Design Document: Load Balancer (Assignment 3)

Program Description

The load balancer distributes incoming connections to a set of prespecified servers. The load balancer frequently checks the status of the servers and prioritizes the server with the least number of requests performed thus far. The load balancer does not parse through the client's message and instead forwards it to the server and only uses the number of requests to prioritize which server to use.

Objectives

1. Parse command line arguments for client port, server port, threads, and request frequency
2. Implement multithreading to ensure that multiple incoming connections can be handled
3. Implement health checks on servers to select the server with least number of requests and ensure synchronization
4. Bridge connections between client and server

Parsing additional command line arguments

The multi-threaded server accepts optional arguments for numbers of threads and for the request frequency to probe for health checks. If number of threads is not specified it is 4 by default and request frequency will just not be performed if not specified. The user must specify at least one client and server port for the program to run. Multiple server ports can be specified. All arguments must be positive integers and port numbers must range from 0 to 65535

Implementation of Parsing additional command line arguments

1. Command line arguments will be validated in a separate function called argParser() which returns a Boolean if any non-valid argument is specified
2. Using getopt(), optional arguments can be checked. If matching argument is found, it is checked if its format and value are valid. If valid, it will store the value. If not valid, function will return false and program will terminate. This is used for request frequency and number of threads.
3. If no optional arguments are found, the default value is stored
4. After optional arguments are parsed, the remaining Non-option arguments are parsed. These are port numbers. The first argument is stored as client port and the rest as server ports. If any additional arguments are found besides port numbers, function returns false.
5. All arguments are validated using isValidArg() which checks if arguments are integers and within the correct range of values.

Multithreading and synchronization between threads for incoming connections
Multi-threading is used to improve throughput and so the load balancer can handle multiple incoming connections. Each thread will handle a request concurrently. The user specifies the number of worker threads available and each thread must wait until the dispatcher thread provides it with work to do. The threads must not busy-wait and if they need to share resources, they must use synchronization mechanisms.

Required Data structures and mechanisms for multithreading and synchronization for incoming connections

Struct workerInfoObj: Contains a Queue of client file descriptors and the int* bestServer(Contains port of best server to use)

Queue: to ensure that requests are processed in FIFO order and no requests are processed twice or missed

Mutex lock (queue_mutex**)** to ensure synchronization between threads and conditional Locks (cond**)** to prevent busy waiting. Used on the queue to prevent deadlocks and to ensure read/write coherency.

P_thread functions to create threads and a thread function for each thread to process requests

Critical Regions: In main(): when enqueueing client file descriptor. In thread_func() when dequeuing client file descriptors.

Implementation of multithreading and synchronization (incoming connections)

1. Threads are created using p_thread_create() depending on the value specified by user. Threads then wait on a conditional lock (if queue is empty) and mutex lock until there is a request in queue.
2. 2. A dispatcher thread accepts connection from client. queue_mutex is used to lock the critical region and dispatcher thread enqueues the client FD. Once enqueued, it will send a signal using pthread_cond_signal() to wake a worker thread to start processing.
3. 3. The worker threads will start processing the request when it receives a wake signal.
4. 4. if the queue is not empty, worker will not sleep and process queue items. If queue is empty, worker threads will sleep and wait for dispatcher thread to send wake signal.

Health checks on servers, selecting best server and ensuring synchronization

Health checks must be performed on servers to check its status and is performed by a dedicated thread. Health checks are triggered after R number of requests or after X seconds. The X value that was selected was 2 seconds. As the timeout is 5 seconds, X had to be lower than 5 seconds. If X is too low, too many health checks would be performed and if was too long it would not detect if a server went down fast enough. Thus, 2 seconds was selected as it had the best trade-off and leaned towards the safer side. Health checks would be performed somewhat frequently to ensure that the load balancer can detect if a server went down fast enough. After performing health checks on all servers, the best server is selected based on the heuristic that the server with the least number of requests thus far be selected. If the number of requests is the same for multiple servers, the one with the least number of errors is selected from that lot. After the best server is selected, a mutex lock is used to ensure synchronization when updating the best server. If all servers are down, a negative value is updated instead of the best server's port number.

Required Data structures and mechanisms for Health checks on servers, selecting best server and ensuring synchronization

pthread_mutex_t healthMutex and pthread_cond_t healthCond : Used to control frequency of health checks

pthread_mutex_t bestMutex : Used when updating the best server

Stuct servers: store number of servers, request frequency , best server and array of struct server_info

Stuct server_info: stores port number, status, entries, and errors for that specific server

void *health_thread(void*): Health thread function which performs health checks

void updateServerStatus(): send health check to specific server and updates its status

bool validateHealthCheck(): validate health check msg returned from server

Int client_connect(): Connects to a server port and returns a fd, if valid returns -1

void reqSignaller(): Tracks number of requests, if number of requests is the same as the user specified request frequency to probe, it will send a signal to start health check. If health check is initated, number of requests is reset.

Critical Regions: When updating best server and when updating and resetting requests.

Implementation of Health checks on servers, selecting best server and ensuring synchronization

1. dedicated worker thread for health checks is used. Thread enters a loop and sets the server status to true by default.
2. Client_connect is called to connect load balancer to a server. If fails to connect, fd is closed. If successful, updateServerStatus() is called.

3. updateServerStatus() sends a get health check message. If server does not respond in 0.5 seconds, server is marked as unavailable and returns. Else it validates the received health check message in validateHealthCheck()
4. If valid Health check, the number of requests and errors that server has is updated in that server's specific server_info struct.
5. The fd for the server is closed and if the server has the least number of requests it updates a temporary variable to store the best server's index.
6. Once loop is completed, the best server must be updated. Since it is a shared resource, a mutex lock(bestMutex) must be used to ensure synchronization. The best server is then read in thread_func() which likewise used bestMutex to ensure R/W coherency.
7. reqSignaller() is called to reset the request counter and a mutex lock and condition variable(healthMutex, healthCond) is used to make the health thread wait.
8. If 3 seconds passes or if the number of requests is the same as the request frequency specified by the user, a signal is sent to wake the thread to perform a health check.

Bridging connection between client and server

Once the best server is provided, the thread can start bridging the client and server. If all servers are down, a 500 Internal Server Error message should be sent, and load balancer should not attempt to connect to that server. If best server is valid, the worker thread should connect to it and then bridge the connection between the client and server. A timeout should be implemented so that the thread is not blocked and stuck waiting for a message from either side. The sockets should only be closed when there is no longer any data left to be read from either side.

Required Data structures and mechanisms for Bridging connection between client and server

int client_connect(): Used to connect to a server's port number. Returns -1 if unsuccesful
void bridge_loop(): Uses a loop and select() to check if there is data to be read either from client or server's socket. If there is, bridge connections is called.
int bridge_connections(): sends and receives from client to server or vice-versa

Implementation of Bridging connection between client and server

1. If a best server is available, client_connect() is called to connect to the server's port
2. Bridge_loop() is called to check if there is data to be read from either client/server socket. If no data is read by the timeout, sends a 500 error to client. Else, bridge_connections() is called.
3. Bridge_connections sends and receives all data in socket from client to server and vice-versa.
4. After all data is sent, client and server file descriptors are closed

Testing (Individual Components)

Command Line Arguments

1. Test to see if program throw error if no arguments specified
2. Program can read all client and server ports
3. Program can read optional flags
4. Program does not accept invalid arguments (Non-integers and outside of range)

Multiple Incoming Connections

1. Program can accept multiple incoming connections. Tested by sending multiple client requests at the same time
2. Threads should not busy-wait. Tested by inputting print statements.
3. Threads acquire correct client file descriptor. Tested using print statements.

Performs Health Checks and selects Best Server

1. Program validates health checks and times out if no message if received. Tested using print statements and by enabling logging on httpserver. To test for invalid messages, disabled logging on httpserver. And to test timeouts, ran program on port that had to server.
2. Program chooses best available server based on heuristic and sends 500 error if all servers are down. Tested by setting httpserver to a certain number of entries and errors.

Bridges Connections

1. Program send 500 error if all servers are down or if best server is unavailable. Tested by not running any httpservers.
2. Program can properly send and receive all requests. Tested using a script that ran multiple PUT/GET/HEAD requests on various files

Testing (Full Program)

1. Test to see if program can process multiple incoming connections. Using a bash script that sent many requests.
2. Test if program can process all requests correctly.
3. Test if program selects best server based on heuristic
4. Test if program can detect if servers go down either initially or between health checks