Design Documentation for `loadbalancer.c`

Jay Montoya | jaanmont | 1742317

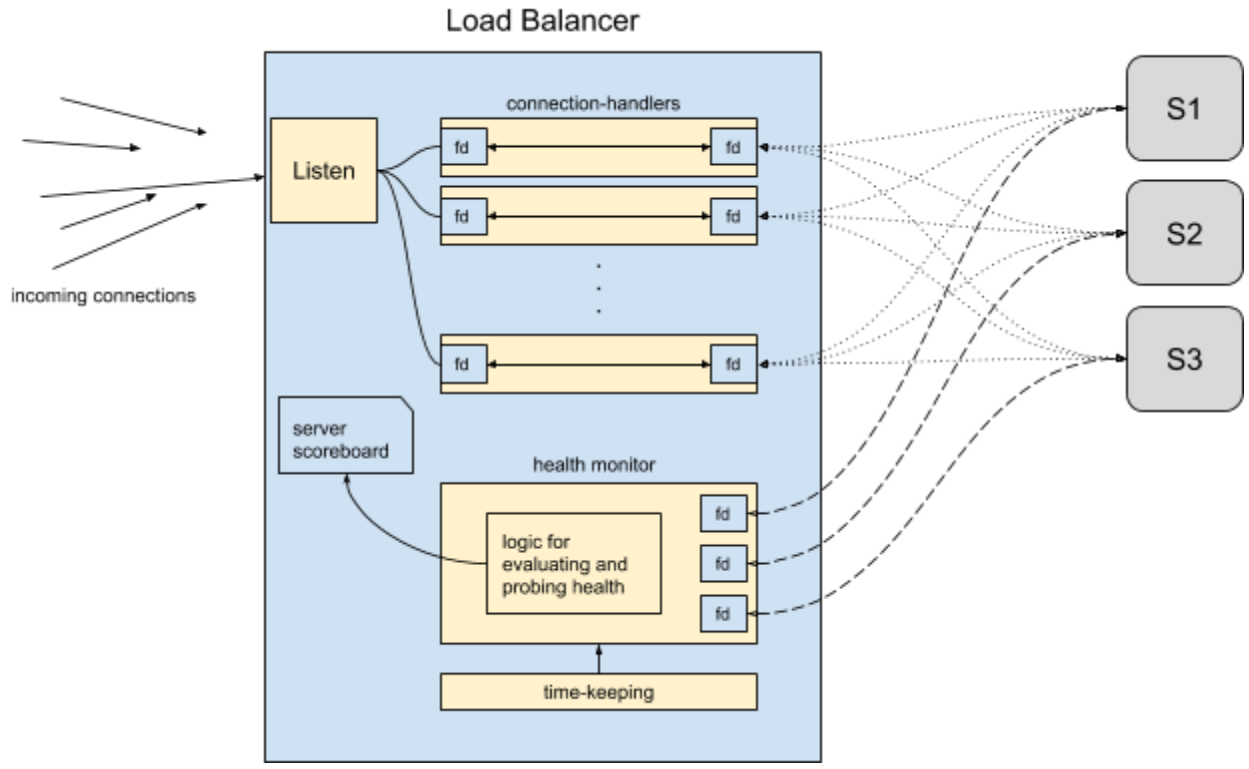
Introduction

This is the design documentation describing loadbalancer.c. The program will distribute connections over a set of servers such as the one implemented in httpserver.c (asgn2). It utilizes concurrency to serve multiple requests at the same time, and uses the health-check mechanism to track the performance of these servers, deciding which one will receive the next connection.

This design is broken into 4 classes of threads:

1. Listening Thread
2. Connection-Handling Thread(s)
3. Health Monitor Thread
4. Time-Keeping Thread

These threads work together to accomplish the load balancing actions, a rough diagram of the design is sketched below.



Listen and Accept Connections (main)

Topics:
Initialization
Connection receiving

Initialization

The responsibility of main() is to initialize the system and receive connections. Below is a list of variables that are defined by the program and used in initializing the system:

```
uint16_t port;
int opt;
size_t num_connections =  DEFAULT_MAX_CONNECTIONS;
size_t requests_between_hc = DEFAULT_REQUESTS_BETWEEN_HC;
pthread_mutex_t connection_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t got_connection = PTHREAD_COND_INITIALIZER;
pthread_mutex_t request_count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t server_health_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t health_check_due = PTHREAD_COND_INITIALIZER;
int num_pending_connections = 0;
int num_slave_servers = 0;
size_t connections_received_since_hc = 0;
bool listener_port_found = 0;
```

Most of these variables are self-explanatory in their logic but a few need more explanation:

connection_mutex: for use with getting and adding a connection to the linked list of connections, used by \
main() and connection-handling threads.
got_connection: same as above

request_count_mutex: for use with updating the connections_received_since_hc variable, preventing contention between connection-handling threads trying to determine if a health check is due.

server_health_mutex: used by the connection-handling threads to obtain the priority server or the health-monitor thread to make changes to the priority server.

health_check_due: used to signal to the health-monitor thread (by the time-keeping or connection-handling threads) that a health check is due.

main() is responsible for interpreting the execution arguments using getopt() and starting the connection-handling threads, the health-monitor thread, and time-keeping thread. Each of these threads receives a subset of the variables above which is defined in the sections below.

Connection receiving

After initialization, main() is responsible for acting as a “dispatcher” of connections to the connection-handling threads. This process is omitted as it is exactly as implemented in the previous assignment: a simple linked list

representing connection descriptors, used in conjunction with a get_connection() and add_connection() function.

Connection Handling

Topics:
Overview, primary goals
Arguments needed for this thread
Determining the forward target
Forwarding the connection

Overview, primary goals

The number of connection-handling threads are specified by the -N optional flag, and is defined as 4 defaulty. The connection-handling thread is responsible for executing the following steps:
WHILE (1):

1. Obtain information about the next connection using get_connection(), this is reused code from the previous assignment and is therefore omitted as it is non-trivial.
2. Obtain the port of the priority server. If no such server exists, send a 500 response.
3. Check if a health check is due by incrementing the variable connections_received_since_hc and seeing if it is equal to the defined thread argument requests_between hc. If they are equal, reset connections_received_since_hc to zero and signal the condition variable health_check due.
4. Obtain a socket to the slave server by using client_connect().
5. Use bridge_loop() to forward the connection.
6. Close both socket descriptors and free the connection struct.

Arguments needed for this thread

The arguments for this thread are defined in the struct below:

```
struct thread_args {
   int thread_id;
   iint* num_pending_connections;              // for use with get_connection()
   int num_slave_servers;
   pthread_mutex_t* connection_mutex;          // for obtaining a connection
   pthread_cond_t* got_connection;             // for obtaining a connection
   pthread_mutex_t* request_count_mutex;       // held when incrementing requests handled by system
   pthread_mutex_t* server_health_mutex;       // for obtaining the priority server pointer
   pthread_cond_t* health_check_due;           // for signalling a due health check to the HC thread
   size_t* connections_received_since_hc;      // for determining when a health check is due
   size_t* requests_between_hc;                // for determining when a health check is due
};
```

Determining the forward target
As explained in earlier sections, the forwarding target is pointed to by the global priority_server pointer to a scoreboard_entry object. All the connection-handler needs to do is consult this pointer, make sure it isn't NULL, and pass that port to client_connect() to proceed with forwarding.

IMPORTANT: Every connection-handler must lock the server_health_mutex to obtain the priority_server pointer. This is to ensure that each connection has the most up-to-date information about the priority server, and does not make that decision when a server probe is being processed. This means that a connection-handler may not forward a request while a health check to all servers is being processed.

It is also important to note that before forwarding a client, the connection-handler must increment the connections_received_since_hc variable to signal if a health-check is due. The code for this is also encapsulated in a critical region to minimize contention between threads. The lock mutex responsible for this section is request_count_mutex.

Forwarding the connection

In order to properly forward a connection, the connection-handler performs the following steps:

1. Call client_connect() to create a socket to the target server.
2. Forward the connection using bridge_loop()
3. Call close() to close the two socket descriptors.

The function client_connect() is the same code as seen in the starter code, except with the following modification:

- If the syscall to connect() returns in error, the connfd variable is closed before returning -1. This was modified after discovering the improper closing of certain file descriptors when slaves servers were problematic.

The function bridghe_loop() was modified a little more heavily, the following changes were made to this function as seen in the starter code:

- Two boolean variables were added: request_sent and response_sent, to determine if a server is unresponsive. Once the client sends something (anything) the request_sent flagged is ticked on. If a timeout with select() occurs, these variables are then observed to determine if a proper “question and answer” took place.

The function bridge_connections() was modified slightly from the starter code:

- If a recv() call to a file descriptor returns 0, the other descriptor is closed before returning.

Health Monitoring and Time-keeping

Topics:
Overview, primary goals
Server scoreboard
Arguments needed for this thread
Algorithm
Function: probe_slaves()
Function: update_problematic()
Function: process_reply()
Function: update_scoreboard_entry()
Function: update_priority_server()
Time-keeping thread

Overview, primary goals

Monitoring the health of the slave servers is vital to determining which server is to receive the next connection. Health monitoring is accomplished by using the *health monitoring thread* to ensure that all servers are probed after every *R* requests or every *X* amount of seconds that pass (whichever comes first). *R* and *X* are defined respectively by the optional parameter **-R** (defaultly 5 requests) and 5 seconds.

The health monitoring thread is started by the connection-listening thread (main) as an initialization of the load balancer. Upon spinning up, the health monitor is responsible for the following:

1. Maintaining a "server scoreboard" to determine which server is the healthiest (implemented as a priority queue).
2. Enforcing mandatory health checks every *R* requests or every *5* seconds by waiting on the conditional variable "health_check_due" which is signalled every 5 seconds or R requests.

Server scoreboard

The "server scoreboard" in this documentation refers to a linked list that will hold information about each server's health in a scoreboard_entry struct. A scoreboard_entry struct is defined as follows:

```
struct scoreboard_entry {
   uint16_t port;
   int16_t total_requests;
   uint16_t total_errors;
   float  request_error_ratio;
   bool problematic;
   struct scoreboard_entry* next;
};
```

There is exactly one scoreboard_entry struct for each server, which can be determined in the initialization of the program through the number of port arguments given for the slave servers.

The responsibility of the health-monitoring thread is to maintain the server-scoreboard as diligently as possible. This involves updating the fields in the subject server's struct appropriately, so that the load balancer can properly indicate which server should have the highest priority.

Arguments needed for this thread

Before speaking on the execution of the health-monitoring thread, we recognize the following variables needed. These variables are to be passed by main() through a health_monitor_args struct, which is defined as follows:

```
struct health_monitor_args {
   int* num_slave_servers;
   pthread_mutex_t* server_health_mutex;
   pthread_cond_t* health_check_due;
   struct scoreboard_entry* priority_server;
};
```

The server_health_mutex must be obtained by the health-monitoring thread whenever conducting a server probe. (It also must be held by the connection-handling threads when selecting a priority server, more in that section). The conditional variable health_check_due is waited on by the health-monitoring thread for a signal

every 5 seconds (coming from the time-keeping thread) or every R requests (as tracked by the connection_handling threads). The priority_server pointer is a pointer to the scoreboard_entry that represents the priority server.

Algorithm

The health-monitoring thread performs the following actions in order:

1. On start-up, lock the server_health_mutex.
2. Call the function probe_slaves(), which takes the number of slave servers and is responsible for sending health checks to the slave servers and modifying their scoreboard entry fields.
3. Call the function update_priority_server() which takes a pointer to the priority server and the number of slave servers. This function executes the logic for selecting the priority server (NULL if none) and setting the given pointer to that scoreboard_entry.

WHILE (1)

4. Release the lock by waiting on the condition variable health_check_due.
5. Repeat steps 2 and 3.

Function: probe_slaves()

void probe_slaves(int num_slave_servers);

The probe_slaves() function is responsible for setting up client connections to the slave servers, issuing a health check to each, and documenting the results. The single parameter, num_slave_servers, represents the true number of slave servers, problematic or not, and must not be 0. After this function returns, each slave server's scoreboard_entry struct has been updated with the most current information about that server.

The following variables are needed by this function:

```
fd_set set, unverified_set;                          // for use with select()
int sockets_to_slaves[num_slave_servers];            // socket to each slave server
struct timeval timeout;                              // for use with select()
struct scoreboard_entry* cursor_entry = scoreboard;  // cursor entry initialized to the leading entry
uint8_t buff[BUFFER_SIZE + 1];                       // buffer for processing server responses
```

This function performs the following steps:

LOOP FOR num_slave_servers:

1. Create a socket to every server by using client_connect() given in the starter code. These client sockets are stored in the sockets_to_slaves array, which is an array that runs parallel to the scoreboard_entries (scoreboard entries are never rearranged in the list). If there was an issue creating the socket, the value is -1 and the server is marked problematic by the function update_problematic() specified in the next sections. If client_connect() returns a successful socket descriptor, then this socket descriptor is added to the fd_set objects 'set' and 'unverified_set'.

LOOP FOR num_slave_servers:

2. Send the health-checks into each socket if it is not -1. This is done easily by constructing a char[] object named probe_message, and using send() to place it on the wire.

WHILE (servers_handled < num_slave_servers):

3. Call select() to monitor the set of socket descriptors ‘set’ for reading, with a timeout of 3 seconds defined by parameters. When select returns, ‘set’ has been modified to only include the descriptors ready for reading.

SWITCH (return value of select):
CASE -1: print an error and return out of the function completely
CASE 0: the timeout of 3 seconds has expired, set the bool expired_timeout to true. ’
DEFAULT:

4. Initialize a cursor scoreboard_entry to the top of the scoreboard.

LOOP FOR num_slave_servers:

5. If the socket is -1, increment servers_handled, move the cursor entry forward, and continue to the next FOR loop cycle. This server could not be reached and has already been marked problematic in step 1.
6. If the socket for the current server is still in ‘set,’ it is ready for reading: Call function process_reply(), and remove the socket descriptor from ‘set’ and ‘unverified_set’. Increment servers_handled.
7. Else. (PERIOD). If expired_timeout is true and the socket in question is in the ‘unverified_set’, mark that server problematic and increment servers_handled. If the timeout has not expired, re-add that socket descriptor to ‘set’ for a retry in the next call to select().
8. Move the cursor entry forward.

Function: update_problematic()

void update_problematic(int port, bool problematic);

The update_problematic() function is a simple function that iterates through the linked list of scoreboard entries, identifies the entry with parameter ‘port’ as it’s port field, and updates the problematic field to the parameter given by ‘problematic’.

The algorithm for this function is omitted as it is non-trivial.

Function: process_reply()

void process_reply(int port, int socket, uint8_t* buff);

The process_reply function is responsible for reading the server reply from ‘socket’, parsing it’s contents, and calling update_scoreboard_entry() to update the entry represented by ‘port’. A buffer is passed as a parameter to be used with reading.

The following variables are needed for this function:

int recv_status;
char* pos;
int num_errors;
int num_requests;
float request_error_ratio;

recv_status is used for validating the success of a call to recv(), the pos pointer is used to point to where the valuable contents of the health-check start. The num_errors and num_requests variables are assigned through parsing, and used to calculate the request_error_ratio.

The function performs the following steps:

1. Call recv() to receive the contents into buff, null-terminate the contents. And close the socket.
2. Create a separate buffer of the char type 'response', which is the exact size of the received contents, and will be used to conduct calls to string functions on.
3. Look for the double CRLF and point 'pos' to this location, if not found using strstr(), indicate an error and return out of the function.
4. Use 'token' and 'rest' to make subsequent calls to strtok_r(), assigning num_errors and num_requests accordingly.

CALCULATE THE REQUEST ERROR RATIO:

5. If num_requests is 0, request_error_ratio is set to 0.0. This avoids a divide by zero error. Else if the number of errors is 0, request_error_ratio is set to -1.0. This is a design decision to represent the undefined result of a divide-by-zero calculation, and is corrected in the logic for update_priority_server(). Else, perform the normal division of num_requests and num_errors (casting to a float value).
6. Call update_scoreboard_entry() to update the results for this probe. Then return.

Function: update_scoreboard_entry()

void update_scoreboard_entry(int port, int total_requests, int total_errors, float request_error_ratio, bool problematic);

The beefier cousin of the update_problematic() function, the update_scoreboard_entry() function updates all verbose fields of the subject scoreboard entry (total_requests, total_errors, request_error_ratio, problematic). These four variables are vital to the success and accuracy of the logic which chooses the priority server in update_priority_server().

Exactly as in update_problematic(), this function iterates through the linked list of scoreboard entries, identified the entry with parameter 'port' as it's port field, and updates the fields given in parameters accordingly.

Once again, this algorithm is non-trivial and therefore omitted.

Function: update_priority_server()

This function is the bread and butter of the load-balancing logic, it utilizes a 2 pass system across the scoreboard entry list to determine the priority server (NULL if none).

The algorithm for this function is included below:

1. Pass through the scoreboard (which is a list of scoreboard_entry structs). Determine a priority server during the pass by only looking at the total_requests and problematic fields of the entry. Keep an integer count of how many servers are qualified.
2. If more than one server is qualified, perform a second pass through the scoreboard, breaking any ties using the request_error_ratio field of the entries.
3. If there is still more than one server qualified, no special attention is given to breaking the tie again, but rather, wherever the cursor last landed is the priority server.

Time-keeping thread:

The time-keeping thread is a minimal thread that does one thing: sleeps 5 seconds and then signals health_check_due to notify the health-monitor thread.