CS 6650 Assignment 3

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 replicated distributed service. You will use a primary-backup protocol to implement a state machine replication approach to replicate an in-memory key-value store on multiple distributed nodes. You will learn

- how to implement a state machine replication approach.
- how to implement a primary-backup (or to be more precise, a client-driven chain replication) protocol to replicate state machines.
- how to implement multi-threaded servers and clients that share a key-value store.
- how to seamlessly handle node failures.
- (optional) how to repair a failed subsystem.

Robot factories under economic depression

Your robot business went well and now you have multiple factories which produce the same robots. However, the demand for special robots has decreased, so you no longer produce special robots and do not maintain expert engineers. Instead, you hired few administrative employees who take care of customer records. However, due to a recent economic depression, you are changing the way how you run your factories.

To save cost, you only run one factory out of many to produce robots. We will call the running factory as the *production factory*. Although you only have one production factory, you keep your customer records up-to-date in all factories: whenever customer record changes in the production factory, the updates are sent to all other factories. We will call the other factories *idle factories*. In case there is a problem in the production factory, an idle factory can become the production factory.

Your main task for the assignment is to realize the factories that keep customer records replicated while producing robots. You will implement both production and idle factory functions in the same server program. The server programs will use primary-backup and state machine replication approach to keep the customer record replicated. You should start implementing the factories based on the given code or the code that you worked for assignment #1.

1 Create a state machine log and customer records

You learned that a state machine replication approach is good for keeping data consistent across multiple locations. However, Paxos or Raft like protocols that typically support the state machine approach are too complicated to maintain, so you want to implement the state machine approach based on simple primary-backup protocols. As the first step, you will implement necessary structures in a single server setting. Create a state machine log that records map (i.e., key-value store) operations and a map that executes the operations in the log.

1.1 Add a customer-record-access workflow in the factory

Your customer records are stored in a map and the state machine log uses a vector to store map operations (detailed in the next subsection). Note that write or update operations to both the map and the log are done only by the admins in a single factory setting (this will change later). However, engineers can read the customer record (i.e., the map, but not the log) upon a customer's read request. Note that because the customer record is accessible by multiple engineers and the admin at the same time, proper synchronization is necessary to avoid race conditions.

The admin follows almost identical workflows as the expert engineers (you can reuse the expert engineer thread in assignment #1 and convert it into the admin thread), and there will be exactly one admin in a factory.

1. The robot manufacturing workflow should now be:
    (a) A customer sends a robot order to the (regular) engineer.
    (b) The engineer creates robot (fills in robot info).
    (c) The engineer sends request to the admin to update the customer record and waits.
    (d) The admin who was waiting for requests fetches the request.
    (e) The admin appends a necessary operation to update the customer record to the log.
    (f) The admin then applies the logged operation to the customer record map.
    (g) The admin notifies record update completion with its id to the waiting engineer.
    (h) The engineer wakes up and ships the robot to the customer.
2. The customer record reading workflow should be:
    (a) A customer sends a record read request for its own record (a customer cannot access other customer records by default, but you will break this rule later).
    (b) The engineer reads the corresponding record from the customer record map.
    (c) The engineer returns the record to the customer.

1.2 Data formats

The necessary data formats are outlined below. When implementing the data structures and accessing them, you should make sure they are properly protected using synchronization primitives as needed.

1.2.1 Customer records and state machine log

You will use std::map and std::vector to implement the customer record and the state machine log, respectively.

- std::map - `http://www.cplusplus.com/reference/map/map/`
- std::vector - `http://www.cplusplus.com/reference/vector/vector/`

Customer records Customer records should be in the following format:

```
std::map<int, int> customer_record;
```

The key of the map is the customer id, and the value is the last order number of the customer.

State machine log The log entry and the log for the state machine should use the following format:

```
struct MapOp {
        int opcode; // operation code: 1 - update value
        int arg1;   // customer_id to apply the operation
        int arg2;   // parameter for the operation
};
std::vector<MapOp> smr_log; // state machine replication log
```

MapOp is a general structure that defines the operation that can be applied to the map and parameters for the operation. The potential to have multiple opcodes defined gives you the idea to host complex data structures with many different operations in the state machine log, but the map is very simple. You will only use opcode 1 which updates the map. For example, if opcode is 1, arg1 is 100, and arg2 is 20, then this means that find the customer record for customer number 100 in the map and update the value to 20 (if the customer record does not exist, create a record for the customer number 100 and assign the value 20).

1.2.2 Customer request, robot info, and customer record

Customers now send customer requests and receive either robot information or customer record depending on the request type.

Customer request The customer's request follows the same structure as the robot order in assignment #1, except that the robot_type is renamed to request_type. The request_type can have two values: 1 - order a regular robot; and 2 - read a customer record. The request should include:

```
int customer_id;    // For robot order: customer id who placed order.
                    // For read request: customer id to read from map.
int order_number;   // # of orders issued by this customer so far.
                    // Record-read request does not count as an order.
                    // Set to -1 for read requests.
int request_type;   // Either 1 - regular robot order request, or
                    //        2 - customer record read request
```

Robot information The robot information structure that is returned to the customer for robot order requests stays the same as in assignment #1, but robot_type and expert_id are renamed to request_type and admin_id, respectively.

```
int customer_id;    // copied from the order
int order_number;   // copied from the order for request_type 1
                    // copied from the record map for request_type 2
int request_type;   // copied from the order
int engineer_id;    // id of the engineer who processed the request
int admin_id;       // id of the admin who updated the record map
                    // -1 indicates that the request was a read
```

Customer record The customer record is returned for customer record read requests. The customer record should include the following information:

```
int customer_id;    // copied from the read request
                    // -1 if customer_id is not found in the map
int last_order;     // copied from the map
                    // -1 if customer_id is not found in the map
```

Client and server stubs You should change and add client and the server stubs as follows:

- ClientStub.Order should take a customer request and return robot information.
- ClientStub.ReadRecord should take a customer request and return a customer record.
- ServerStub.ReceiveOrder should be renamed to ServerStub.ReceiveRequest and return customer requests.
- ServerStub.ShipRobot should take robot information and send the robot information.
- ServerStub.ReturnRecord should take a customer record and send the customer record.

1.3 Command line arguments

The command line argument formats for the client and the server programs stay the same as assignment #1. However, the allowed request types (i.e., former robot type) are 1, 2, and 3 (see below) for the client program and the allowed number of admins (i.e., former number of expert engineers) is 1 for the server program.

1.4 Print the customer record from the client program

You may want to make sure that the server program correctly updates the customer record. To check this, you should add a request type 3 workflow to the client program. Request type 3 only exists as a command line argument for the client program. It is a special command that lets a customer thread to scan through all customer records and print them on the screen. Request type 3 workflow does not require server program modifications at all, because it only affects the client program behavior.

When request type 3 is given as the command line argument for the client program, each customer thread issues record read requests as many as # orders to scan all customer records and prints the returned customer record from the server on the screen. We will assume that customer ids are linearly assigned (e.g., if you run 100 customers, ids can start from say 11 and end at 110), and the # orders argument will set the maximum customer id to scan for.

For example, if the client program is executed with the command below,

```
./client 123.456.789.123 12345 1 128 3
```

the client program will spawn one customer thread, and the customer thread will send 128 read request to the server where the customer_id varies from 0 to 128, order_number is fixed to -1, and request_type is fixed to 2. If the server returns customer_id = -1, it means the record does not exist, so you can skip printing the record. If a valid record is returned you should print them in the following format:

```
// customer_id [tab "\t"] last_order
11      99
12      99
13      99
...
```

Since all customer threads will be issuing the same request redundantly, it is recommended that you use only 1 customer thread when using the request type 3.

Using request_type 3, you can read what is stored in your server. For example, you can run the client program with request_type 1, and then run the program again with request_type 3 to check whether the server is holding the desired customer record.

1.5 Before you move on to the next step

Thoroughly test your server program with the client program (e.g. by reading records after placing orders, or placing orders and reading records at the same time by running multiple client programs at the same time).

2 Replication without failure considerations

To replicate customer records from the production factory to idle factories, you will implement a primary-backup protocol (or client-driven chain replication) as described below. You can regard the primary node as the production factory and backup nodes as idle factories. We assume no failures at the moment.

2.1 Server program

You will mostly work on modifying the server program to implement the primary backup protocol. The server program that plays the primary role acts like a client that issues requests and the server program that plays the backup role acts like a server that processes the request.

2.1.1 Command line arguments

The number of admins per factory is fixed to 1 so the server program does not need to take the relevant argument. However, there are many additional command line arguments that the server program should take. The following summarizes the set of arguments that are necessary:

1. Port number,
2. Unique ID (≥ 0) of the factory,
3. Number of peer factory servers, and
4. Unique ID, IP, and port of all peer factory servers.

That is, the execution of the server program should follow the format,

```
./server [port #] [unique ID] [# peers] (repeat [ID] [IP] [port #])
```

For example, let's assume you are running three servers with configurations:

- Server ID: 0, IP: 11.11.11.11, port: 12345

- Server ID: 1, IP: 22.22.22.22, port: 12345
- Server ID: 2, IP: 33.33.33.33, port: 12345

Then the three servers should run with the commands:

```
./server 12345 0 2 1 22.22.22.22 12345 2 33.33.33.33 12345
./server 12345 1 2 0 11.11.11.11 12345 2 33.33.33.33 12345
./server 12345 2 2 0 11.11.11.11 12345 1 22.22.22.22 12345
```

This allows servers to find each other, and send and receive data replication requests. Through the command line argument, you should be able to configure your servers to work with arbitrary number of peers. Note that the server ID, IP, and port mapping given to each server as arguments should be consistent across all servers.

2.2 Protocol

Different from Section 1, you will use multiple server nodes. Also, MapOps in the smr_log are not applied to the customer_record immediately, but are applied only after the MapOp has been replicated to all nodes; we regard such fully replicated MapOp as "committed." In addition to the smr_log and the customer_record, all nodes keep track of the last written index of the smr_log, up to which index the MapOps in the smr_log has been committed, and which server is the production factory:

```
int last_index;        // the last index of the smr_log that has data
int committed_index;   // the last index of the smr_log where the
                       // MapOp of the log entry is committed and
                       // applied to the customer_record
int primary_id;        // the production factory id (server id).
                       // initially set to -1.
int factory_id;        // the id of the factory. This is assigned via
                       // the command line arguments.
```

Because of communication delays, the information in each node can be slightly different. You may add additional variables and data structure as needed.

Any factory can become the production factory and you as the owner of the factories should tell the customers to send robot order to only one of the factories (You should configure the client program to send the robot order to only one of the server programs). The factory that receives the robot order automatically becomes the primary factory and you should assume that there is only one primary factory at any given moment: e.g., you can have the client program to send 100 robot orders to factory 0 and later in time you can have the client program to send another 100 robot orders to factory 1, but the client programs should never send robot orders to factory 0 and 1 at the same time. However, client programs can send record read requests to any servers (both primary and backup servers) concurrently at anytime.

The replication protocol itself (does not include the robot creation process) that you should implement, works as follows assuming no failures at all.

1. Primary receives customer record update request.
2. If primary_id of the primary is not set to its own id, primary sets primary_id to its own id and establish connections to all backup nodes.

3. Primary appends the request to its own log and updates last_index accordingly.

4. Primary repeats the following to all backup nodes in a sequence.

   (a) Primary sends replication request "factory id, committed_index, last_index, MapOp" to the ith backup node.

   (b) The ith backup node,

      i. Sets primary_id if it is not pointing to the current primary;

      ii. Writes MapOp to req.last_index of its smr_log and update self.last_index;

      iii. Applies MapOp in the req.committed_index of smr_log to the customer_record and update self.committed_index; and

      iv. Respond back to the primary.

   (c) Primary receives response from the ith backup node.

5. Once primary successfully communicates with n backup nodes, it applies the MapOp of last_index in the smr_log and assigns the last_index value to committed_index.

2.3 Roles within the server

The primary node role described above should be carried out by the admin thread that is described in Section 1 and the backup node role should be handled by a new admin thread which is detailed below. In short, the server program should implement three roles:

1. Engineer: handles client requests. For robot order requests, the engineer communicates with the admin to produce robots. For customer record read request, the engineer directly accesses the customer record map and returns the requested record.

2. Production factory admin (PFA) : the admin of the production factory that sends customer record replication requests to idle factory admins. In an idle factory, PFA is inactive and does nothing.

3. Idle factory admin (IFA): the admin in an idle factory that handles PFA requests. The IFA in the production factory is inactive and does nothing.

The implementation of your factory in Section 1 partially implements engineer and PFA roles already, and you will have to implement the IFA role from scratch.

If you carefully review the roles, you will notice that the engineer and the IFA carry out server-like operations (respond to requests) and the PFA carries out client-like operations (send requests). Therefore, you can implement engineer and IFA roles within the same thread and the PFA role in a separate thread.

2.3.1 Role 1: engineer

If you have completed the tasks in Section 1 the engineer role should be already implemented in the regular engineer thread from assignment #1.

However, you should implement the IFA role in the same thread, so you will need to make changes to the engineer thread code (but not the code for the engineer role itself) accordingly. See Section 2.3.3 for the IFA role that should be implemented in the engineer thread side by side with the engineer role.

2.3.2 Role 2: PFA

The PFA role can be implemented in the existing admin thread. The replication protocol for the primary can be implemented between 1.(e) and 1.(f) in the workflow of Section 1.1.

The PFA mainly communicates with the IFAs over the network to replicate customer records. Similar to how your client program connects to the server, the PFA creates socket connections to idle factory servers and use the socket connections for communication. Because the PFA thread does not yet know if it is in the production factory when the server program starts, the PFA should make connections to the other servers only after it receives the first customer record update request from the engineer thread. When the production factory no longer receives any more requests, the PFA may maintain the socket connection until the server program terminates and even if another idle factory later becomes the production factory.

2.3.3 Role 3: IFA

The IFAs communicate with the PFA to replicate data. The PFA will establish connections to the idle factory servers and because each server maintains only one open port for the socket connection, you should figure out whether the PFA or the customer connected through the port for each connection. Therefore, it makes sense to create the IFA as part of the engineer thread in Section 1 and have the customer and PFA to identify itself by sending an one-time message. Depending on who is connected to the thread, you should respond with either the engineer or the IFA role.

Once the thread figures out that its role is the IFA, it should wait for the PFA's replication request and respond to it. Note that the IFA should only apply committed entries in the log to the customer record map. Because there can be customer's read requests to the map through the engineer thread, the map should be properly protected with locks.

2.4 Client program

The client program mostly stays the same as what you have created for Section 1, but it should now send an identification message (see Section 2.3.3) after connecting to the server and before issuing the first request.

2.5 New stubs and communications

Similar to assignment #1, you should implement necessary stubs that hide the communication details for all network operations. For example, stubs should be created for the send and receive messages in 4.(a) and 4.(c) of Section 2.2 and also for the PFA/customer identification message in Section 2.3.3.

2.6 Some subtleties to handle

Note that the production factory will always have one more committed updates than the idle factories because the committed index is incremented only after the production factory makes sure all idle factories replicated the update request.

Therefore, after customers finishes sending a set of robot orders, the production factory's committed_index and last_index should both be x, while the idle factories' committed_index is $x - 1$ and the last_index is x.

If you switch production factory to one of the idle factories, the new production factory should append the new incoming MapOp at log index $x + 1$, change the committed_index to x, apply xth log entry to customer record, and resume replicating the new MapOp at log index $x + 1$.

2.7 Before you move on to the next step

Thoroughly test your server programs with the client program. Run multiple client programs concurrently to read records from both production and idle factories while some client programs are writing to the production factory. Once you run the client program through a production factory, switch the production factory to another factory and resume replicating the data. Use request type 3 to check the data in all servers.

3 Replication with failure considerations

Now that you have a fully replicated system under no failures, you should start preparing for server failures. You will assume fail-stop model and modify your code accordingly. Your programs should be functional as long as at least one server is up and running.

Similar to Section 2, you should assume that there are fixed number of factory server programs that work together to replicate the data. You should assume both primary and backup servers can fail at any time (i.e., consider forcefully terminating a server with Ctrl+c). Server termination/failures can be noticed by socket errors, such as recv()/send()/connect() errors. Refer to Linux man pages to find out the type of errors and how to react to them: `https://linux.die.net/man/`; search for recv, send, connect, etc.

(Bonus points) Once the failure is detected, you can repair the failed server and have the repaired server to rejoin the live servers (i.e., restart the server with the same configuration). The repaired server should recover missing data from peer servers and participate in the replication process for the up-to-date requests.

3.1 Handling failures

You should first design your programs to handle failures gracefully without crashing. The server programs should not print error messages for other server failures and treat such failures as normal operations. You will mostly modify your code to handle failure cases for socket-related function calls.

3.1.1 Primary server failure

If the primary server fails during an execution, the replication process will stop and clients connected to the primary server won't be able to issue any requests.

In case the primary server fails,

- The backup servers should terminate their IFA threads and set the primary_id to -1.
- The client program that was communicating with the primary server should stop sending any more requests and terminate gracefully: the client program should terminate normally without crashing.

After the primary fails, you should be able to choose one of the backup nodes to become the next primary and have the client program to send robot order requests to it. The remaining servers should be able to replicate data with the new primary while the old primary node remains dead. The new primary will try to connect to the failed primary to replicate data, but the connection will fail. The new primary can treat such failure as a backup failure that is described below.

3.1.2 Backup server failure

If the backup server fails during an execution, the replication process led by the primary should not stop. However, the client that was issuing record read requests to the failed backup server won't be able to issue any more read requests.

In case a backup server fails,

- The primary server sets the socket corresponding to the failed backup server to -1. During a replication process, the primary will ignore the failed backup server and treat the MapOp as committed if the MapOp is replicated to all live backup servers.
- The client program that was communicating with the backup server should stop sending any more request and terminate gracefully: the client program should terminate normally without crashing.

3.2 Handling recovery (Optional for bonus points)

You can get full points for the assignment if you complete up to handling failures. Handing recovery counts toward bonus points that can fill in missing points in programming assignments #1 and #2.

You should now consider failed servers being repaired and joining the live servers. To make the problem simple, we will assume that the repaired server will rejoin when no customers are interacting with the live servers. "Repaired" doesn't mean all data being recovered, but simply restarting the failed server program with the same command line arguments as the failed instance.

Again for simplicity, you should assume that there is always a primary node among the live servers when the repaired server restarts. The primary should always try to connect to the failed servers when it replicates customer records. Thus, if the repaired server program restarts, the primary should be able to connect to the repaired server. Before issuing the latest replication request, the primary should send all existing log entries one by one to the repaired server so that the data state in the repaired server becomes up-to-date as rest of the backup servers. The primary then resumes replicating the incoming requests.

Implement the above and test for various failure cases: e.g., killing all of your backup servers and bringing them back up.

4 Playing with the implementation

Let your client program run for a long time with a small number of customers and check if the servers maintains the same data. Kill the server nodes while executing customer requests. Your live servers should be able to maintain the replicated data and process robot orders as long as one server instance is alive.

This assignment focuses on the correctness of the implementation rather than the performance, but it is worth to measure the performance. You should measure the performance under no failed servers.

Write performance Change the number of server programs from 1 to 3. Use 8 customer threads in the client program. Run one client program for 10 seconds or longer by changing the number of robot orders and measure the mean latency (microsecond) and throughput (orders/second).

Read performance Use one client program to fill in the server programs with customer data: use 128 customers that issue 4 requests each. Then, for each server program, run a client program that uses 128 customer threads to issue read requests (request_type 2) for over 10 seconds. Change the numbers of server and client programs from 1 each to 2 each and let the client programs run concurrently. Sum up the throughput from 1 to 2 client programs depending on the experimental set up.

All measurement should be done using Khoury Linux machines. Make sure your program does not print any messages on screen during execution (it is okay to print at the end of the execution).

5 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. If you have implemented the recovery case, clearly describe how you implemented it. Add 1) write performance and 2) read performance graphs as described in Section 4 with a short explanation.

2. All source code files with a Makefile. The code should compile with a "make" command to create both "server" and "client" programs in Khoury Linux machines.

6 Due date

10/30/2025 (Tuesday), 11:59 pm. Refer to the syllabus on Canvas for the late policy.

7 Grading - 15 points (+2 bonus points will fill in missing points from this and other programming assignments.)

1. State machine log and customer records (3 pt).

2. The client program works as described (3 pt).

3. Customer records are replicated under no failures (3 pt)

4. Customer records are replicated under injected failures (3 pt)

5. Repaired servers can rejoin the live servers and missing data is recovered (bonus 2 pt)

6. Report (3 pt).