CSE 511: Lab 3 Documentation

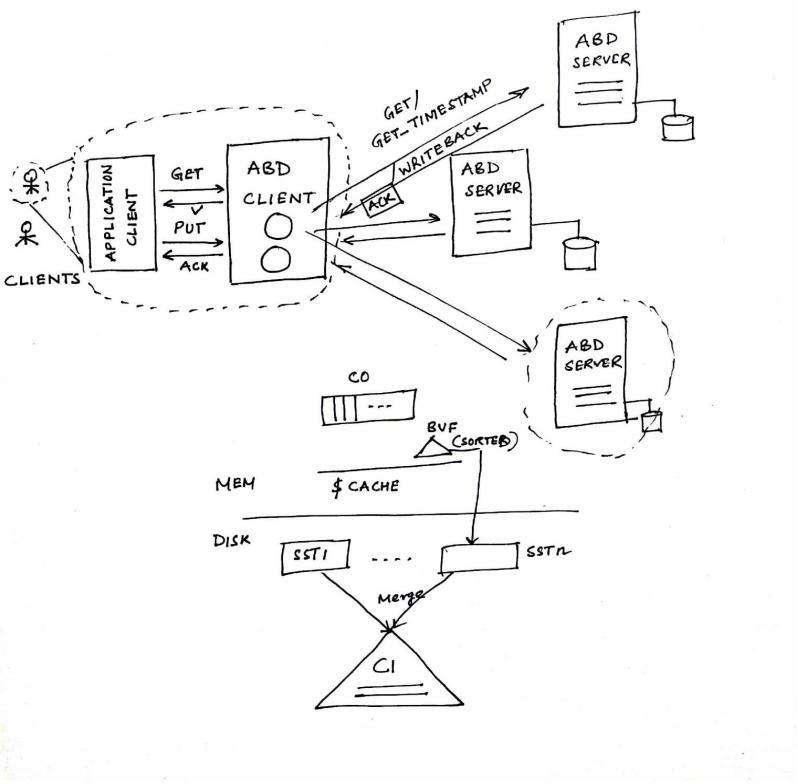
Team: Game of Threads

Members: Soumen Basu, Makesh Tarun Chandran, Vivek Bhasi

1. Problem Statement

In this assignment, we implemented a Distributed Shared Memory (DSM) in the form of a key-value store using the ABD algorithm. We also implemented a Distributed Blocking algorithm using distributed mutual locks as a baseline. Finally, we performed a comparative experimental performance evaluation of the ABD algorithm against the blocking version. We based our server base on the multi-threaded design. The key-value store was based on an LSM-Tree persistent data structure.

2. ABD Implementation and Design details



- *AbdServer* class implements the server interface for ABD. This is built on top of the multi-threaded design on LSM Tree based persistence. The C0 size is set to 200 for experiments but this can be configured. The LSM based implementation can be switched to naïve implementation if desired.
- *Utils.py* supports two new API for server-side, *get_timestamp and write.* The value field is modified to embed the tag(logical time) inside that. Note that get_timestamp is an optimization to reduce the network data transfer. Get already has timestamps embedded in value.
- Values are actually strings like the following:
  `"<actual_value> : <t.int> - <t.node_id>"`
- Like before the utils are called via the *factory* method *call_api*

- *AbdClient* class implements the interface that the vanilla application clients would use.
- *AbdClient* exposes *get* and *put* methods to the application client. The *get* internally makes use of *_get* method. This one sends the get request to all servers and waits for majority to respond (using *Barriers*) and then sends the data to writeback through server's write method. Note that server's methods are not directly exposed to the application client. Only ABD client knows them.
- *Logical_time* module takes care of the necessary logical time related operations such as get max timestamp, increment, and compare.

3. Blocking Implementation

- The Distributed Blocking algorithm hinges on the use of fine-grained locks for each key-value pair in the database of each server. The client in this model works in three phases: a request lock phase, read/write phase, and a release lock phase.
- This locking uses the idea of a majority based distributed mutex protocol.
- The server is similarly build on top of a multi-threaded design using LSM Tree as persistent.

3.1 Client side

- Request lock phase:
    - When a client wishes to read a value stored in the DSM, it first issues a read request (which asks permission before the actual read) to all the servers.
    - It then waits for the receipt of "lock grant" messages from majority of the servers. Grant message means that no other client has acquired the lock for that key-value pair and no one has acquired permission to write to it either.
    - In case the client doesn't get enough responses in a particular time, it retries for 3 times and then timeout occurs and the client aborts its request.
- Read/ Write phase:
    - After the client gets the lock, it sends out the actual read/write message to all the servers.
    - It then waits for the response from all the servers. Note that the way the algorithm is implemented, all the read values will be the same as during any read/write, no other client is allowed to access the same key-value pair stored on any server. However, to ensure that all the servers respond and are done with their part, we make the client wait for all their responses.
- Release lock phase:
    - Once the operation is done with, the client sends messages to all servers telling them to release the lock for the particular key-value pair.
- Fine grained locks are maintained as a dictionary of locks.

3.2 Server side:

- On receipt of a lock request the server checks it's copy of fine grained lock for that key and if the lock is free it sends lock grant message or else sends lock denied message.
- Read/ Write are same as base server.
- On receipt of a release lock message, the server releases its copy of the lock held on the key.

4. Issues faced

a. We used timeouts and retries in the lock. We didn't use the queue based algorithm discussed in the class.

b. The configuration and deployment is manual and takes time. This could be automated using AWS CLI (Boto3).

c. The server id vs host-port mappings are needed to be manually entered in the server_conts file in client side. We would probably like to automate that like a real system does.

5. Requirements

a. Python 3

b. PickleDB for naive implementation (if used)

6. Populate DB

Can be populated using the script *populate_db.py.*

```
python populate_db.py -n <num_entries> -s <key-value-size> -d <db_name> -t
<type> -i <node_id_range>
```

`<type>: should be "pickle" for Naïve persistence and should be "file" for LSM.`

Keys are integers starting from 1. We kept keys as integers for the ease of sorting.

7. Running Servers

The blocking server:

```
python blocking_server.py <0.0.0.0> <port> <cache_size> <db_name>
```

For the LSM Tree based key-value server (db_name is just C1 file name):

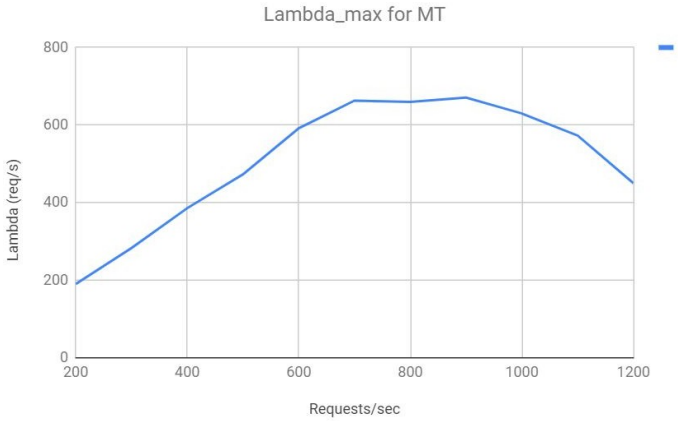
```
python abd_server.py <0.0.0.0> <port> <cache_size> <db_name>
```

8. Deployment

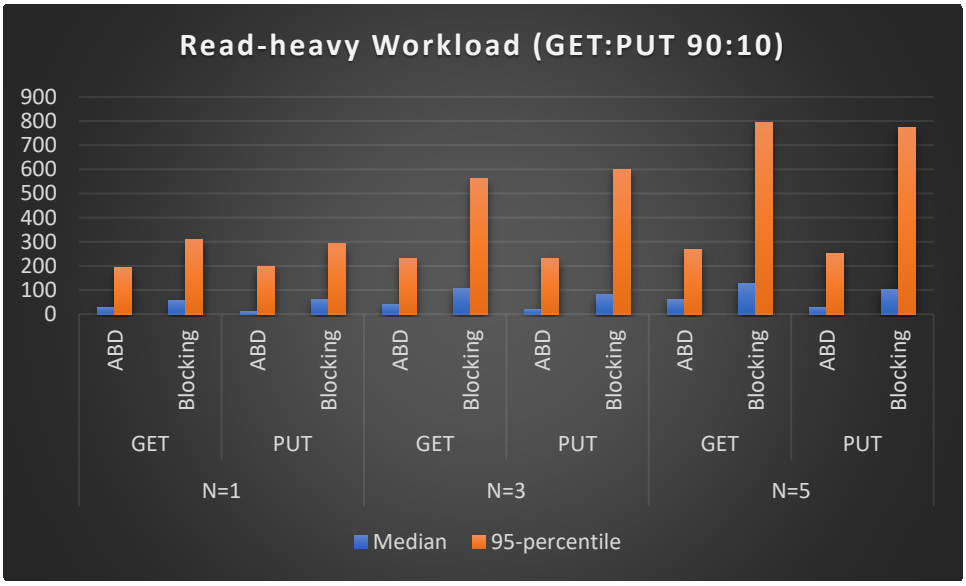
- Generate the DB in each of the server machines (We are maintaining replication factor of N=number of servers)
- Run the servers in each machine using the previous commands.
- Go to the client machines and change the server host and port ids in the *server_consts.py (SERVER_ID_MAP)*
- The application clients can run by creating the ABD client objects and calling their get and put apis.

9. Experimental Setup and Results

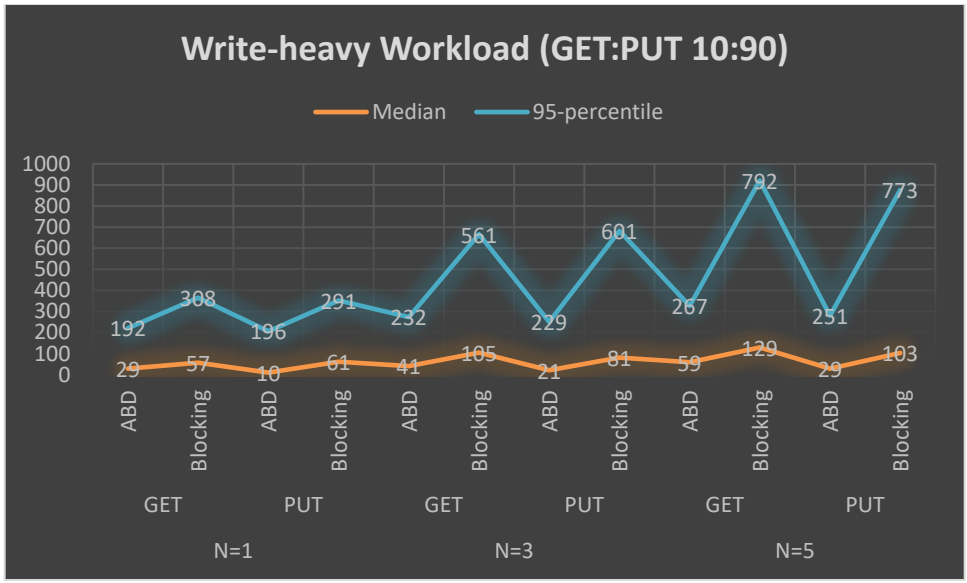
a. Server: We ran on t2.small AWS EC2 instance (Ubuntu 18.04 w/ 20GB gp2)
b. Client Node: t2.xlarge (Ubuntu 18.04 w/ 20GB gp2)
c. Servers and clients were created within a data center
d. Request Distribution is deterministic in nature (after 100 sec increase the number of requests by 100).
e. Skewed key popularity distribution: 10% keys make 90% requests
f. *Lambda_max Measurement*: We used similar measurements like Lab-1 for this purpose. It was measured as 663 requests/sec. We approximated to lambda_max = 650 req/s. Following graph shows the lambda_max graph.



GET/ PUT Latencies for different settings:



This bar plot shows the GET and PUT latencies (Median and 95-percentile) for Read Heavy workloads (90:10, GET:PUT). ABD and blocking are shown. Clearly ABD performs better than blocking as the number of servers increase.



This plot shows the GET and PUT latencies (Median and 95-percentile) for Write Heavy workloads (10:90, GET:PUT). ABD and blocking are shown. Clearly ABD performs better than blocking as the number of servers increase.

It is worthy of noting that ABD performs almost similarly for both type of workload. This is because both GET and PUT involve two round trips. Although our optimization for lowering data transfer by transferring only the timestamp for *write_query* improves the PUT latencies slightly than GET.