Karl Munson
CSE130

Design Doc

Objective :

Create a multithreaded load balancer that forwards client requests to a pool of http servers and decides the correct server to send the request based on data collected from the servers in the pool.

Specification:

Loadbalancer will work with HTTP servers that follow the assignment 2 specification. It will be able to handle multiple concurrent clients. Loadbalancer receives connections from clients and creates connections to the httpservers it has. Loadbalancer can also request healthchecks from servers and does not do any processing of requests from clients. It will process responses to healthchecks that the loadbalancer requests from its servers to determine entry and error counts for the logs of those servers.

The load balancer is called with port numbers for servers it will use and one for itself to receive client requests from. The first port number provided will be the port that the loadbalancer listens on and other provided port numbers will be treated as ports that httpservers are listening on. Loadbalancer can also be provided with option arguments -N <parallel connections> and -R <requests>. -N is the number of parallel connections the loadbalancer can handle and it is default set to four. -R is the number of requests in order to trigger a healthcheck and its default value is set to five. Loadbalancer requests healthchecks from all the servers every -R requests or every X seconds which is specified in the implementation section of this document.

Loadbalancer forwards requests from clients to the servers it has connections to and then returns a response from those servers to the client. The load balancer does not process these requests and responses between clients and only forwards them. The load balancer returns a response to the client after a timeout also specified below. The load balancer continues sending data between client and server until one connection is closed by either the client or the server. This is detected using select().

Loadbalancer performs a healthcheck on startup and then on the previously mentioned conditions. Using the information from the healthcheck the load balancer chooses the server with the fewest entries and uses that server. If it is a tie between entries the server compares based on errors and if that is also a tie it chooses the server closer to the start of the array of servers. If the healthcheck fails the server is marked as dead. This can occur if the response code is not 200, the form of the response is not <errors>\n<entries>, or if the server doesn't respond.

Implementation:

-Data Design:

serverList <structure>:

--struct serverInstance* servers; This is a pointer to the array of serverInstances storing information about each of the servers that load balancer is using.

--int size; This is the length of the array pointed to by servers.

--int request_limit; This is the number of requests before triggering a healthcheck.

--Description: This structure is used to provide information to the thread servers_health which performs the healthcheck.

serverInstance <structure>:

--int errors; This value records the number of requests that resulted in errors. This value is obtained from the healthcheck everytime one is performed.
--int entries; This value records the number of entries in this server's log file. This value is obtained from the healthcheck everytime one is performed.
--int port; This is the port number that this server is located on and is used to connect to the server while forwarding requests from the client of the loadbalancer.
--Description: This structure holds relevant information for each server so it can be decided whether or not to use the server and which port to connect to to use the server.

Worker_info <structure>:

--struct serverInstance* servers; This is a pointer to the array of serverInstances storing information about each of the servers that load balancer is using.
--int size; This is the length of the array pointed to by servers.
--int client_fd; This is the file descriptor for the client of loadbalancer which is sending a request.
--int request_limit; This value contains the number of requests after which a healthcheck should be performed on all of the servers that loadbalancer is using.
--pthread_t worker_id; This is the initializing value for the worker thread.
--pthread_cond_t condition_var; This is the condition variable specific to this worker thread.
--pthread_mutex_t* lock; This is a pointer to a lock that the worker threads share.
--Description: This function provides information to a worker thread so that the thread can bridge the connection between the client and a server.

Queue <structure>:

--int fd; The variable contains the file descriptor for the connection between the client and the loadbalancer.
--struct queue* next; This variable points to the next entry in the queue.
--Description: This structure is a queue for holding all of the client file descriptors until they can be assigned to a worker thread which then fulfills the request.

Dispatcher_info:

--int * port; This is a pointer to an array of ints which contains the port numbers specified in the starting of the load balancer. The first entry is the port the load balancer will be connected to.
--int port_count; This int provides the length of the array at port.
--int requests; This value contains the number of requests after which a healthcheck should be performed on all of the servers that loadbalancer is using.

--int parallel_cons; This value specifies the number of worker threads that loadbalancer will have to handle parallel connections.
--Description: This structure provides the dispatcher thread with all of the information that was parsed and processed in the main function.

request_count <variable>:
--Description: Request count is a global variable that holds the number of requests that have been handled by the load balancer. This value is protected by a mutex while being edited to avoid race conditions related to it.

-Architecture:
Functions:
client_connect:
--Inputs:
-unit16_t connectport; port number to connect a client to
--Outputs:
-int; returns the file descriptor of the connection
--Description: This function connects a client to a socket and returns the file descriptor for the connection. This is used when clients are connecting to the load balancer.
Server_listen:
--Inputs:
Int port: Port number that loadbalancer is sending a request to.
--Outputs:
Int: Returns the file descriptor between the load balancer and an httpserver.
--Description: This function is for connecting the loadbalancer to and httpserver for passing information from a client and for healthchecks.
bridge_connections:
--Inputs:
Int fromfd: This is the file descriptor that is being read from.
Int tofd:This is the file descriptor that is being written to.
--Outputs:
Int: Returns the number of bytes sent and returns 0 on closed connection
--Description: This function passes information between two sockets and is called by bridge loop which decides when to call bridge_connections().
Bridge_loop:
--Inputs:
Int sockfd1: This is the client file descriptor.
Int sockfd2: This is the server file descriptor.
--Outputs:
Void return type
--Description: This function loops and calls bridge-connections() until it detects that one of the descriptors has closed or if bridge connections returns 0 or an error. This is implemented via forever loop with a switch statement on the result of a select function with a timeout. If at any

point bridge_connections() returns an error or 0 there is nothing left to forward so the function returns.
Servers_health:
--Inputs:
Void =* healthcheck_info: This is a pointer to a serverList object which contains information needed for this function to be implemented.
--Outputs:
Stays running until the program closes.
--Description: This function runs a healthcheck on the servers that loadbalancer uses. It has a permanent while loop with a timed wait condition that is entered unless signaled by a worker thread that the number of requests is now divisible by -R mentioned earlier. After waiting the function loops through all of the servers that the load balancer has and sends each one a healthcheck and parses the response to update entry and error values for each server which was passed via the serverList structure. There are checks to check for bad responses, timeout, and poorly formatted but 200 code requests.

Parallel:
--Inputs:
void* server_info: This points to a worker_info object which contains information mentioned above which is needed to implement Parallel.
--Outputs:
Stays running until the program closes.
--Description: This function runs on a loop and waits until it has been signaled by the dispatcher and assigned a client file descriptor. After waiting it goes through an array of serverInstances and looks for the lowest entry count. After this the function bridges the connection using bridge_loop. If a server is unable to be chosen a prebuilt 500 code error is sent to the client. After performing its job the function increments request, resets its client file descriptor to -1, signals the healthcheck thread if requests % -R ==0, and signals the dispatcher thread that a worker is now available.
Dispatcher:
--Inputs:
void* dis_info: this points to a dispatcher_info structure and contains the information parsed from main().
--Outputs:
Stays running until the program closes.
--Description: This function handles assigning client requests to worker threads and initializing threads and structures for the program. Dispatcher first initializes an array of serverInstance structures with info for each server the load balancer has a port number for. After the function initializes a serverList structure with information parsed from main() and the list of serverInstances and starts the server_health thread with this information. After this the Dispatcher initializes worker_info structures with relevant information and starts up threads for each worker the number of which is specified by -N. After starting all of the threads Dispatcher listens on the first port number provided and accepts connections. After receiving a connection

from a client the file descriptor for the client is placed on a queue. The queue is then fed into a function that loops while the queue is not empty, assigning the file descriptors to worker threads running on Parallel(). If all threads are busy dispatcher waits on a condition variable until signaled by a worker thread.

Main:

--Inputs:

Int argc: Contains the number of arguments in argv.

Char ** argv: Contains parameters for loadbalancer which are then parsed by main.

--Outputs:

Int: Main returns 0 on success and 1 on failure.

--Description: Main reads in parameters using getopt and processes the flags and port numbers. It converts the string versions of the port numbers to an array of ints and passes this array, the value for -R, and the value for -N into a structure which is then passed to a thread created from the dispatcher function.

-Design Choices:

This section is encompassed of design choices that are not covered by the functions or data design.

Threads:This program uses N + 2 threads to operate where N is the number of parallel connections which is default four. N of the threads are used to host workers that handle parallel connections, one thread is used to handle the dispatcher which accepts connections from clients and hands them out to the worker threads, and the last thread is used to run the health check thread which needs to run at the same time as the rest of the threads.

Timers: I decided to wait 3 seconds between health checks and for timeouts in other parts of the code. I chose this as it prevents too many health checks from happening and wasting time and resources as every 100ms would do. On the flip side 3 seconds is often enough for the load balancer to be working with relevant data for the servers it is using if there haven't been any recent requests.

Healthchecks on requests: After R requests which is default 5 but can be specified, the server performs a healthcheck on all the servers and updates the entry and error values. This occurs after the Rth request fully completes so that data is first used for request 6.