state(4) channelstate (3)
TRACE(240): recv command(11) fd(4)
OBJECT ID: 0 VALUE: 100
READ 0
Buffer = READ 0
Object id:0 server id: 2
TRACE(245): state(4) channelstate (3)
TRACE(265): recv command(11) fd(5)
OBJECT ID: 0 VALUE: 100
```

3. Write object to server:

```
WRITE 0 100
Buffer = WRITE 0 100
TRACE(30): WRITE 0 100 0
TRACE(30): obj(0) server1(0) server2(1) server3(2)
TRACE(30): state(1) channelstate (3)
TRACE(35): state(1) channelstate (3)
TRACE(40): state(1) channelstate (3)
TRACE(50): recv command(2) fd(3)
TRACE(50): Lock Response(1)
TRACE(50): lock_resp_recvd(0) locc_req_sent(3)
TRACE(55): recv command(2) fd(4)
TRACE(55): Lock Response(1)
TRACE(55): lock_resp_recvd(1) locc_req_sent(3)
TRACE(60): recv command(2) fd(5)
TRACE(60): Lock Response(1)
TRACE(60): lock_resp_recvd(2) locc_req_sent(3)
TRACE(60): Locked. WRITING
TRACE(60): state(6) channelstate (3)
TRACE(65): state(6) channelstate (3)
TRACE(70): state(6) channelstate (3)
TRACE(110): recv command(7) fd(5)
TRACE(115): recv command(7) fd(3)
TRACE(120): recv command(7) fd(4)
TRACE(120): UNLOCKing
TRACE(120): state(3) channelstate (3)
TRACE(125): state(3) channelstate (3)
TRACE(130): state(3) channelstate (3)
TRACE(145): recv command(3) fd(4)
TRACE(145): lock_resp_recvd(2) locc_req_sent(2)
TRACE(150): recv command(3) fd(3)
TRACE(150): lock_resp_recvd(1) locc_req_sent(1)
TRACE(155): recv command(3) fd(5)
TRACE(155): lock_resp_recvd(0) locc_req_sent(0)
```

```
        Buffer = READ 0
        Object id:0 server id: 1
        TRACE(170): state(4) channelstate (3)
        TRACE(190): recv command(11) fd(4)
        OBJECT ID: 0 VALUE: 100
```

4. Write to Server when 2 replicas are out of order: Channels 0 and 1 are suspended.

```
        SUSPEND 0
        Buffer = SUSPEND 0
        TRACE(275): state(8) channelstate (3)
        TRACE(295): recv command(13) fd(3)
        SUSPEND 1
        Buffer = SUSPEND 1
        TRACE(300): state(8) channelstate (3)
```

```
TRACE(320): recv command(13) fd(4)


Buffer = WRITE 0 200
TRACE(375): Value2 200 Value3 (null)
TRACE(375): WRITE 0 200 0
TRACE(375): obj(0) server1(0) server2(1) server3(2)
TRACE(375): state(1) channelstate (7)
Channel not connected. Adding to pending messages
TRACE(375): state(1) channelstate (7)
Channel not connected. Adding to pending messages
TRACE(375): state(1) channelstate (3)
Lock command send failed
TRACE(380): UNLOCKing
TRACE(380): state(3) channelstate (7)
Channel not connected. Adding to pending messages
TRACE(380): state(3) channelstate (7)
Channel not connected. Adding to pending messages
TRACE(380): state(3) channelstate (3)
Lock Send failed
TRACE(395): recv command(2) fd(5)
TRACE(395): Lock Response(1)
TRACE(395): lock_resp_recvd(0) locc_req_sent(1)
TRACE(405): recv command(3) fd(5)
TRACE(405): lock_resp_recvd(0) locc_req_sent(0)

Buffer = READ 0
Object id:0 server id: 2
TRACE(415): state(4) channelstate (3)
TRACE(435): recv command(11) fd(5)
OBJECT ID: 0 VALUE: 100
```

5. Write order: Maekawa algorithm is run to ensure clients write in the same order

```
Client 0:
WRITE 0 100
Buffer = WRITE 0 100
TRACE(5): Value2 100 Value3 (null)
TRACE(5): WRITE 0 100 0
TRACE(5): obj(0) server1(0) server2(1) server3(2)
TRACE(5): state(1) channelstate (3)
TRACE(10): state(1) channelstate (3)
TRACE(15): state(1) channelstate (3)
TRACE(35): recv command(2) fd(3)
TRACE(35): Lock Response(-1)
TRACE(35): lock_resp_recvd(0) locc_req_sent(3)
TRACE(40): recv command(2) fd(4)
TRACE(40): Lock Response(-1)
TRACE(40): lock_resp_recvd(1) locc_req_sent(3)
TRACE(45): recv command(2) fd(5)
TRACE(45): Lock Response(-1)
TRACE(45): lock_resp_recvd(2) locc_req_sent(3)
TRACE(70): recv command(2) fd(3)
TRACE(70): Lock Response(1)
TRACE(70): lock_resp_recvd(3) locc_req_sent(3)
TRACE(75): recv command(2) fd(4)
TRACE(75): Lock Response(1)
TRACE(75): lock_resp_recvd(3) locc_req_sent(3)
TRACE(75): Locked. WRITING
TRACE(75): state(6) channelstate (3)
TRACE(80): state(6) channelstate (3)
TRACE(90): recv command(2) fd(5)
TRACE(90): Lock Response(1)
TRACE(90): lock_resp_recvd(3) locc_req_sent(3)
TRACE(90): state(6) channelstate (3)
TRACE(125): recv command(7) fd(4)
TRACE(130): recv command(7) fd(5)
TRACE(135): recv command(7) fd(3)
TRACE(135): UNLOCKing
TRACE(135): state(3) channelstate (3)
TRACE(140): state(3) channelstate (3)
TRACE(145): state(3) channelstate (3)
TRACE(155): recv command(3) fd(3)
TRACE(155): lock_resp_recvd(2) locc_req_sent(2)
TRACE(160): recv command(3) fd(4)
TRACE(160): lock_resp_recvd(1) locc_req_sent(1)
TRACE(165): recv command(3) fd(5)
TRACE(165): lock_resp_recvd(0) locc_req_sent(0)
```

Client 1:

```
LOCK 0
Buffer = LOCK 0
TRACE(5): obj(0) server1(0) server2(1) server3(2)
TRACE(5): state(1) channelstate (3)
TRACE(10): state(1) channelstate (3)
TRACE(15): state(1) channelstate (3)
TRACE(30): recv command(2) fd(4)
TRACE(30): Lock Response(1)
TRACE(30): lock_resp_recvd(0) locc_req_sent(3)
TRACE(35): recv command(2) fd(3)
TRACE(35): Lock Response(1)
TRACE(35): lock_resp_recvd(1) locc_req_sent(3)
TRACE(40): recv command(2) fd(5)
TRACE(40): Lock Response(1)
TRACE(40): lock_resp_recvd(2) locc_req_sent(3)
UNLOCK 0
Buffer = UNLOCK 0
TRACE(45): UNLOCKing
TRACE(45): state(3) channelstate (3)
TRACE(50): state(3) channelstate (3)
TRACE(55): state(3) channelstate (3)
TRACE(65): recv command(3) fd(3)
TRACE(65): lock_resp_recvd(2) locc_req_sent(2)
TRACE(70): recv command(3) fd(4)
TRACE(70): lock_resp_recvd(1) locc_req_sent(1)
TRACE(75): recv command(3) fd(5)
TRACE(75): lock_resp_recvd(0) locc_req_sent(0)
```

REFERENCES:

1. https://gist.github.com/badboy/6267743
2. M. Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. ACM Transactions on Computer Systems, pages 145–159, May 1985.