Marshall Morse
1674857
mhmorse

Asgn3 Design Document

Overview

For this project I will be using the same dispatching method as asgn2 to handle multithreading. The main function parses input for R and N and the server ports. It then probes those servers and creates a minimum heap of the servers using a for loop and a addServer and SortHeap functions. Each thread's activation function is handleRequests who's exactly the same as my asgn2 and when a request comes in this function has a thread call handleRequest which is different than asgn2. Handle request gets the right server by calling getNextServer on the requestHeap created and populated in main before initializing threads. then bridges the client socket that made the request with the server using bridgeLoop from the starter code.

The heap manages all of the multithreading concerns that come with having a shared server heap. More on that in the next section. Once a connection is bridged properly the thread will end up back in handleRequests and if a new request is in the requestQueue then it will repeat the process and if not it will go to sleep.

There is also a health check thread that is sent to a health check loop which makes the thread do a 5 second timed wait on its own mutex and conditional variable. Then it unlocks and calls the updateServers function which loops over the heap sending health checks to each and then updates their info. After every server is updated it sorts the heap. After accepting a new connection thread in the main loop if a request count var is greater than the input R or 4 if no argument was provided then we wake up the healthcheck thread.

Important Note: the big O notation used below considers n to be the number of servers!

Server Minimum Heap

This data structure will keep track of the server to be sent requests. It will operate like a min-heap however once populated with server data it will be of a fixed size. It will be initialized and populated in the main. Once populated it will call it's sortHeap function to balance it with the most viable server at the root. I chose a minimum heap because it only costs O(log(n)) to get the server we want to bridge a request to. The cost comes with the heapify call after we get the optimal server with O(1). This is much better than using a O(nlog(n)) sorting algorithm every time an update to a server values loads var, fails flag, or fails var is made. A heap also helps modularize the code. I used atomic operations to change all of the atomic variables mentioned below. The server variables each of getter functions which I won't explain due to their simple nature. The heap uses a variable to keep track of who is sorting and if someone else wants to get the viable server then that server will have to wait until the thread sorting is done.

Member Variables

Struct Server Variables

Atomic_fast32_t loads: keeps track of total requests or loads a server receives.
Atomic_fast32_t failures: keeps track of a servers total failures
const int port_number: immutable and unique server port number

atomic_fast8_t fail_flag: tracks when a server either can't take requests due to timeout or the server crashed. Ie poll(2) call returns -1 for an error.

int port_number: the servers file descriptor.

Member Variables

Struct server servers[server_count]: an array of the servers which makes up the heap

Const Int server_count: keeps track of number of servers in the heap

pthread_mutex server_lock: keeps track of number of servers in the heap

pthread_conditional_var server_lock: keeps track of number of servers in the heap

atomic_int isSorting: variable to keep track of the conditional variable and to put threads asleep when they try to access the heap when it's being sorted or sort it when it's already being sorted.

Member Functions

SortHeap(ServerMinHeap* this, **int** isOneUpdate)

This function will build the heap over again if the isOneUpdate input is not set. This is the case where we call SortHeap after probing each server with healtchecks and updating. If the isOneUpdate input is 1 we are only going to call heapify on the root index of our heaps server list. This decision was made because it's unnecessary to iterate over each server to rebuild the heap after every adjustment to just one when a thread serves a request to a server. After every we get the optimal server and update that optimal server.

There are some multithreading concerns however. At the beginning of this function we are going to set the heaps isSorting flag so that other threads looking to use the heap have to wait until the sorting is over. We are then going to lock it with the heaps lock. We then sort according to input. After this is done we set the isSorting flag to 0 to indicate we are no longer sorting then we wake a thread with pthread_conditional_signal and finally give up the lock.

getNextServer(ServerMinHeap* this)

This function gets the server at index 1 of our min heap whose value is guaranteed to be the server we want. That said there is a lot more complexity that comes along with it. First we lock the server that calls it then we check the isSorting heap flag. And if no one else is sorting we get the optimal server, update its load value, and unlock and then call sort with the isOneUpdate input set to 1 so that it calls Heapify on the root node of our min heap. When sorting it will lock and unlock so that's why we unlock before calling the sort function. After that returns we return the optimal server port number. If the server at the root of the heap is down we return -1 indicating that all servers are down. This function is called in the handle request function in main. If it succeeds without returning -1 then we will make a server connection then bridge the listen socket with server socket.

This function does have some multithreading considerations in that we are to give threads a lock when

Heapify(ServerMinHeap* this)

Heapify function, pretty straightforward. No locking is done within this function. O(log(n)).

addServer(ServerMinHeap* this, loads, fails, fail_flag, index)

Given input it creates a new server in the heap at input index based on other input.

BuildHeap(ServerMinHeap* this)

Typical function to build the heap. No locking is done within this function. O(nlog(n)) however it will be faster than that given that its actually n/2*log(n) and we probably won't use extremely large n. So the constant is probably important to note.

UpdateServer(ServerMinHeap* this, **Atomic_fast32_t** loads, **Atomic_fast32_t** failures, **UINT8** fail_flag, **int** index)

This function will first find the server in the heaps server list at the input index and update it's according values from the input parameters.

Healtcheck Function

This function's job is to send a health check request to a server indicated by an input port number, parse the responses, and fill an input health object with the data from it. This function will be called before the main thread starts accepting requests. After, it will be called every 5 seconds or optionally every R requests whichever comes first. It will be a void function with no return value. It will accept as input a pointer to the server min-heap object and a reference to a health object which is a struct that holds the status flag of the server, loads, and fails. It is also important to note that this function uses select with a 5 second timeout to decide if the server is down or not.

UpdateServers Function

This function's job iterates over the heap and updates each server's value with the results of a call to the health check function. It is called in the health check thread's loop.

Testing

See write up.