CS 6650 Assignment 3 (Optional - 5 pt)

Read the entire document from the beginning to the end before you start to program. Understand what you need to do for each step of the assignment and plan you time accordingly.

Overview

Through this assignment, you will learn how to build a distributed transaction system based on two-phase commit. You will learn:

- how to implement a standalone server that processes a transaction.
- how to implement sharded servers.
- how to implement two-phase commit using the sharded servers and the transaction manager.

Distributed transactional key-value store for an online bidding system

You will be building a distributed key-value storage system that supports distributed transactions to implement an online bidding system for a robot auction. Customers may read the current bid of a robot and place a higher bid. However, if there was an update to the bid by another user to the same robot in-between reading bid information and placing a new bid, the user's bidding will fail because the bid is based on an outdated data. The user places bids to multiple robots in a bundle and if one of the bidding fails, then all others in the same bundle will be retracted.

The list of robots and the bidding information in the auction is too large to be stored in a single resource manager (RM) server so you will have to partition the data into multiple RMs. The data is maintained in a key-value format in an array rather than a map. The key is the robot id and the value is the bid, bidder, and the version information of the key-value pair.

The read requests and a bundle of bid placements are executed as a transaction. The transaction may commit or abort depending on the transaction decisions by the RMs and the transaction manager (TM). You should start from the given code (the solution code for assignment #1 + extra): the server code will be used to implement the TM and the RM and the client code will be used for the transaction-issuing client.

For Section 1, you should create "client" and "server" programs. For Section 2, you may copy the code for Section 1 and create "2pc-client", "tm", and "rm".

You may assume no node and network failures in this assignment.

1 A standalone resource manager (RM) and a client

You will first implement a standalone RM that directly interacts with the client and makes its own transaction decisions. In a later section, the standalone RM will be transformed into a TM and a distributed RM that implement two-phase commit.

1.1 Data

You will implement a key-value table using an array as the key (robot id) is an integer type. The value corresponding to the key is a bid-customer_id-version triplet. Each RM needs to know which key range it is covering. The RM keeps a kv_base value as the start of the key range: if kv_base = 2000, then the array index 0 corresponds to key 2000 and index 10 corresponds to key 2010. The RM also maintains the table size, kv_tbl_size, which indicates the number of key-value pairs the table holds. The last key that the RM holds is kv_base + kv_tbl_size - 1. Instead of filling in the key-value table on the go, we will assume that the key-value tables are pre-filled when the RM starts: you should dynamically allocate the array as in the comment below.

```
struct kv_value {
        int bid;          // current bid
        int customer_id; // bidding customer's id
        int version;      // version number determined by the system
        kv_value () { bid = 0; customer_id = -1; version = 0; }
};
int kv_tbl_size;
int kv_base;

// may initialize with kv_table = new kv_value[kv_tbl_size];
struct kv_value *kv_table; // indexed by robot_id
```

1.2 Transaction (TX) format and semantics

A transaction (TX) is a set of read and write operations that are bundled together. A TX may commit as long as it does not violate the commit conditions, and otherwise it should abort. You will implement a semantic that checks read-write conflicts. For example, if the data that a user has read in a TX is overwritten by another user at the time of evaluating the TX, then the TX should abort. If none of the data that are read in a TX have been overwritten, then the TX can commit. Whether the data is overwritten or not can be noticed by comparing the version number of the read for a key-value pair in the TX and the version number of the kv_value in the kv_table.

We assume that the customer first reads a set of robot bidding information and then places the bid for all robots that the customer has read about. The customer will only increment the bid by 1 from the information that was read. For simplicity we will assume that the customers always reads 3 robot information and thus places 3 bids per a TX request.

- Read: a read request is immediately sent to the RM and RM returns the data to the user. The user adds the key and the version number of the read in the TX instance and uses the returned bid value for the bidding by incrementing it by 1.
- Write: a write request is not immediately sent to the RM and is added to the TX instance. The writes are sent as part of a TX request to the RM. When the TX commits, the write included in the TX instance will be applied to the RM at once. If the TX has to abort, the writes in the TX will simply be discarded. Data updated (written) by the same TX should have the same version number and the version number is determined at the TX commit (or prepare) time by the system (by the standalone RM in this section, and by the TM in later sections).

Therefore, the TX should keep track of all read information and write operations in the following format:

```
struct tx_read {
        int robot_id;
        int version;  // version returned from a read
};
struct tx_write {
        int robot_id;
        int bid;
        int customer_id;
};
struct tx {
        int version; // version that will aplly to writes
        struct tx_reads[3];
        struct tx_writes[3];
};
```

1.3 An example of transaction (TX) execution

For example, let's assume customer 10 is trying to bid for robot 1, 2, and 3. The customer and the RM interact as follows:

1. customer begins TX (create a TX instance);
2. customer sends a read request for robot_id 1 to RM;
3. RM returns kv_value for robot 1 to the customer;
4. customer receives read response (bid: 10, customer_id: 20, version: 12);
5. customer adds read info (robot_id: 1, ver: 12) into the TX;
6. customer adds write info (robot_id: 1, val: 11) into the TX;
7. customer sends a read request for robot_id 2 to RM;
8. RM returns kv_value for robot 2 to the customer;
9. customer receives read response (bid: 20, customer_id: 40, version: 28);
10. customer adds read info (robot_id: 2, ver: 28) into the TX;
11. customer adds write info (robot_id: 2, val: 21) into the TX;
12. customer sends a read request for robot_id 3 to RM;
13. RM returns kv_value for robot 3 to the customer;
14. customer receives read response (bid: 5, customer_id: 30, version: 12);
15. customer adds read info (robot_id: 3, ver: 12) into the TX;
16. customer adds write info (robot_id: 3, val: 6) into the TX;
17. customer sends the TX to the RM;

18. RM checks each and every tx_reads in the TX and compares its version number against the kv_value in the kv_table. If no kv_value has higher version number than the tx_reads, the RM applies all tx_writes to the kv_table (tx commit). Otherwise, simply discard the TX (tx abort);

19. RM responds back whether the TX committed or aborted;

20. customer receives the response.

Assume that customer 16 is trying to bid for robot 1, 5, and 6 concurrently with customer 10 above. Let's assume that customer 16 successfully committed the TX after customer 10 reads the bidding information. Then when customer 10's TX is sent to the RM, the RM should abort the TX, because there is a read-write conflict for robot 1.

1.4 Client program

The client program either sends read requests or TX requests. The client program creates N customer threads. The customer thread connects to the RM and issues T TX'es in a loop to the RM. For each TX, the customer thread issues 3 reads for 3 "distinct random" robots within the key range and creates three corresponding bids (writes) in the TX as in Section 1.3. Each customer thread should first establish a connection to the RM and execute the steps in Section 1.3 in loops T times.

1.4.1 Command line argument

The client program should take ip and port of the RM, the key range, number of customer threads, and the number of TX'es that each customers should issue. Finally, there are different request mode as in assignment #2: 1 - tx requests or 3 - special print command. For example,

```
./client [RM ip] [RM port] [range start] [range end] \
         [# customers] [# req] [req type]
./client 127.0.0.1 12345 0 19999 2 100 1
```

means client will connect to port 12345 of server ip 127.0.0.1, and 2 customer threads will issue 100 TX'es each to key range (robot id range) 0-19999.

1.4.2 Printing stats for request type 1

From the client program you should measure and print,

1. The number of transactions that have successfully committed.

2. The number of transactions that have aborted.

3. The transaction commit rate: (# of committed transactions) / (# of committed + aborted transactions).

4. The transaction throughput: how many transactions are processed per second (all committed and aborted transactions / second).

5. The transaction goodput: how many transactions are successfully committed per second (committed transactions / second).

You can ignore the stats for read operations and focus only on the TX requests for this assignment.

1.4.3 Special print request - request type 3

Similar to assignment #2, you should implement a special request type to print the kv_table on the screen.

```
./client 127.0.0.1 12345 1000 2000 1 100 3
```

The above command translates to creating a single customer thread which issues 100 read requests to key range from 1000-1099 and printing the key and the returned bid, customer_id and version in the following format.

```
key [tab] bid [tab] customer_id [tab] version
1000    10      3       13
1001    3       1       5
1002    22      17      30
...
```

1.5 Standalone resource manager (RM)

The base code for the server program has an engineer thread and the expert engineer thread. In this assignment, we will call them the worker thread and the transaction thread, respectively. Similar to keeping only one admin thread for assignment #2, we will always keep only one TX thread per RM.

1.5.1 Command line arguments

The RM should take three parameters

1. Port number.
2. Number of key-value pairs in the table.
3. Base (starting) key number.

For example,

```
./server 12345 10000 20000
```

means the RM is listening to port 12345, it maintains 10000 key-value pairs, and the key range is from 20000 to 29999.

1.5.2 Resource manager (RM) functions

An RM is a stand alone key-value store that works based on four functions.

- Read: returns the bidding information (kv_value) corresponding to the robot_id from the kv_table (step 3, 8, and 13 in Section 1.3).
- Prepare(struct tx): evaluates the TX and returns whether the TX can commit or not (step 18 in Section 1.3).
- Commit(): applies the last received TX to the key-value table (step 18 in Section 1.3).
- Abort(): aborts the last received TX (step 18 in Section 1.3).

The commit/abort calls will be triggered by an external message in a later section, but you should implement them as local function calls in this section. Implementation details are described as follows.

Read If the customer sends a read request, the worker thread will find the kv_value by looking up the kv_table using the robot_id. kv_base value should be used to determine the correct index to access the value corresponding to the robot_id.

Prepare Multiple customers may issue TX concurrently to the RM, but the TX will be processed one at a time by the TX thread.

1. When a customer sends a TX request, the worker thread receives a TX.
2. Then the worker thread sends the TX to the TX thread and waits for a response from the TX thread.
3. The TX thread maintains a local version number which is incremented by 1 whenever it receives a TX from the worker thread. The TX thread assigns this number as the version number of the TX (this number is used in the commit function).
4. The TX thread then inspects all read entries in the TX and checks whether the key-value pairs of the reads in the kv_table are updated as in step 18 of Section 1.3. The TX thread makes a decision for the fate of the TX.
5. The TX thread calls the commit or abort function call (see below for more details) depending on the decision made in the previous step.
6. Then the TX thread responds back to the worker thread with fate of the TX (committed or aborted).
7. The worker thread responds back to the customer with the result from the TX thread.

Commit/Abort Commit function applies all write requests within the TX to the kv_table. The version number of the written kv_value uses the version number of the TX. Abort function call simply discards the TX and does nothing else.

1.5.3 Protecting the key value pairs with locks

Worker threads and the TX thread concurrently access the key-value pairs within the same server. The data should be properly protected with synchronization primitives.

1.6 Data, stubs and message implementation

You should implement all data, stubs, and messages explained above. Different from assignment #1 and #2, the worker thread receives messages that have significantly different structures and lengths: a read request may include a single robot_id and a TX request may include a transaction struct that includes many data components. You may add an identifying integer at the beginning of the client request message buffer so that the RM can "recv" the first integer and figure out which type (e.g., read vs prepare) of message is incoming. Then the RM can "recv" the correct size of the message and unmarshall the data accordingly.

For example, if the read request message size is 16 bytes and the transaction request message size is 64 bytes you may send 20 bytes and 68 bytes messages for each request, respectively. For read request, the first 4 byte (size of an integer) will indicate that it is a read request and the rest 16 byte will be the read request body. The same applies to the transaction request but it will have a 64 byte TX request body. If all requests include the request type in the first 4 bytes of a message,

the server may first call recv(socket, buffer, 4) to figure out which request has arrived and then depending on the request type call recv(socket, buffer, 16 or 64).

1.7 Test your program

If you increase the number of customers and decrease the key range, the transaction commit rate will decrease. Change the parameters to observe such trends. Set the RM to have a large key range, send requests to a small subset of the range using the client, and check if the requests are delivered to the correct range using request type 3. Thoroughly test your program to make sure that there are no bugs/race-conditions.

2 Partitioning the standalone resource manager (RM) and adding a transaction manager (TM)

Now that you have implemented a standalone RM (server) that processes transactions, you should modify the code to create the TM (tm) and the distributed version of RM (rm) that works with the TM. You need to copy the standalone RM code and divide the role in the standalone RM into to the TM and the distributed RM. Similarly, you should create "2pc-client" based on the code for the "client."

2.1 Distributed resource manager (RM)

The distributed RM is almost identical to the standalone RM, but has a few key differences.

1. Both TM and the customers connect to the distributed RM. However, both connections can be handled by the worker thread as is. The customers only sends read requests and the TM sends prepare/commit/abort requests.

2. The TX request comes with a version number assigned by the TM and the distributed RM no longer assigns version numbers to the TX'es.

3. The distributed RM responds back to the TM with its local transaction decision for a prepare request and executes a TX commit/abort depending on the TM's final decision.

Because there is only one TM in the entire system and the TM processes one TX at a time (see below for more details about the TM), the transaction thread in the distributed RM has at most one pending TX request in its request queue (customers no longer send TX requests directly to the distributed RMs and sends the TX request to the TM).

For simplicity, we will assume that the TM sends the same TX instance to all distributed RMs. Thus, a TX may include read/write information for the keys which may not belong to the distributed RM. The RM should only evaluate and apply the keys corresponding to its own range. This an inefficient design because it sends unnecessary data to the RMs. In a real system, only the read/write information that is managed by the RM will be sent.

Prepare Now the prepare for the distributed RM works as follows:

1. Th TM sends a TX request and the worker thread receives a TX.

2. Then the worker thread sends the TX to the TX thread and waits for a response from the TX thread.

3. The TX thread stores the TX as a local variable.
4. The TX thread then inspects all read entries in the TX and checks whether the key-value pairs of the reads in the kv_table are updated as in step 18 of Section 1.3. The TX thread makes a decision for the fate of the TX.
5. The TX thread responds back to the worker thread with the TX decision.
6. The worker thread responds back to the TM with the decision.

Commit/abort Now the commit/abort function call is triggered by the TM's request.

1. The TM sends a commit/abort request, and the worker thread receives this final TX decision.
2. Then the worker thread sends the decision to the TX thread and waits for a response from the TX thread.
3. The TX thread either commits or aborts the stored TX.
4. The TX thread responds back to the worker thread with a job completion signal.
5. The worker thread responds back to the TM with the job completion signal.

Read Reads are handled the same as in the standalone RM.

Command line argument The command line argument stays the same as the standalone RM. However, when multiple RMs run together, they should cover continuous non-overlapping ranges: e.g., RM0 covers 0-99, RM1 covers 100-199, RM2 covers 200-299 and so on.

2.2 Transaction manager (TM)

The TM takes customers' transaction requests and relays them to the RMs. However, it does not take any read requests. TM communicates with all RMs only once before responding back to the client request: the response to the client request is done in parallel with the commit/abort phase because we assume no failures.

Similar to the RM, most TX processing functionalities in the TM should be implemented in the TX thread.

Command line arguments The TM should communicate with all RMs so it should take all information about the RMs. It should be able to communicate with arbitrary number of RMs depending on the configuration.

```
./tm [port] [# RMs] (repeat [ip] [port] [# kv pairs] [base key])
./tm 12345 3 11.11.11.11 11111 100 0 22.22.22.22 22222 100 100 \
     33.33.33.33 33333 100 200
```

The example above shows a TM with its port number 12345. This TM connects to three RMs with ip 11.11.11.11, 22.22.22.22, and 33.33.33.33, with port numbers 11111, 22222, and 33333, respectively. Each RM maintains 100 keys each ranging from 0-99, 100-199, and 200-299.

TM operations When the TM starts up, it creates the single TX thread similar to the standalone RM. The TX thread should establish socket connections to all RMs so that it can send the TX prepare/commit/abort requests.

The TM should work as follows:

1. When a customer sends a transaction request, the worker thread receives a TX.
2. Then the worker thread sends the TX to the TX thread and waits for a response from the TX thread.
3. The TX thread maintains a local version number which is incremented by 1 whenever it receives a TX request from the worker thread. The TX thread assigns this number as the version number of the TX.
4. The TX thread sends the TX (prepare) request to all RMs.
5. The TX thread waits for decisions from all RMs.
6. Once all decisions have arrived, the TX thread evaluates whether the TX can commit. If all answers are "commit" the final decision is a commit, otherwise the final decision is an abort.
7. Send the final decision to the worker thread (so that it is sent to the customer) and to all RMs (so that they commit/abort the TX).
8. The worker thread responds back to the customer with the decision, and the TX thread waits for and receives job completion signals from all RMs.

The TM multicasts the TX prepare/commit/abort requests to all RMs. To do this, you can call multiple send requests to different RMs in a row and then call multiple recv requests in a row. This should be hidden under the prepare/commit/abort stub and the prepare stub should return the final decision based on the collected decisions from the RMs.

2.3 2PC Client

Different from the client for the standalone RM, the 2pc-client program now connects to multiple RMs and the TM. The 2pc-client works the same as the old client except that the read requests are sent to specific RMs depending on the read key and the TX requests are sent only to the TM. Similarly, the read requests for the special print request (request type 3) should be sent to the RMs that store the key that the customer is trying to read.

The command line argument should be changed such that it takes the TM information, all RM information, the request key range (start and end), the number of customer, and the number of TX requests per customer:

```
./client [TM ip] [TM port] \
         [# RMs] (repeat [ip] [port] [# kv pairs] [base key]) \
         [key range start] [key range end] [# customer] [# tx] \
         [request type 1 or 3]

./client 12.34.56.78 12345 \
         3 11.11.11.11 11111 100 0 22.22.22.22 22222 100 100 \
         0 199 8 100 \
         1
```

Note that the key range should not exceed the range covered by all RMs.

3 Playing with the implementation

All measurement should be done using Amazon EC2. You may use the Ohio region in addition to the N.Virginia region. To use the Ohio region you should create a new key-pair. Make sure you TERMINATE (shutting down a machine is not enough) all instances after each use. Make sure your program does not print any messages on screen during execution (it is okay to print at the end of the execution).

Standalone RM experiments (2 graphs) Run 1 standalone RM (server) and 1 client in different VMs. Use different numbers of key-value pairs in the standalone RM: 16, and 32768. For different numbers of key-value pairs per RM, run different numbers of customer threads in your client programs: 1, 4, 16, and 64. Run each experiment for over 10 seconds.

2PC experiments (2 graphs) Run 3 RMs, 1 TM, and up to 2 clients in different VMs. Use different numbers of key-value pairs stored in each RM: 16, and 32768. For different numbers of key-value pairs per RM, run different numbers of customer threads in your client programs: 1, 4, 16, and 64. Run each experiment for over 10 seconds.

What to plot Measure the throughput and goodput for the above experiments, and plot them. The x-axis is the number of customer threads in each client program, and the y-axis is the throughput/goodput.

4 What to submit

Submit the below in a zip file. Make sure that you place the files in a folder and zip the folder.

1. Maximum two-page report in a pdf format. Summary of your software design, what works and what doesn't work, if any, and how to run your binary, if it is different from the specification above. Add four graphs that are explained in the previous section and add a short description for each graph.
2. All source code files with a Makefile. The code should compile with a "make" command to create all "server", "client", "tm", "rm", and "2pc-client" programs.

5 Due date

11/21/2020 (Saturday), 11:59 pm.

6 Grading - 5 points

The points work as bonus points for assignments #1 and #2: it only fills in missing points for the assignments #1 and #2.

1. Standalone RM and its client (1 pt).
2. Standalone RM and client interaction tests (1 pt).
3. TM, distributed RM, and 2PC client (1 pt)

4. 2PC components interact correctly (1 pt).
5. Report (1 pt).