# CSE 130 Assignment 3: HTTP Proxy
# Design
Robin Mathison

**Objective:**

The objective of this program is to deliver a working http proxy that acts as a client-server meditator. This program will accept and process client requests as well as send corresponding requests to servers. Thus, the http proxy will assume both client and server roles. The http proxy will also poll servers, perform and process healthchecks, cache responses, and implement multithreading.

**Overview:**

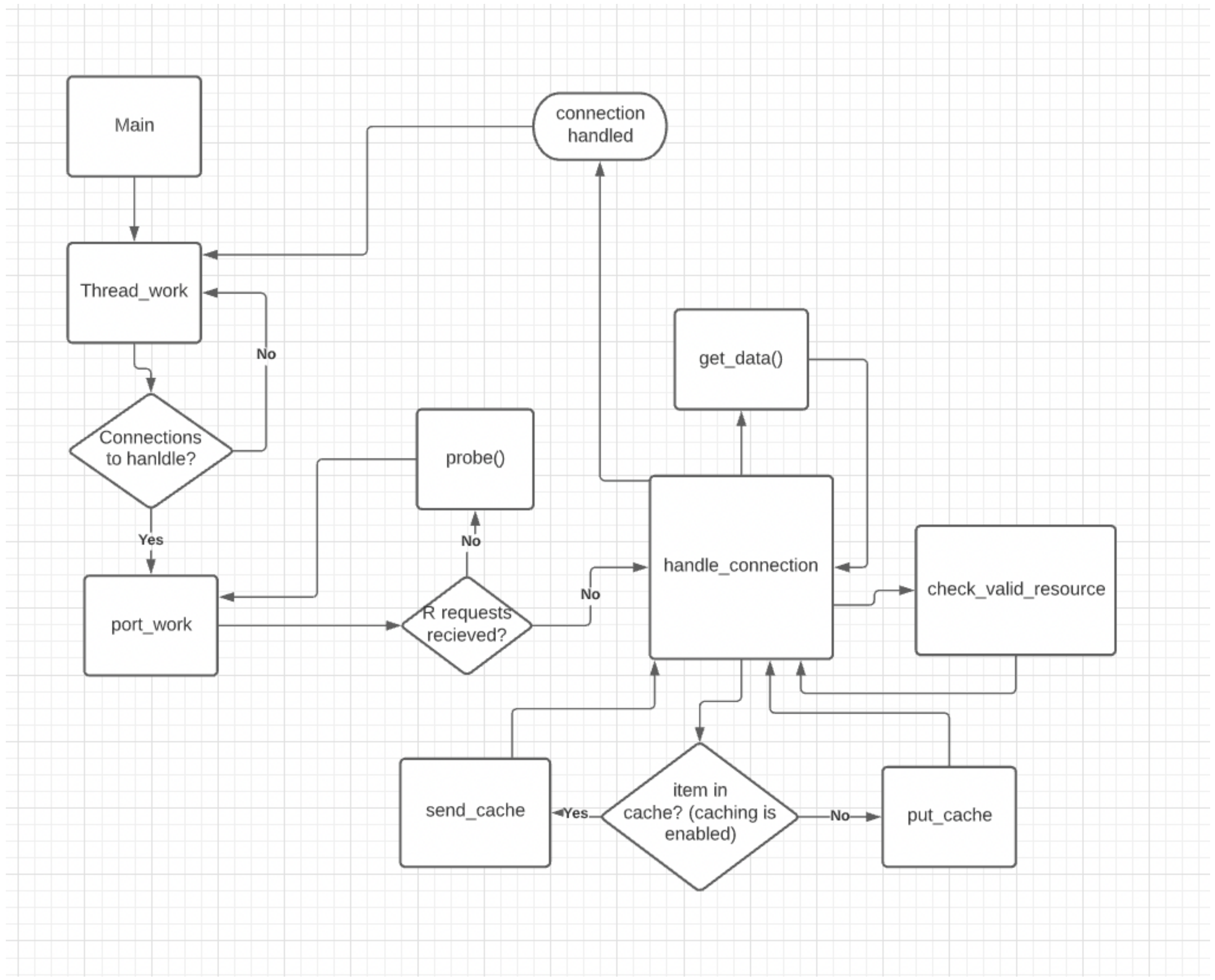

*Figure 0: Program Flow*

**Notable Global Variables:**
uint16_t ports[BUF_SIZE] = the array of server ports
int num_ports = the number of server ports
int poll_freq = the frequency of polling server for their healthcheck status
int num_requests = counter of requests received between polling
int cache_size = size of cache
int file_capacity = max file size that can be stored in cache
int replace = current index of array containing filenames in cache
char** replace_order = array of filenames in the order they were cached
List work_list = FIFO list of connection file descriptors to handle
List ordered_ports = FIFO list of ports to balance load across
Dictionary response_cache = dictionary mapping filename keys to cached content

**Main:**
**Inputs:** int argc, char *argv[]

- Required: 1 proxy port #, servers' port #s (at least 1)
- Optional: -N (num parallel connections, default 5), -R (frequency of health polling, default 5), -s (number of elements in cache/ cache capacity, default 3), -m (maximum size in bytes of cache entry, default 1042)
    - If either -s or -m are 0, no caching performed

**Functionality:**
The main function processes the argv[] program arguments corresponding to the flags listed above. The notable logic used was that once the proxy's port number is processed, all other server port numbers are verified to be different from the proxy's. Similarly, no duplicate port numbers are processed, nor are port values equal to 0. The main function also initializes the global data structures used throughout the program. Afterwards, it probes all of the provided servers with a call to probe_servers(). Lastly, it dispatches N worker threads to thread_work() and signals to them on the condition that there are client connections to handle.

**Probe_Servers:**
**Functionality:**
This function parses a global array containing all valid server port #s processed in main(). For each port #, a healthcheck request is formatted and sent. If a valid response is received, the response is parsed using strtok_r() to extract tokens ending in '\n'. Once the 5th token is extracted, the function calls sscanf() to store the # of errors. The next token is found and sscanf() is called again to store the # of entries. Once this information is found, the # of errors and # of entries in the log of the current port are compared to the previous and a global ordered list is updated such that the server with the least logged requests is in front and the server with the most

requests is in the back. Any server that doesn't respond or sends an error is ignored until the next call to probe_servers().

**<u>Thread_Work:</u>**
**Functionality:**
In this function, threads wait on a conditional mutex until work has been added to work_list. Once there is work, threads wait on another mutex 1 by 1 to access the connfd of the client from work_list. They delete the element accessed, unlock the mutex, then go into port_work().

**<u>Port_Work:</u>**
**Input:** int connfd
**Functionality:**
This function begins with locking a mutex such that threads access the first element from ordered_ports (where the ordering is a result of probe_servers()) 1 by 1. If the list is empty, the port # is set to 0. Otherwise, threads will delete the accessed element and append it to the back of the list such that they balance their load across all available servers. If the port # accessed by the thread is = 0, the thread deletes the element and gets the next one. If there are no other elements to access, then the port # = 0. After unlocking the mutex, threads enter handle_connection().

**<u>Handle_Connection:</u>**
**Inputs:** int connfd, uint16_t port
**Notes:**
- All calls to send() and recv() are validated. If any of them return -1, a 500 Internal Error is sent.
- If the port # = 0, a 500 response is sent.
- All error responses result in the server closing the connection with the client.

**Functionality:**
In this function, a thread will begin by receiving a client's request over connfd. Then, it calls get_data() to parse the request. If the return of this call = -1 then a 400 bad request response is sent, as the request was invalid. Next, the server ensures the request type is GET. If not, it sends the 501 Not Implemented error response.
If caching is enabled, the thread calls send_cached() to determine if the request is cached. If so, the thread sends the response and returns to thread_work. If not, or if caching is not enabled, the thread opens a connection with the server on port. If the connection fails, the server calls mark_bad_port() and sends the 500 Internal Error message, then returns to thread work. Otherwise, it sends the client's request and calls recv() to read the response into a buffer. The response is parsed using strtok_r, looking for tokens ending in '\n'. Depending on the # of tokens found coupled with the # of bytes read and the content length, I determine if there are more bytes to read. In this case, the thread calls recv() until it has received the entirety of the response body.

If caching is enabled, the thread calls put_cache() to store the response into the cache before closing the connection to the server and returning to thread_work().

**<u>Send_Cached:</u>**
**Inputs:** char* filename, int connfd, uint16_t port
**Output:** returns 1 if file in cache, returns 0 if not
**Notes:** Any calls to functions that result in error yield a return value of 0.
**Functionality:**

The thread first enters a critical section protected by a mutex and looks up the filename in question. If it is found, the thread stores the response in allocated memory. Otherwise, 0 is returned. The thread leaves the critical region then sends a HEAD request for filename over port. The thread then parses both the cached header and HEAD response header using strtok_r and extracting tokens ending in '\n'. Once it sscanf()s the Last-Modified: lines, the thread uses strptime() and timegm() to compare the times. If the cached time is greater than or equal to the head response time, the thread sends the response from the cache over connfd and returns 1. Otherwise, 0 is returned.

**<u>Put_Cache:</u>**
**Inputs:** char* cache_string, char* filename, int r_size
**Notes:** If filename already exists in the cache, it is ignored and this function does nothing.
**Functionality:**

This entire function is inside the same mutex that protects all other caching operations. The function inserts the filename key into a dictionary with values cache_string and r_size, where the former is the entire response (head + body) and size_r is the number of bytes of cache_string. The function also keeps track of a global counter that represents the index of a global array. The array stores the filenames in the order that they were added into the cache. If the dictionary's size is equal to the cache size indicated by the initial program argument, the filename at the array's current index is deleted from the dictionary before being replaced by the new filename. Otherwise, the current filename is placed into the array at the current index. Filenames are placed into the array in the order they were cached, so the global counter increments every iteration until it reaches the size of the cache - 1, then the index is set back to 0.

**<u>Mark_Bad_Port:</u>**
**Input:** port
**Note:** The operation of this function works to help load balance with port_work() and probe_servers().
**Functionality:**

This entire function is inside the same mutex that protects all other port operations. It parses the list of ports until it finds the index where the port input is stored. It replaces port with 0.

**<u>Other Functions:</u>**
The functions check_valid_resource() and get_data() maintain their same functionality from assignments 1 & 2.

**<u>Design Points:</u>**

- All global variables and data structures are protected by their respective port, connection, or cache mutex. This ensures thread safety of these critical regions.
- All parts of the program where global variables are not being updated are concurrent with one another.
- Load balancing is achieved via the logic in probe_servers(), where the ordered list of ports is initialized. This balance is maintained by handling connections over the current port of the list. mark_bad_port() ensures that bad ports are replaced with 0, signaling to the proxy to try a different port.
- The cache is implemented as a binary search tree/dictionary to aid performance for quicker lookups. Its adjacent array-- replace_order-- is responsible for maintaining the order that items were cached.
- The List type is a doubly linked list that is used as a FIFO queue for ports and connection fds.
- The Dictionary type is a red/black binary search tree.