Multi-Threaded Web Server with IPC and Semaphores - Technical Report

by Luís Correia 125264; Guilherme Martins 125260
Operating Systems | P3

Introduction

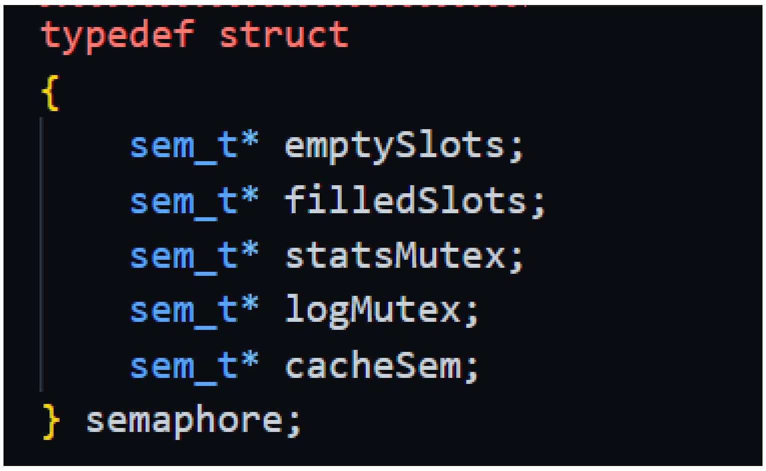
The following document aims to provide a proper technical report that covers details regarding the implementation of several aspects of the project, challenges in its development and the solutions that allowed us to surpass them, as well as a thorough performance analysis of the web server itself.

Implementation Details

Inter-process synchronization

Whenever one finds themselves building any sort of multithreaded project, implementing synchronization is always crucial, in order to avoid undesirable situations such as the emergence of race conditions or deadlocks.

For this project, in order to make queue-handling operations thread-safe, a custom *semaphore* struct, *semaphore*, was created.

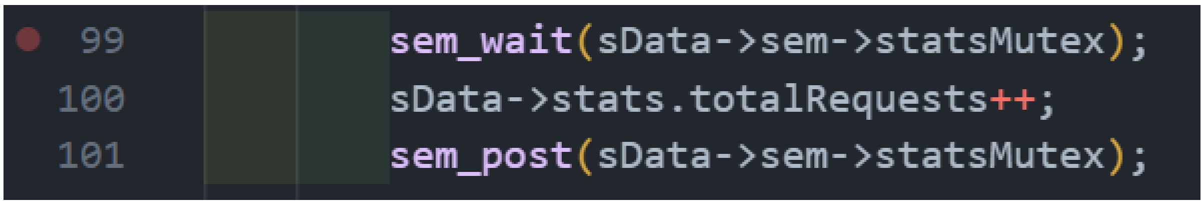


```
typedef struct
{
    sem_t* emptySlots;
    sem_t* filledSlots;
    sem_t* statsMutex;
    sem_t* logMutex;
    sem_t* cacheSem;
} semaphore;
```

Fig. 1: the semaphore struct.

It contains pointers to 5 different semaphores, essentially being a wrapper to make handling them in code easier: *emptySlots*, which counts how many free positions are available in a queue of a certain size; *filledSlots*, which counts how many positions in the queue are occupied; *queueMutex*, which makes it such that no two threads can access the queue simultaneously; *statsMutex*, which protects updates to shared statistics; and *logMutex*, which ensures that logging to a shared file/stream is thread-safe.

In the *main.c* file, a *semaphore* struct, *sem*, is created, which, when the main process is forked and creates the worker processes, is passed as an argument to the *CreateThreadPool()* function in order to create a new *threadPool* struct that uses the semaphores in *sem* to coordinate its threads.

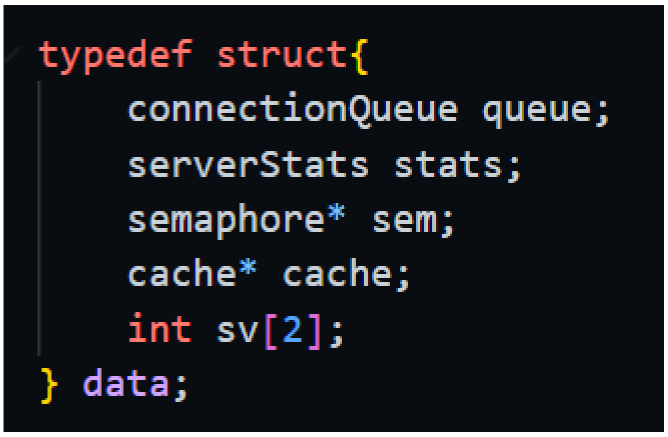


```
            sem_wait(sData->sem->statsMutex);
            sData->stats.totalRequests++;
            sem_post(sData->sem->statsMutex);
```

Fig. 2: in the main logic loop in worker.c, the statsMutex semaphore from the sem struct is locked and unlocked to prevent the emergence of race conditions when updating statistics (in this case, totalRequests).

Master-worker communication

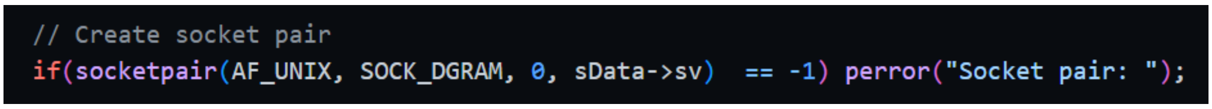
In order to allow for a proper establishment of hierarchy and coordination between the master process and the worker processes, proper communication had to be set up such that the worker processes could properly use their thread pool to perform work related to the connections established to the server.
The first major challenge encountered in this project was, precisely, passing the client file descriptors between the master process and a worker process. Said challenge was overcome via the use of socket pairs.



```c
typedef struct{
    connectionQueue queue;
    serverStats stats;
    semaphore* sem;
    cache* cache;
    int sv[2];
} data;
```

Fig. 3: the data struct. The member int sv[2] is the two-integer-long array that will become the socket pair.

Each *data* struct - which is used for handling shared data - contains an array of two integers, *sv*. Upon the creation of one of these structs with the function *createSharedData()*, the function *socketpair()* is called on *sv*, taking four parameters: the domain (*AF_UNIX*, since the two processes that are going to communicate are in the same machine); the socket type (*SOCK_DGRAM*, to establish a UDP connection that terminates after a single package is sent); the protocol (set to 0, as is the default for *AF_UNIX* sockets); and, finally the two-integer-long array - *sv*. If the function does not encounter any errors, a socket pair is established, allowing for a bidirectional "pipe-like" connection between two processes.

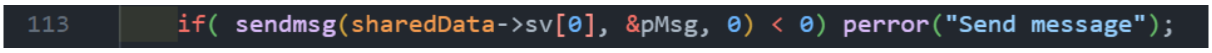


```c
// Create socket pair
if(socketpair(AF_UNIX, SOCK_DGRAM, 0, sData->sv)  == -1) perror("Socket pair: ");
```

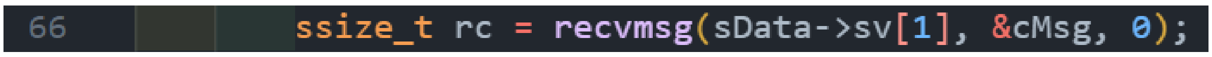
Fig. 4: the creation of the socket pair, in the createSharedData() function.

When the master process accepts a connection via the *acceptConnection()* function, it will eventually attempt to send a message with *sendmsg(sharedData->sv[0], &pMsg, 0)*, where *sv[0]* is the socket of the socket pair that only the master process interacts with (and &pMsg being the memory address of a *msghdr* struct that contains the message's buffers and destination address).

A worker process will then receive said message with *recvmsg(sData->sv[1], &cMsg, 0)*, where *sv[1]* is the socket of the socket pair that the worker process interacts with. The message here contains the client file descriptor, which will then be used by the worker's thread pool's threads in order to parse the client's HTTP request.



```
    if( sendmsg(sharedData->sv[0], &pMsg, 0) < 0) perror("Send message");
```

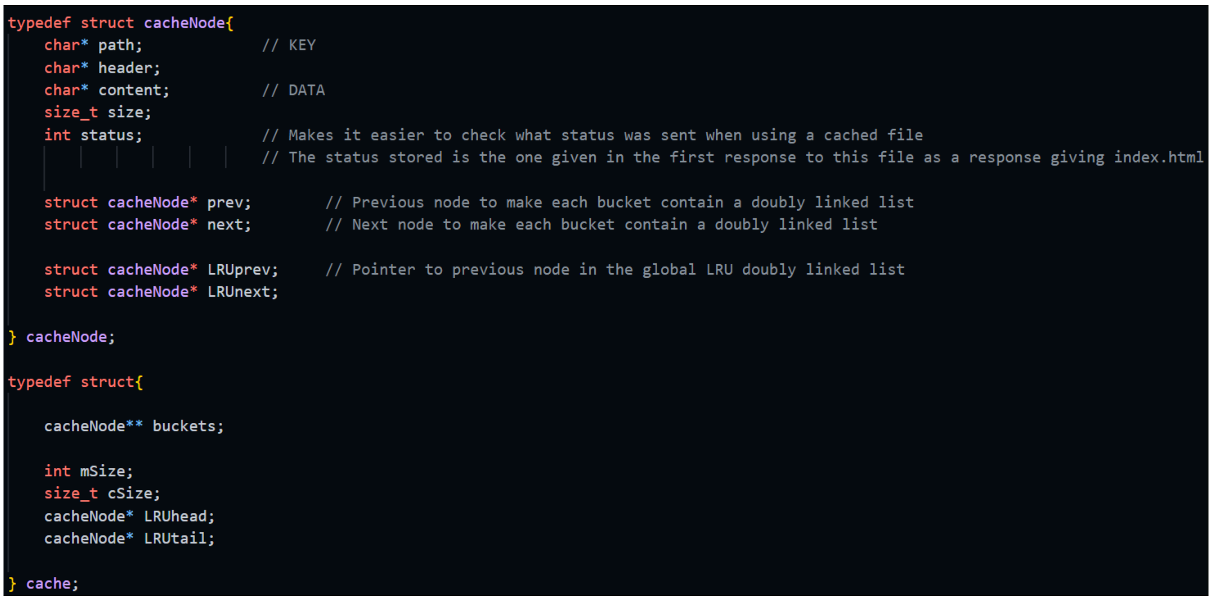


```
        ssize_t rc = recvmsg(sData->sv[1], &cMsg, 0);
```

Figs. 5 and 6: Message transmission with the help of the socket pair. The first image shows sendmsg() in master.c, which sends the message across the socket pair, with the second one showing where the message is received in worker.c, with recvmsg().

Server caching mechanism

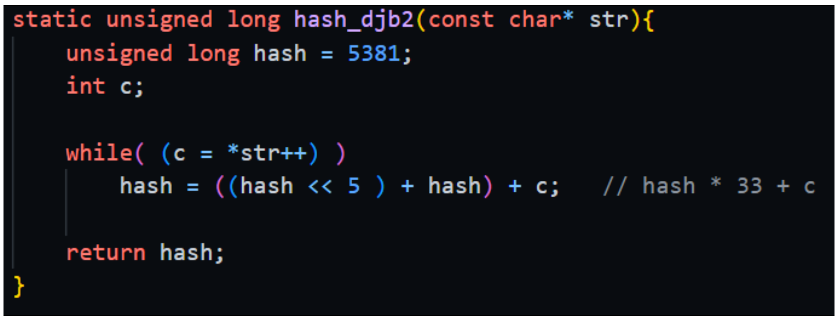
The server employs a caching mechanism designed to optimize resource delivery and manage memory effectively in its multi-threaded environment. The cache is built upon a combination of a hash table with chaining for rapid key-based access and an LRU (Least Recently Used) doubly linked list for efficient cache eviction. This ensures both fast lookup performance and a practical memory management policy.



```c
typedef struct cacheNode{
    char* path;             // KEY
    char* header;
    char* content;          // DATA
    size_t size;
    int status;             // Makes it easier to check what status was sent when using a cached file
                            // The status stored is the one given in the first response to this file as a response giving index.html

    struct cacheNode* prev;        // Previous node to make each bucket contain a doubly linked list
    struct cacheNode* next;        // Next node to make each bucket contain a doubly linked list

    struct cacheNode* LRUprev;     // Pointer to previous node in the global LRU doubly linked list
    struct cacheNode* LRUnext;

} cacheNode;

typedef struct{

    cacheNode** buckets;

    int mSize;
    size_t cSize;
    cacheNode* LRUhead;
    cacheNode* LRUtail;

} cache;
```

Fig. 7: the cacheNode and cache structs.

The core data struct is *cacheNode*, which stores the request path (used as the hash key), the complete HTTP header, the file content and its size. The path is hashed using the well-known *djb2* algorithm to determine the appropriate bucket within the hash table. Any collisions are resolved through separate chaining.

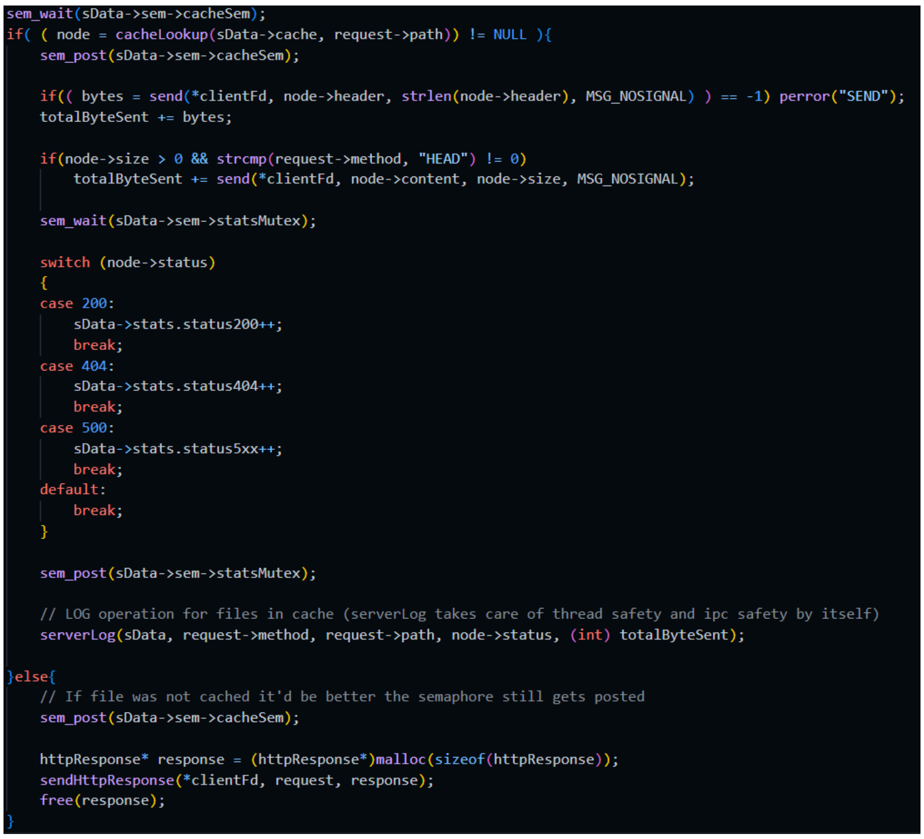


```c
static unsigned long hash_djb2(const char* str){
    unsigned long hash = 5381;
    int c;

    while( (c = *str++) )
        hash = ((hash << 5 ) + hash) + c;   // hash * 33 + c

    return hash;
}
```

Fig. 8: the implementation of the djb2 algorithm.

For memory control, the cache implements an LRU policy. All cached items are linked in a separate doubly linked list, and the most recently accessed one is moved to the list's head while the least recently accessed item is at the tail.
During insertion (*cacheInsert*), a size check is performed against the configured maximum capacity (determined by the amount defined in the *server.conf* file). If the insertion would exceed this limit, the server starts removing the item pointed to by LRUtail until sufficient space is available.

Given that the cache resides in shared memory and is accessed by multiple worker threads, all critical cache operations - cacheLookup, cacheInsert, and cacheRemove - are protected by a dedicated semaphore inside the *sData* struct, *sData->sem->cacheSem*. When a request is processed by a worker thread, the thread first acquires the *cacheSem* lock; then, it calls *cacheLookup*. A cache hit triggers an LRU update, moving the node to *LRUhead* before releasing the lock and serving the content. A cache miss leads to the lock being released so that other threads are not blocked while the file is fetched from the disk.
Upon successfully retrieving a file from the disk and preparing the response, the *sendHttpResponse* function attempts to insert the new item into the cache, re-acquiring the *cacheSem* lock for the atomic insertion process. This strict use of the semaphore prevents race conditions, ensuring data integrity during concurrent modification of the hash table and the LRU list pointers.

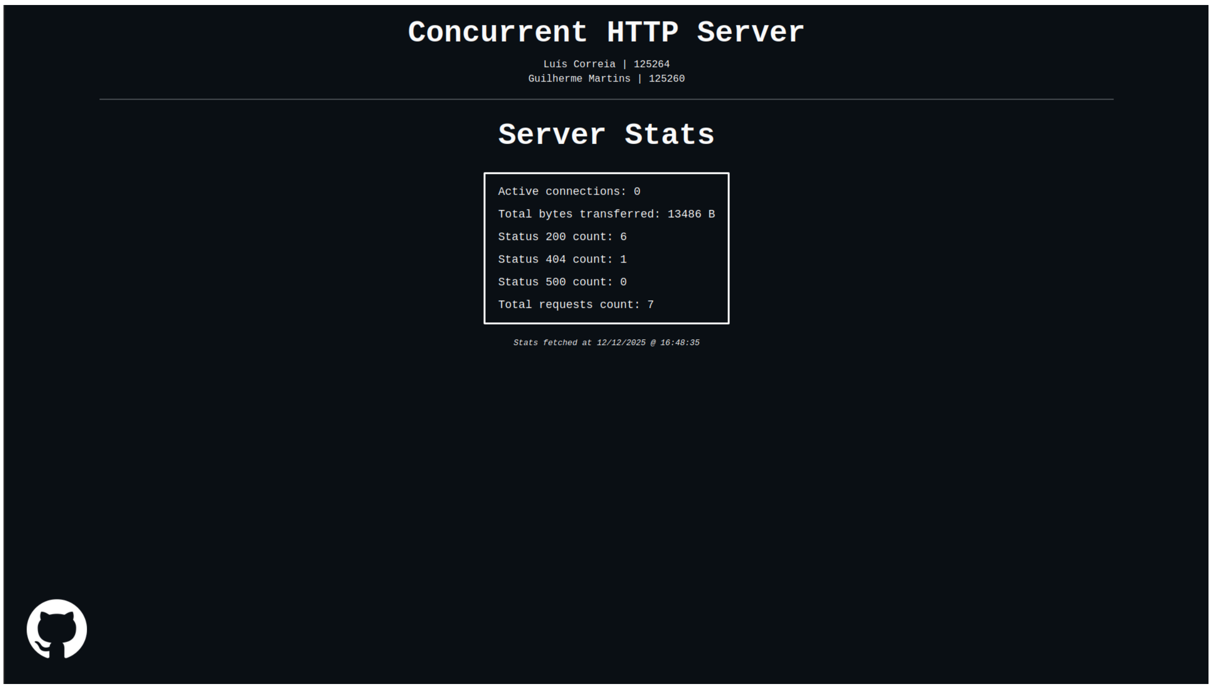


```c
sem_wait(sData->sem->cacheSem);
if( ( node = cacheLookup(sData->cache, request->path)) != NULL ){
    sem_post(sData->sem->cacheSem);

    if(( bytes = send(*clientFd, node->header, strlen(node->header), MSG_NOSIGNAL) ) == -1) perror("SEND");
    totalByteSent += bytes;

    if(node->size > 0 && strcmp(request->method, "HEAD") != 0)
        totalByteSent += send(*clientFd, node->content, node->size, MSG_NOSIGNAL);

    sem_wait(sData->sem->statsMutex);

    switch (node->status)
    {
    case 200:
        sData->stats.status200++;
        break;
    case 404:
        sData->stats.status404++;
        break;
    case 500:
        sData->stats.status5xx++;
        break;
    default:
        break;
    }

    sem_post(sData->sem->statsMutex);

    // LOG operation for files in cache (serverLog takes care of thread safety and ipc safety by itself)
    serverLog(sData, request->method, request->path, node->status, (int) totalByteSent);

}else{
    // If file was not cached it'd be better the semaphore still gets posted
    sem_post(sData->sem->cacheSem);

    httpResponse* response = (httpResponse*)malloc(sizeof(httpResponse));
    sendHttpResponse(*clientFd, request, response);
    free(response);
}
```

Fig. 9: the caching mechanism in worker.c.

Statistics-displaying HTML page

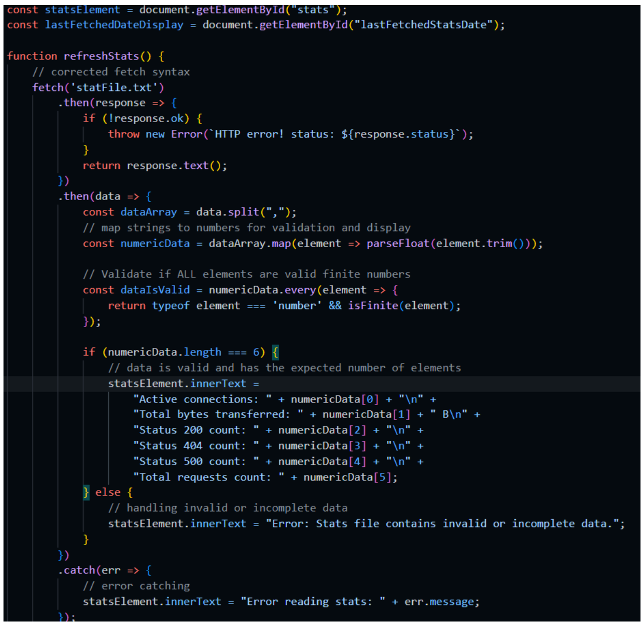
One feature implemented to the server was an HTML page that displays several statistics related to the server and to the connections established to it.



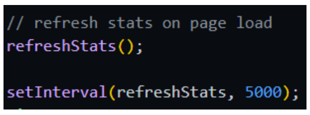
```
Concurrent HTTP Server
Luis Correia | 125264
Guilherme Martins | 125260

Server Stats

Active connections: 0
Total bytes transferred: 13486 B
Status 200 count: 6
Status 404 count: 1
Status 500 count: 0
Total requests count: 7

Stats fetched at 12/12/2025 @ 16:48:35
```

Fig. 10: the stats-displaying page. Note: the image on the bottom left corner redirects to the Github page for the project's repository when clicked.

First, these statistics need to be saved, and that is done by the *updateStatFile* function, which simply rewrites the contents of the *statFile.txt* text file, in the *www* directory, with updated statistics every time it is called. *updateStatFile* is then called on a loop every 5 seconds on the *showStats()* function. This function starts running on a thread created by the master process upon the server's initialization.



```javascript
const statsElement = document.getElementById("stats");
const lastFetchedDateDisplay = document.getElementById("lastFetchedStatsDate");

function refreshStats() {
    // corrected fetch syntax
    fetch('statFile.txt')
        .then(response => {
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }
            return response.text();
        })
        .then(data => {
            const dataArray = data.split(",");
            // map strings to numbers for validation and display
            const numericData = dataArray.map(element => parseFloat(element.trim()));

            // Validate if ALL elements are valid finite numbers
            const dataIsValid = numericData.every(element => {
                return typeof element === 'number' && isFinite(element);
            });

            if (numericData.length === 6) {
                // data is valid and has the expected number of elements
                statsElement.innerText =
                    "Active connections: " + numericData[0] + "\n" +
                    "Total bytes transferred: " + numericData[1] + " B\n" +
                    "Status 200 count: " + numericData[2] + "\n" +
                    "Status 404 count: " + numericData[3] + "\n" +
                    "Status 500 count: " + numericData[4] + "\n" +
                    "Total requests count: " + numericData[5];
            } else {
                // handling invalid or incomplete data
                statsElement.innerText = "Error: Stats file contains invalid or incomplete data.";
            }
        })
        .catch(err => {
            // error catching
            statsElement.innerText = "Error reading stats: " + err.message;
        });
```



```javascript
// refresh stats on page load
refreshStats();

setInterval(refreshStats, 5000);
```

Fig. 11: Javascript code that fetches the data from the .txt file and displays it (the elements with IDs "stats" and "lastFetchedStatsDate" are both <p> elements).
Fig. 12: stat refreshing logic.

In the *statsPage.html* file, in a <*script*> tag, a Javascript function was implemented to fetch the *statFile.txt* file and parse its content, and then update the two <p> elements on the page: one which displays the actual stats, and one which displays the time and date at which those stats were fetched from the .txt file.
This function is then called every time the page is refreshed and every 5 seconds to update the displayed stats (the same time interval between *updateStatFile()* calls and, subsequently, updates to the *statFile.txt* file).

Performance analysis

ApacheBench benchmarking

The following information was obtained by benchmarking the server with *ApacheBench*. For each case, 100000 HTTP GET requests were made to the server, but with different concurrency settings.

Concurrency	Total time taken (s)	Requests per second (mean)	Time per request (ms) (mean)	Min. connection time (ms)	Mean con. time (+/- s.d.) (ms)	Median connection time (ms)	Max connection time (ms)
10	17.332	5769.73	1.733	<0	<2 (+/- 1.3)	1	20
100	12.981	7703.43	12.981	2	13 (+/- 6.5)	12	104
250	13.438	7441.86	33.594	3	33 (+/- 142.1)	13	2267
450	13.944	7171.63	62.747	3	61 (+/- 241.9)	14	4377

Fig. 13: Data obtained by benchmarking with ApacheBench (100000 requests). All tests were performed on the same virtual machine, on the same computer.
.

The best overall throughput, measured as Requests per second (mean), is achieved at a concurrency of 100, yielding 7703.43 requests/s. This peak performance suggests that 100 concurrent connections optimally utilize the server's resources (processes, threads, and I/O capacity) without introducing significant overhead from context switching or lock contention. As concurrency further increases to 250 and 450, the throughput begins to decline, dropping to 7441.86 requests/s and 7171.63 requests/s, respectively. This decrease indicates the onset of resource saturation and inefficiency, where the overhead of managing a larger number of simultaneous tasks outweighs the benefits of parallel execution.This is strongly correlated with the Time per request (mean), which increases sharply from 1.733 ms at concurrency 10 to 62.747 ms at concurrency 450. A higher time per request directly reflects the longer waiting time experienced by individual clients due to increased queueing and competition for server resources.

Known Issues

With the project being now done there are some issues that have not been resolved in time or were simply too persistent and a solution was not found.

- There is a problem with counting the amount of total requests. This causes another problem with counting the number of response status.
- Keep Alive connections are not supported.
- There WAS a bug where the server would stop answering requests and would say nothing in the terminal besides a “connection reset by peer” in apache bench. We do not know what caused this problem as neither valgrind nor hellgrind showed anything relevant to it. This happened when running ab with 500 or more concurrent requests (-c 500). BUT as of now when trying the same thing it works. It is possible the issue returns as we don’t know what caused it to be solved

Difficulties and problems encountered

Whilst working on this project, we, as was to be expected, experienced plenty of difficulties, although some of them were worse than others.
To start, we would like to address the passing of file descriptors between processes. We already knew, even back when we started the project, that it would not be a trivial matter, however, we did not expect that it would lead to the emergence of so many more problems. With a lot of research and brainstorming we landed on the idea of using the *recvmsg* and *sendmsg* functions, although later on we were made aware of other functions - such as the *pidfd_getfd()*, for example - that could have made the solution a tad simpler.
Due to the nature of the solution we chose to implement, we ended up changing a part of the project structure in some ways: we forsook the implementation of the connection queue, as the *recvmsg* and *sendmsg* functions made it obsolete; changed how we synchronized the different threads and processes as the aforementioned functions are thread-and-process-safe, making the threads wait until there is a message to be received or space for it to be sent.
Another section of the project that proved to be rather difficult was the implementation of the LRU cache, as it used more complex data structures and had to be placed in the shared memory. At first, we thought the cache had to be implemented as a single cache object for all processes, but, we quickly realized that such an implementation would force us to refactor a significant part of the *worker.c* and *http.c* files, as well as all of the server cache code, and that lead us to decide to follow with the implementation of a per-process cache.
These are what we consider to have been the hardest parts to implement in the whole project; however, pretty much every single aspect of the project proved to be far from trivial and full of challenges, which forced us to conduct extensive and rigorous research on several aspects, and give our best to be able to finish the project in the short timespan established for it.

Conclusion

The project successfully implemented a high-performance concurrent server with the stability of the producer-consumer pattern, with its technical details and challenges and solutions in feature implementations being thoroughly reported on this document.
Performance analysis via ApacheBench revealed optimal throughput at a concurrency of 100 requests/s (7703.43 req/s), with stability degrading at higher loads.
Overall, the server successfully validates the design principles of robust IPC, strong synchronization, and effective resource management.

References

- https://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html
- https://dev.to/jeffreythecoder/how-i-built-a-simple-http-server-from-scratch-using-c-739

- https://pubs.opengroup.org/onlinepubs/009695399/basedefs/netinet/in.h.html
- https://pubs.opengroup.org/onlinepubs/007904875/basedefs/sys/mman.h.html
- https://pubs.opengroup.org/onlinepubs/7908799/xsh/semaphore.h.html
- https://stackoverflow.com/questions/29331938/what-things-are-exactly-happening-when-server-socket-accept-client-sockets
- https://stackoverflow.com/questions/11461106/socketpair-in-c-unix
- https://www.ibm.com/docs/en/ztpf/1.1.2025?topic=apis-socketpair-create-pair-connected-sockets
- https://www.ibm.com/docs/en/i/7.6.0?topic=processes-example-worker-program-used-sendmsg-recvmsg
- https://www.geeksforgeeks.org/c/memset-c-example/
- https://stackoverflow.com/questions/8481138/how-to-use-sendmsg-to-send-a-file-descriptor-via-sockets-between-2-processes
- https://www.quora.com/How-can-you-pass-file-descriptors-between-processes-in-Unix
- https://stackoverflow.com/questions/1788095/descriptor-passing-with-unix-domain-sockets
- https://stackoverflow.com/questions/14721005/connect-socket-operation-on-non-socket
- https://stackoverflow.com/questions/2358684/can-i-share-a-file-descriptor-to-another-process-on-linux-or-are-they-local-to-t
- https://www.ibm.com/docs/en/zos/3.2.0?topic=calls-sendmsg
- https://linux.die.net/man/7/unix
- https://linux.die.net/man/3/cmsg
- https://www.ibm.com/docs/en/zos/2.5.0?topic=functions-sendmsg-send-messages-socket
- https://stackoverflow.com/questions/8359322/how-to-share-semaphores-between-processes-using-shared-memory
- https://mimesniff.spec.whatwg.org/#javascript-mime-type
- https://www.reddit.com/r/learnjavascript/comments/r4ss8t/how_to_parse_data_from_txt_file_using_javascript/
- https://pubs.opengroup.org/onlinepubs/007904975/functions/fread.html
- https://linux.die.net/man/2/send
- https://man7.org/linux/man-pages/man3/getopt.3.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/getopt.html
- https://serverfault.com/questions/146605/understanding-this-error-apr-socket-recv-connection-reset-by-peer-104