Vinson Chen

# Assignment 3: Load Balancer Design Document

This project has an assumption that there is a server with logging working. The purpose of this project is to be able to forward to multiple different servers. This assignment will add:

1. Forwarding
2. Load Balancing

Before the documentation starts explaining the logic behind the program, there are a few data structures and functions created to support this program.

Data Structures Created:

1. Queue

    This data structure is used to pass information between the dispatch thread and the worker threads. The queue is a single linked list. Below is the data structure of the queue.

```
//queue.c
typedef struct NodeObj{
   int data;
   struct NodeObj* next;
} NodeObj;

typedef NodeObj* Node;

typedef struct QueueObj{
   Node front;
   Node back;
   int length;
   pthread_mutex_t * mut;
} QueueObj;

//queue.h
typedef struct QueueObj* Queue;
```

    By hiding the `NodeObj`, `Node`, and the `Queue` in queue.c, the program can make those three datatypes hidden from the rest of the program, thus mimicking a class function. Since this program is multi-threaded, the queue needs a mutex to lock the Enqueue and Dequeue for atomic modifications made on the queue. The four functions used for the queue is:

```
int getLength(Queue Q);
```

This function gets the length of the queue.

```
pthread_mutex_t* getMutex(Queue Q);
```
This function returns the mutex in the queue.

```
void Enqueue(Queue Q, int data);
```
This function appends the data to the back of the queue. Note that Enqueue is locked with the mutex on function start and unlocked with the same mutex on function end.

```
int Dequeue(Queue Q);
```
This function deletes the data from the front, and returns the data before being deleted. Note that Dequeue is locked with the mutex on function start and unlocked with the same mutex on function end.

Queue function was borrowed from Patrick Tantalo on
https://classes.soe.ucsc.edu/cse101/Fall19/Examples/C/Queue/Queue.c
https://classes.soe.ucsc.edu/cse101/Fall19/Examples/C/Queue/Queue.h

2. ConnectObj

The purpose of this data type is to contain the contents of a server that the forwarding is connected to. This data structure is defined below:

```
struct ConnectObj{
  int identifier;
  int port;
  int inc;
  int total;
  double percentage;
  int alive;
};
```

- `int identifier` is the number associated with the server. This isn't really needed, it's just a way of determining which port to use when the both ports have the same total and percentage.
- `int port` is the number to the port. This is the number to connect to on a forward program to server program.
- `int inc` is the number of incorrect requests sent from a health check.
- `int total` is the number of total requests sent from a health check.
- `double percentage` is defined as (total - inc) / total. This will be used as a tie-breaker if the total amount is the same.
- `int alive` is an integer that determines whether or not the server is alive from a health check.

3. Health Check Object

    The purpose of this data type is to contain the requirements for the Load Balancer thread as well as the worker thread. This data structure is defined below:

```
typedef struct healthcheck_obj{
  pthread_cond_t* hcCond;
  pthread_mutex_t* hcMut;
  struct ConnectObj* serverPorts;
  int serverPortSize;
  int* serverUse;
}HealthCheckObj;
```

- `struct ConnectObj* serverPorts` is the container of the ports information. This data is an array of ConnectObj.
- `int serverPortSize` is the size of the serverPorts.
- `int col` is the column of the array, also the max size of the file name.
- `pthread_cond_t* hcCond` is the pointer to the condition variable that wakes up the Load Balancer thread
- `pthread_mutex_t* hcMut` is the pointer to the mutex variable that locks so the Load Balancer and the worker threads will be atomic.

4. Thread Arguments

    The purpose of this data type is to be a compacted argument structure for each of the threads. There are two data structures: HCArgs and ThreadArgs. This data structure is defined below:

```
typedef struct healthcheck_args{
  int time;
  int check;
  HealthCheckObj* healthChecker;
}HCArgs;
```

- `int time` is the sleeping time in between health checks.
- `int check` is the number of requests before another health check.
- `HealthCheckObj* healthChecker` is the data type that is shared between HCArgs and ThreadArgs.

```
typedef struct thread_args {
    pthread_cond_t* thrCond;
    int identifier;
    Queue* workList;
    HealthCheckObj* healthChecker;
}ThreadArg;
```

- `int identifier` is the number associated with the thread. This is needed as opposed to `pthread_self()` due to the structure of file locking.
- `pthread_cond_t* thrCond` is the pointer to the condition variable with the Queue's mutex variable.
- `Queue* workList` is the Queue data structure defined from above. This allows the transfer of data between the dispatch thread and the worker threads.
- `HealthCheckObj* healthChecker` is the data type that is shared between HCArgs and ThreadArgs.

Now that the data structures created from the program have been stated, the document will talk about the structure of the program. From the top, there are 2 portions of the program: forwarding to many servers so they will all act as if there is one server, load balancing the servers so one server doesn't have too many requests.

1. **Main**
    a. **Purpose**

        The main function should be able to accept client requests and delegate the each client requests to different servers.

    b. **Functionality**

        The main for this program should be pretty much the same as the main from assignment 2. The program will be checking different flags.

        i. -N
            1. This determines the number of threads the server will have. If this is omitted, the server will have 4 threads. -N should be followed by an integer value.

        ii. -R
            1. This determines how many requests the clients will make before a force health check is called.
2. **Forwarding**
    a. **Purpose**

        The program will have one dispatch thread, and many worker threads to disperse the requests into different servers.

    b. **Components**

        i. Server Start (Dispatch thread)

            On server start, the program will only have one thread (the main thread). This thread will become the program's dispatch thread. The dispatch thread will create a `ThreadArgs` data structure needed for each of the new threads it will create. Almost all of the arguments in the data structure will be the same (I.E, pointers will point to the same variable, values will be the same number). The only difference between each of the data structures is `int identifier`.

            Once the `ThreadArgs` data structures have been defined, the dispatch thread will use a for loop to create N amount of threads using `pthread_create()`. Each thread will have a pointer to the queue to pass the data from the dispatch thread to the worker threads.

The dispatch thread will also create a health checker thread which will check the health of each server listed in the arguments. The health checker thread will use HCArgs instead. There is a small sleep of 0.01 to ensure the Health Check will reach the mutex before the worker threads. This way the program can guarantee that the Health Check can reach the lock first. In theory this should not be a problem, but this step was included to deter heisenbugs.

When the dispatch thread receives a curl request from the client, the dispatch thread will use the function `Enqueue()` to append the data to the queue and wake up the worker threads to process the request using `pthread_cond_signal()` with the conditional associated with the queue. The dispatch thread will also include an internal counter to determine how many requests have been sent to the queue. Once the counter reaches a mod of the threshold, -R argument, the dispatch thread will signal the Load Balancer.

ii. Threads running (Worker threads)

These worker threads will be in an infinite while loop. This is because the number of worker threads should be static (number of threads should not increase nor decrease). To begin working, each worker thread requires a request to process given by the dispatch thread from above. Since the process requires only one thread to handle each request, there will be a lock on the code that deals with processing data from the queue.

To deal with this anomaly, the program will introduce locks. This mechanism guarantees that one thread will be running the locked code at a given point in time. This is called the critical region. To start the critical region, the program will use a mutex to lock the code using `pthread_mutex_lock()` with the mutex associated with the queue. To finish out of the critical region, the program will use the same mutex to unlock the code using `pthread_mutex_unlock()` also with the mutex associated with the queue.

Inside the critical region contains two parts: checking when the queue is not empty, and getting the data from the queue. To check when the queue is not empty, the program can use a while loop and busy-wait or sleep until the queue becomes not empty. This is not optimal since every thread will be busy-waiting, thus all of the threads will use up most of the computer's computing power.

To solve this problem, the program will introduce conditionals. This allows each thread to sleep without using any of the computer's cpu until the condition has been met. To implement the conditional, the program will use `pthread_cond_wait()` using the conditional and the mutex associated with the queue. This will force the thread to sleep until a `pthread_cond_signal()` or a `pthread_cond_broadcast()` has been called with the same conditional.

The function, `pthread_cond_wait()`, will still be in a while loop as a confirmation that the queue is not empty. After the while loop, the thread will use the function `Dequeue()` to receive and delete the first request from the queue.

The worker thread now contains a client request to send to the server. At this point, the Load Balancer Thread (will be discussed later) will determine the best server to connect to. The worker thread will connect to this best server to forward the client's request to the server to process.

3. **Load Balancing**
   a. **Purpose**

      When there is a program that is forwarding to many different servers, the program needs to determine which server is the best server to send to.

   b. **Components**
      i. Health Check Probing

         On main, the program creates a load balancer thread. The goal is to check every server and then find the best server for the worker threads to connect to. Like the worker thread, this thread will be encapsulated in a while true loop. This means that the thread will never end. On the beginning of the while loop, there will be a mutex lock and a conditional wait inside. Instead of using a pthread wait, the program will use `pthread_cond_timedwait()`. This will allow the program to either be signalled due to the request count, or the timed wait.

         Once the load balancer is called from it's sleep, the thread will create a number of threads equal to the number of ports. Each thread created will check the health of one server. By checking the health of the server, the thread will: open the server connection, send it a health check request, wait for the response, parse the response for correctness, and store the value if it is correct.

         There will be three areas where the health of the server can be incorrect. The first is when `client_connect()` is -1. This means that there was an error in connecting with the server side. The second error can occur while you wait for a response. The thread can't tell the difference between slow or dead. This means, if the response takes too long, it's most likely dead. The last error can occur on an incorrect response. There are two types of incorrect response, status code and incorrect message. If there are no errors listed above, then the server is healthy. If the server is healthy, then the thread will update it's error and total values and leave it for the main Load Balancer thread to use.

ii. Choosing Best Server

Once the health check is complete, the main Load Balancer thread will combine all of the threads together using `pthread_join()`. This will be done in a for loop so the Load Balancing thread will wait until all of its threads are completed. Once they are completed, the total and incorrect values should be ready for processing. The load Balancer will check to see the lowest total number in a while loop to determine the best server. Once the best server is found, the thread is done with the Health Checking. One subtle design choice is the locking of the while loop. There will not be any mutex locks or unlocks. Instead, the thread will continue working until it sees another `pthread_cond_timedwait()`.

## 4. Summary

Main will be the same as before, except it will support -N and -l for number of threads and logging file name. Also main will create N worker threads to process the client requests. The request is passed in a queue using `Enqueue()`. The dispatch thread will wake the worker threads using `pthread_cond_signal()` where the worker threads will receive using `Dequeue()` in a mutex lock using `pthread_mutex_lock()` and `pthread_mutex_unlock()` and a `pthread_cond_wait()` inside a while the queue is empty loop. The dispatch thread will keep a counter to determine when to forcefully wake up the Load Balancer thread.

When the worker threads receive the client side socket from `Dequeue()`, the worker thread will connect to the server using http server protocols. The worker threads will then feed forward the request to the server and wait. After waiting a little bit, the worker thread will retrieve the response and return the response to the client.

In the Load Balancing portion, there are num_server amounts of threads created with each thread monitoring one server using a `pthread_create()`. All servers will end with a fail and not update, or pass and update. The main Load balancing thread will use `pthread_join()` to wait for and combine the thread it delegated. Once finished, the thread will choose the best server to use using a for loop. This will compare to see which server contains the smallest total number from the health checking.