Design Document: Load Balancer

Joshua Cheung
Cruz id: johcheun

5/30/2020

1 Goals

The goal of this program is to create a load balancer that will distribute connections over a set of servers that will be able to handle multiple requests concurrently, as specified by the assignment spec. We will also be implementing a periodic healthcheck that will be run every x amount of seconds, or until we have reached a certain amount of requests. This healthcheck will keep track of the performance of the servers, and will also be responsible for distributing requests equally among the servers.

2 Design

First, I will briefly go through the starter code functions provided for the assignment.

client_connect(port number)

- This function takes a port number to establish a connection to the client.
- Returns: the socket number that it is connected to, and if the connection failed then returns -1

server_listen(port number)

- This function takes a port number and creates a socket to listen on the port
- Returns: the socket number that it is connected to, and if the connection failed then returns -1

bridge_connection(socket from, socket to)

- This function takes two ends of connections and keeps track of number of bytes sent
- Returns: number of bytes sent, 0 if connection is closed, and -1 on error

bridge_loop(socket from, socket to)

- This function forwards all messages between both sockets until the connection is interrupted. It essentially does what was assigned in assignment 2, that it processes the requests and sends the respective messages back and forth between the client and server
- Returns: void
-

Next, we have our own implemented functions.

2.1 Main

1) First we use getopt similar to the last assignment to check for respective flags
    - -N flag for number of threads
    - -R flag for number of requests made before sending an additional healthcheck
    - First excess argument is the port number of the loadbalancer
    - Other excess arguments represent additional servers in which we will distribute requests
2) We then will loop through each server and send a healthcheck for each respective server, then we will store the healthcheck information into a struct and then pass it into a healthcheck thread.

3) Then we will create N amount or default amount 4 of worker threads that will be used to process requests made from the client that are using curl commands. For each thread we will link it to a server to do processing on from our dispatch function. If we cannot connect to the server, then we increment our variable that keeps track of which servers are down. If we find that all the servers are down, then we will later on send a 500 internal server error back to the client
4) If at least one server is up then we will then infinite loop to be able to accept client requests at any time. We will be going into this more in depth later, but we will be using a queue data structure for processing the requests in order, which will be distributed to each server. In our while loop we will also include a counter to check whether we have received R amount of requests, which we would then send a health check response on each server.

Algorithm 1: Main

```
Parse command line arguments
For each server:
        If pthread healthcheck accept port number
                Create healthcheck struct with correct information
                Send struct to server
                Get number of errors and entries from server args
                Signal worker threads that server is available
        Else
                Increment variable keeping track of servers not available

If all servers are down
        Send 500 Internal Server Error to client

For each thread
        Construct healthcheck struct
        Create worker thread that calls our dispatch function

Infinite Loop
        Accept client request
        if we have reached R number of requests
                Send health check
        Lock queue
        Enqueue request to queue
        Signal worker threads to requests
        Unlock queue
```

2.2 Dispatch Function

The dispatch function is responsible for giving an equal amount of requests to each server, so that one server will not get overloaded while other servers are not working. We will be using an infinite loop to be able to constantly distribute requests to servers.

1) First we will attempt to connect the threads to an available server by calling our findServerLeastLoad function to determine which server has the least load (requests)
2) After we have found our server, we will lock the list of servers to form a connection.
3) Then after we have formed a connection, we will dequeue a request from our queue structure so that it can be processed. To do this, we will lock the queue, dequeue a request, then unlock it to prevent race conditions.
4) We will finally call bridgeloop() to process the request and send the appropriate responses back and forth between the client and the respective server.

Algorithm 2:

Args - Takes in a struct containing

- List of servers
- Number of servers

Infinite Loop
    Parse
    Lock server list
    Call findServerLeastLoad on list of servers
    Unlock server list
    Call client_connect() on server with least load
    Lock the request queue
    Dequeue client socket
    Unlock request queue
    If client socket > 0
        Call bridgeloop on client socket and server with least load

2.3 Healthcheck Function

For our send healthcheck function it will be working as a concurrent thread that is infinitely always running. Its responsibility is to send a healthcheck function to the server every 5 seconds.

1) Parse struct passed in containing list of servers and number of servers
2) Using struct timeval timeout and seeing if select is equal to 0 OR if we have reached R number of requests, we will then lock the servers and loop through each server and send a healthcheck message to that server, and then recv the response to get the number of entries and errors for that server, and then add these to another array to keep track of these totals.
3) Afterwards we will reset the timer loop infinitely through this process

Algorithm 3:

args - struct containing number of servers and list of servers

Set countdown value to 5 seconds
Infinite Loop:
    If R number of requests of countdown is equal to 0
        Lock list of servers to pause worker threads
        For each server
            Send healthcheck GET request
            Receive message from server into buffer
            Parse buffer to get number of errors and entries
        Unlock list of servers so worker threads can resume

2.4 FindServerLeastLoad Function

The purpose of this function is to determine the server that has the least amount of requests. We will have two lists, list of entries and a list of errors where each index represents a unique server, and the value of the index represents the number of entries or errors that the server has completed. This function will be called by our dispatch when determining which server to distribute our request to.

Algorithm 4:

args - will be taking in list of errors and list of entries

minimumValue = large number
minimumIndex = large number
;

Loop through entry list
    If entrylist value <= minimum value
    minimumValue = entrylistValue
    minimumIndex = currentIndex

Return minimumIndex (index represents unique server)

2.5 Queue of Requests

Our queue is responsible for putting requests in a line by enqueuing and dequeuing them when we receive a process and when we complete a process respectively. Each node in our queue will have a client socket value from which client the request came from, and have a next value that points to the next request in the queue. We have three separate functions for our queue, createNode, enqueueNode, and dequeueNode

1) createNode
    - Contains a key for the socket value
    - Contains pointer to next Node
2) enqueueNode - args - client socket value
    - Create node
    - Set key of node to client socket value
    - Set value of next to be NULL signifying that it is the last value of the list
3) dequeueNode
    - If queue is empty, return NULL
    - Otherwise, set front of queue to next Node