Running blockbench

Wejdene HAOUARI

15 Mars 2021

Abstract

This document provides a walk through running the BLOCKBENCH benchmark for hyperledger fabric 2.2 blockchain. It also provide an architecture of the BLOCKBENCH project and the steps to follow in order to add hyperledger Sawtooth.

Infrastructure

We have used the folowing hardware configuration: Virtuel machine with 4GB of RAM and 2CPU, we have installed ubuntu 16.08

1 BLOCKBENCH setup

The first step is to clone the BLOCKBENCH repository:

```
git clone https://github.com/ooibc88/blockbench
```

1.1 Hyperledger Fabric Network setup

We are going to launch a simple network with a single orderer and two peers from different organizations.

1. Pull down the platform-specific binaries and images of fabric v-realse-2.2 and places them in the bin sub-directory of the current working directory.

```
curl -sSL https://bit.ly/2ysbOFE | bash -s -- 2.2.2 1.4.9
```

2. We need to add the add the bin directoy to the PATH environment variable in order to use them in the BLOCKBENCH project without without fully qualifying the path to each binary.

```
export PATH=<path to download location>/bin:$PATH
```

2. Set global variable

```
CHANNEL_NAME=mychannel
```

3. Now we have to go to the fabric 2.2 repository in BLOCKBENCH repository and launch the network:

```
cd blockbench/benchmark/fabric-v2.2/
./network.sh up createChannel -ca -i 2.2 -c ${CHANNEL_NAME}
```

4. The next step is to deploy the chaincode, we are going to use KVSTORE as an example, it is the same setup of the other workload we have only to change CC_NAME variable.

```
CC_NAME=kvstore
CC_SRC_PATH="../contracts/fabric-v2.2/${CC_NAME}"
./network.sh deployCC -ccn ${CC_NAME} -ccp ${CC_SRC_PATH}
```

5. Start up the helper processes:

Fabric 2.2 does not provide C++ SDK, for this reason the authors wrap up the Fabric's service into a json web service, implemented by NodeJS. In macro benchmarks, client drivers will contact these helpers processes to interact with Fabric network.

There is two type of processes helper, the first one is block server that with provides the ledger information, it will be contacted by the Status_thread. The second one is the transactions servers that will json requests and routes them to Fabric network, it will be contacted my multiple Client_Thread.

```
## Install Dependencies and Prepare Identities
 cd services/;
 npm install;
 node enrollAdmin.js
 node registerUser.js
 # Launch processes helpers
 node block-server.js ${CHANNEL_NAME} 8800 > block-server.log 2>&1 &
 MODE=open_loop
 node txn-server.js ${CHANNEL_NAME} ${CC_NAME} ${MODE} 8801 >
 txn-server-8801.log 2>&1 &
 node txn-server.js ${CHANNEL_NAME} ${CC_NAME} ${MODE} 8802 >
 txn-server-8802.log 2>&1 &

```

Note: $MODE is determined by individual benchmarks. In most cases, macro benchmarks opt for $MODE=open_loop and micro benchmarks opt for $MODE=closed_loop

1.2 Workload setup

1. First we need to install the dependencies:

```
sudo apt-get install build-essential
sudo apt-get install libtool
sudo apt-get install autoconf
sudo apt-get install libcurl4-gnutls-dev
```

The authors have also used restclient-cpp dependencies

```
git clone https://github.com/mrtazz/restclient-cpp.git
cd restclient-cpp/ && ./autogen.sh && ./configure && sudo make install
cd ..
```

2. install BLOCKBENCH kvstore executable:

When reading the README file, we need a script called driver in order to run the workload. The script is not existing by default, However we can notice a Makefile in the kvstore repository, so the first that we need to do is to install BLOCKBENCH kvstore executable:

```
cd blockbench/src/macro/kvstore/
make
```

2. The next step is to prepare the endpoint

```
##endpoint=[block-service-address],[txn-service-address1],[txn-service-address2]..
endpoint=localhost:8800,localhost:8801,localhost:8802
```

3. Finally we can launch the client driver processes:

```
./driver -db fabric-v2.2 -threads 1 -P workloads/workloada.spec -txrate 5
-endpoint ${endopoint} -wl ycsb -wt 20
```

2 Workload usage

2.1 KVSTORE

-threads n: execute using n threads (default: 1)

-wt deploytime: waiting time in second before start to submit transactions for deployment the smart contract/chaincode

-db dbname: specify the name of the DB to use (e.g., hyperledger)

-wl workload: specify the type of smart contract to run (choices: ycsb, donothing, smallbank. By default: donothing)

-P propertyfile: load properties from the given file. Multiple files can be specified, and will be processed in the order specified

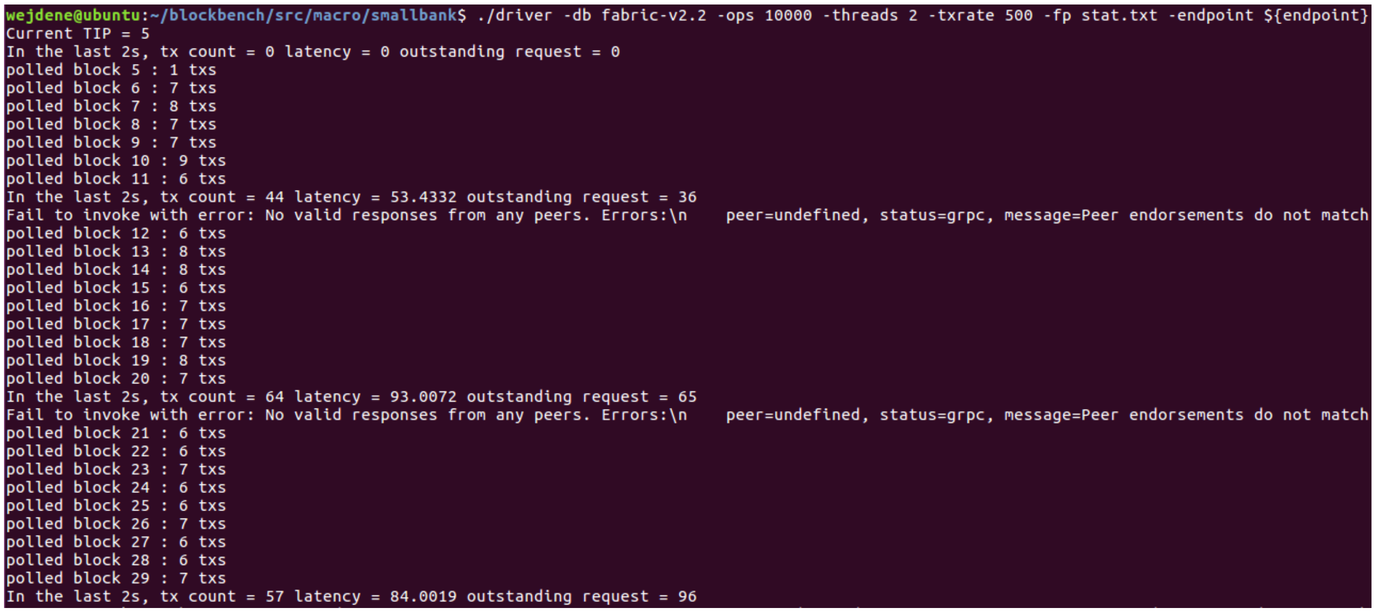
-endpoint: The endpoints of blockchain server

txrate: transaction rate

2.1.1 Output

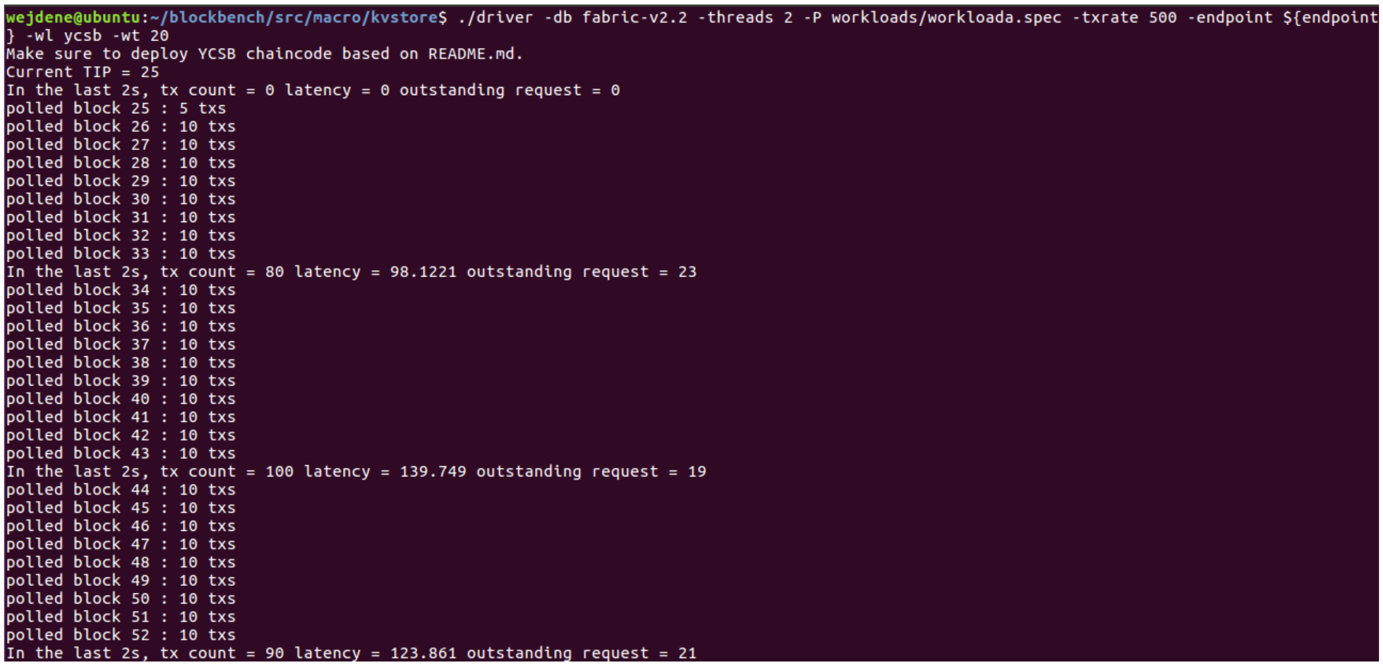
Output explanation:

1. The driver periodically polls for new blocks and it report the number of blocks that were included in that block



```
wejdene@ubuntu:~/blockbench/src/macro/smallbank$ ./driver -db fabric-v2.2 -ops 10000 -threads 2 -txrate 500 -fp stat.txt -endpoint ${endpoint}
Current TIP = 5
In the last 2s, tx count = 0 latency = 0 outstanding request = 0
polled block 5 : 1 txs
polled block 6 : 7 txs
polled block 7 : 8 txs
polled block 8 : 7 txs
polled block 9 : 7 txs
polled block 10 : 9 txs
polled block 11 : 6 txs
In the last 2s, tx count = 44 latency = 53.4332 outstanding request = 36
Fail to invoke with error: No valid responses from any peers. Errors:\n    peer=undefined, status=grpc, message=Peer endorsements do not match
polled block 12 : 6 txs
polled block 13 : 8 txs
polled block 14 : 8 txs
polled block 15 : 6 txs
polled block 16 : 7 txs
polled block 17 : 7 txs
polled block 18 : 7 txs
polled block 19 : 8 txs
polled block 20 : 7 txs
In the last 2s, tx count = 64 latency = 93.0072 outstanding request = 65
Fail to invoke with error: No valid responses from any peers. Errors:\n    peer=undefined, status=grpc, message=Peer endorsements do not match
polled block 21 : 6 txs
polled block 22 : 6 txs
polled block 23 : 7 txs
polled block 24 : 6 txs
polled block 25 : 6 txs
polled block 26 : 7 txs
polled block 27 : 6 txs
polled block 28 : 6 txs
polled block 29 : 7 txs
In the last 2s, tx count = 57 latency = 84.0019 outstanding request = 96
```

Figure 1: smallbank output for Fabriv 2.2



```
wejdene@ubuntu:~/blockbench/src/macro/kvstore$ ./driver -db fabric-v2.2 -threads 2 -P workloads/workloada.spec -txrate 500 -endpoint ${endpoint
} -wl ycsb -wt 20
Make sure to deploy YCSB chaincode based on README.md.
Current TIP = 25
In the last 2s, tx count = 0 latency = 0 outstanding request = 0
polled block 25 : 5 txs
polled block 26 : 10 txs
polled block 27 : 10 txs
polled block 28 : 10 txs
polled block 29 : 10 txs
polled block 30 : 10 txs
polled block 31 : 10 txs
polled block 32 : 10 txs
polled block 33 : 10 txs
In the last 2s, tx count = 80 latency = 98.1221 outstanding request = 23
polled block 34 : 10 txs
polled block 35 : 10 txs
polled block 36 : 10 txs
polled block 37 : 10 txs
polled block 38 : 10 txs
polled block 39 : 10 txs
polled block 40 : 10 txs
polled block 41 : 10 txs
polled block 42 : 10 txs
polled block 43 : 10 txs
In the last 2s, tx count = 100 latency = 139.749 outstanding request = 19
polled block 44 : 10 txs
polled block 45 : 10 txs
polled block 46 : 10 txs
polled block 47 : 10 txs
polled block 48 : 10 txs
polled block 49 : 10 txs
polled block 50 : 10 txs
polled block 51 : 10 txs
polled block 52 : 10 txs
In the last 2s, tx count = 90 latency = 123.861 outstanding request = 21
```

Figure 2: KVSTORE output for Fabriv 2.2

2. Based on the txs in the mined blocks, the driver determines the number of txs that were processed in the last 2 seconds

3. For these txs the driver reports the accumulated latency (ie., the sum of the latencies of the single txs) in seconds. To determine the latency of a tx the driver keeps track of when a tx was

submitted.

4. The throughput for one client could be defined as txCount/pollInterval, for several clients one could sum all txCount by all clients and divide by the overall poll interval, since all clients are started and stopped at the same time.

3 Adding hyperledger Sawtooth

3.1 BACKEND: Hyperledger Sawtooth

1. The first step is to set hyperledger Sawtooth network we have to choose the number of validators and consensus.

2. The second step to is add the appropriate transaction family to be queried by the workload, by default smallbank transaction family is implemented, and we can use intkey as a key value storage for KVSTORE workload.

3. The final step is to implement REST API for queering and execute transactions on the blockchain.

3.2 Workload : KVSTORE

3.2.1 Workload configuration

the README file of the github project didn't explain the configuration file inputs and I had to look at the source code to understand the workload spec file parameters. Bellow are the parameters for KVSTORE Workload (these parameters are part of core_workload interface):

TABLENAME_PROPERTY: The name of the database table to run queries against(usertable)

FIELD_COUNT_PROPERTY: number of fields in a record (10)

FIELD_LENGTH_DISTRIBUTION_PROPERTY: field length distribution [”uniform”, ”zipfian” (favoring short records), and ”constant”] (constant)

FIELD_LENGTH_PROPERTY: the length of a field in bytes(100)

READ_ALL_FIELDS_PROPERTY: deciding whether to read one field (false) or all fields (true) of a record.(true)

WRITE_ALL_FIELDS_PROPERTY: write one field (false) or all fields (true) of a record.

READ_PROPORTION_PROPERTY: proportion of read transactions (0.95)

UPDATE_PROPORTION_PROPERTY: the proportion of update transactions(0.05)

INSERT_PROPORTION_PROPERTY: the proportion of insert transactions (0.0)

SCAN_PROPORTION_PROPERTY: the proportion of scan transactions (0.0)

READMODIFYWRITE_PROPORTION_PROPERTY: read-modify-write transactions (0.0)

REQUEST_DISTRIBUTION_PROPERTY: the distribution of request keys [”uniform”, ”zipfian” and ”latest”] (uniform)

MAX_SCAN_LENGTH_PROPERTY: max scan length (number of records) (1000)

SCAN_LENGTH_DISTRIBUTION_PROPERTY:he name of the property for the scan length distribution. Options are ”uniform” and ”zipfian” (favoring short scans)(uniform)

INSERT_ORDER_PROPERTY: the order to insert records. Options are ”ordered” or ”hashed”.(hashed)

INSERT_START_PROPERTY (0)

RECORD_COUNT_PROPERTY: Number of records (no default value)

OPERATION_COUNT_PROPERTY: Number of operations (no default value)

3.2.2 Step for adding Hyperledger Sawtooth

In this section we are going to analyze how KVSTORE work in order to identify the part that we need to add in order to integrate hyperledger sawtooth.

1. As figure 4 shows, in order to run KVSTORE workload on hyperledger Sawtooth, the first step is to extends extends DB.h interface that define the methods needed in order to interact with the blockchain. figure 3 show class diagram of DB class. We need to implement the corresponding API in the rest API component of hyperledger SAWTOOTH.

2. The next step is to add SAWTOOTH to the DB factory.

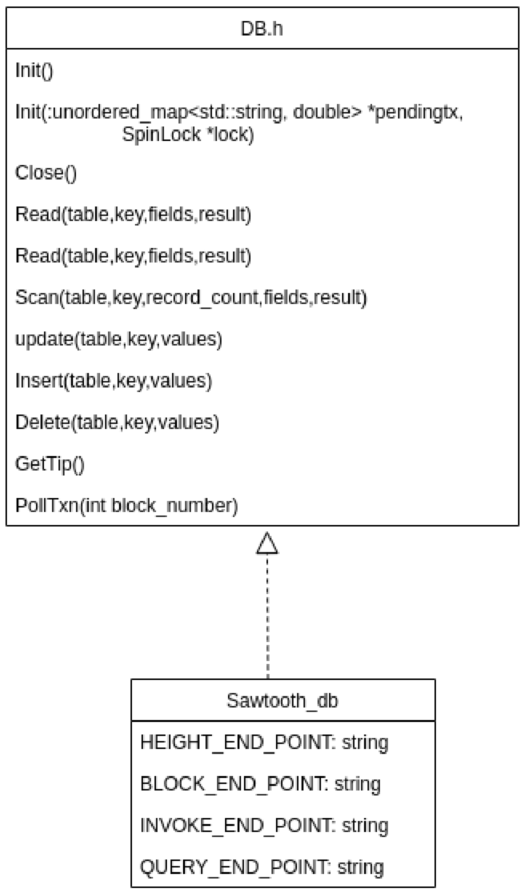


Figure 3: Class diagram: DB

3. The final step is to configure the workload.

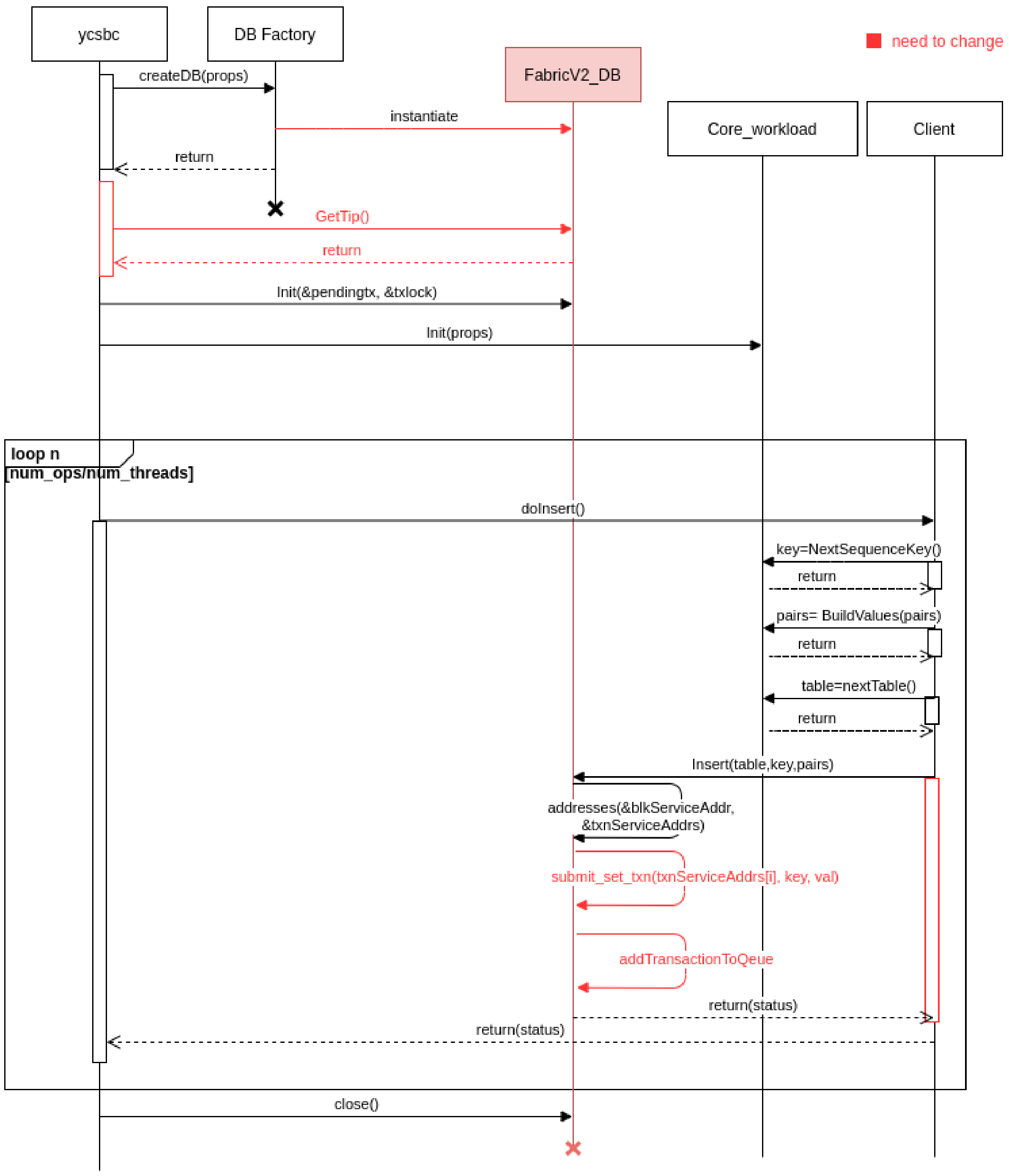
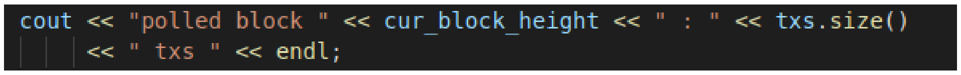


Figure 4: Sequence diagram: kvstore workload Fabric 2.2

4 Observation

1. Hyperledger Fabric 1.2 and 2.2 doesn't have support saturation experiments (varying client request rate) also security experiments (attacks simulated via network partition).

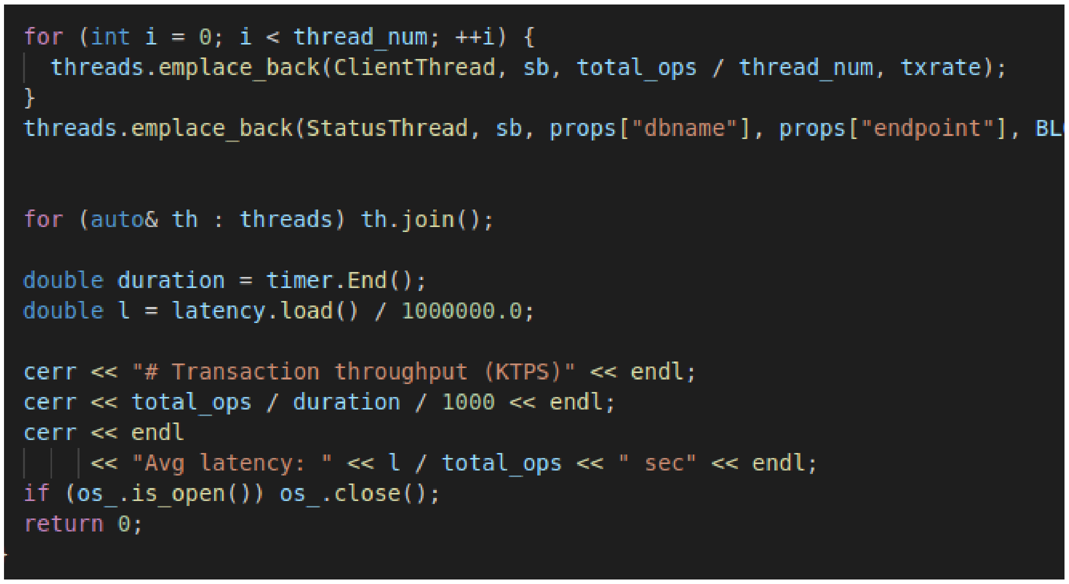
2. After analyzing KVSOTRE and Smallbank workloads code, we have noticed that the result are written directly to the console and not into files as figure 3 shows. Also the output not in user friendly format to create graphs.



```
cout << "polled block " << cur_block_height << " : " << txs.size()
      << " txs " << endl;
```

Figure 5: Writing stats to console

3. For smallbank (also KVSTORE) workload, the transactions throughput and the average latency are calculated at the end of the program as figure 4 shows, however, the stats thread continue to run even thought the client finished sending requests , for this reason we cannot see the throughput and the average latency because the program never end.



```
for (int i = 0; i < thread_num; ++i) {
  threads.emplace_back(ClientThread, sb, total_ops / thread_num, txrate);
}
threads.emplace_back(StatusThread, sb, props["dbname"], props["endpoint"], BL


for (auto& th : threads) th.join();

double duration = timer.End();
double l = latency.load() / 1000000.0;

cerr << "# Transaction throughput (KTPS)" << endl;
cerr << total_ops / duration / 1000 << endl;
cerr << endl
     << "Avg latency: " << l / total_ops << " sec" << endl;
if (os_.is_open()) os_.close();
return 0;
```

Figure 6: Throughput calculation