CSE 5344 - 001

Project 1

9/27/2022

Lucas Streanga

1001612558

Design Overview

The transition from Java to C++ requires a few design changes. Firstly, the standard library does not offer high-level classes for networking. As such, two classes are created, Socket and ServerSocket. These function similarly to their java counterparts. Both are declared in **include/socket.h** and defined in **src/socket.cpp**. A simple exception class, SocketException, is included as well.

```cpp
class Socket
{
    friend class ServerSocket;
    //Read 512 bytes at a time
    static const std::size_t buffer_size = 512;
    //Socket Descriptor. Important!
    int socketd;
    //We need to hang onto these too
    sockaddr_in client_address;
    socklen_t client_address_len;


public:

    Socket(): socketd{0}, client_address{0}, client_address_len{0} {}
    Socket(const Socket &) = default;
    void close();
    std::string read() noexcept(false);

    //This one writes raw bytes. Note the use of shared pointer and const,
    //Indicating this function does not own or modify this memory
    void write(const std::shared_ptr<std::byte []> & buffer, std::size_t size) noexcept(false);
    //This one writes a string. String is not modified (const)
    void write(const std::string &) noexcept(false);

    //returns the address as human-readable
    std::string get_address();

    //Destructor does not close socket.
    ~Socket() = default;

};
```

```cpp
class ServerSocket
{
    //socket descriptor. We need to hang on to this
    int socketd;
    //Socket address. castable to sockaddr for bind()
    sockaddr_in sock_address;
    //needed for setsockopp
    int opt;
    //file descripor set needed for select
    fd_set read_fds;
    fd_set write_fds;
    //timeout
    struct timeval timeout;

public:
    //This can throw if the socket can't be binded
    ServerSocket(unsigned short port) noexcept(false);
    //Returns a client socket on accepting connection
    //This can throw
    Socket accept() noexcept(false);
    //This function will wait for a connection for 5 seconds
    //Why do we have this? So accept does not block indefinitely
    //Why is accept blocking indefinitely bad? Thread safety with aborting main with ctrl-C
    //Returns true if a connection happens, false if not...
    bool await(unsigned int seconds = 5);

    //we can close the socket here (very convenient)
    ~ServerSocket();

};
```

These classes act as wrappers for the underlying C api for sockets. The goal is to decouple our program from lower-level functions, giving us easy to use objects with modern expectations, such as well-defined lifetimes, RAII, safe memory access, exceptions, etc. Methods which may throw have the noexcept(false) tag, and include Socket.read(), Socket.write(), ServerSocket constructor, and ServerSocket.accept().

ServerSocket is constructed with a port number. It then binds and listens on the port. It constructs a socket using sys/socket.h. ServerSocket holds onto **socketd**, the file descriptor for the socket, **opt**, the int describing the options needed for lower-level functions, and the socket address as **sock_address**. This function can throw if binding fails.

ServerSocket.accept() returns a Socket. It's important to note this function will not block, as we use the function **ServerSocket.await()** to wait for a connection with a timeout. ServerSocket.accept() essentially runs the library function accept, and may throw an exception if accept fails.

Socket has a few functions. **get_address()** returns the address in human readable format (i.e. 127.0.0.1 for localhost). **close()** closes the socket. **read()** will read data from the socket and return a string. It runs the library function read() with a buffered reader approach. This will throw if reading fails. **write()** has two overloads, one for writing a string to the socket and the other for writing raw bites. Both can throw.

Requests are handled in request.cpp. Functions are used here, not classes. **process_request(Socket client_socket)** will process a request. This works similarly to the Java code. One difference is that SocketExceptions are handled in process_request(). This is because our

threads are running in detached mode, and therefore the main thread cannot catch exceptions in process_request().

The main while loop is simple.

```
while(true)
{
    if(terminated)
        break;

    Socket client_socket;
    if(server_socket->await())
    {
        try { client_socket = server_socket->accept(); }
        catch (SocketException & e)
        {
            std::cerr << e.what() << std::endl;
        }

        //why no catching exceptions? Since the thread is detached, exceptions must be caught there.
        //Notice a new thread is created and will run the process_request function
        std::thread(process_request, client_socket).detach();
    }
}
```

Thread and Memory Safety

Multi-threaded programs must be carefully written to ensure correct behavior. In my design, process_request() is run on a newly created thread. This thread is then detached. This means this thread will run independently of main. This also means that this thread could continue executing even after main exits. As such, the client socket is copied and passed by value to process_request(). I also used the underlying API **select()** in the method await(). This means that instead of blocking on accept indefinitely, we will instead block on await for a timeout, set to 5 seconds. This will allow our program to exit gracefully, instead of being killed while in an accept() call.

In order to have correct functionality, process_request() should have no side effects. While read() and write() could rely on the ServerSocket which belongs to main, exceptions handle this in case the ServerSocket gets destroyed while the process_request thread is still executing.

In C++, especially when working with older libraries and multi-threaded applications, signals **must** be used. We use CTRL-C, SIGINT, to end the program. Not handling this signal could lead to memory leaks, hanging sockets, half-transmitted data, etc. It is operating-system defined how these resources will be relinquished on SIGINT. As such, a new thread is created at the beginning of main called **signal_thread**. This thread runs detached, and will simply wait for SIGINT. On receiving this signal, it runs the **signal_handler()**, which will tell main the program has been terminated via a global variable. Then, the while loop in main will break on the next iteration. This means all objects will be properly destroyed, all sockets closed, and all memory deleted. These signal handling functions are defined in signa_handler.h.
Running this program through Valgrind reveals no memory leaks or memory read/write errors when ending with CTRL-C.

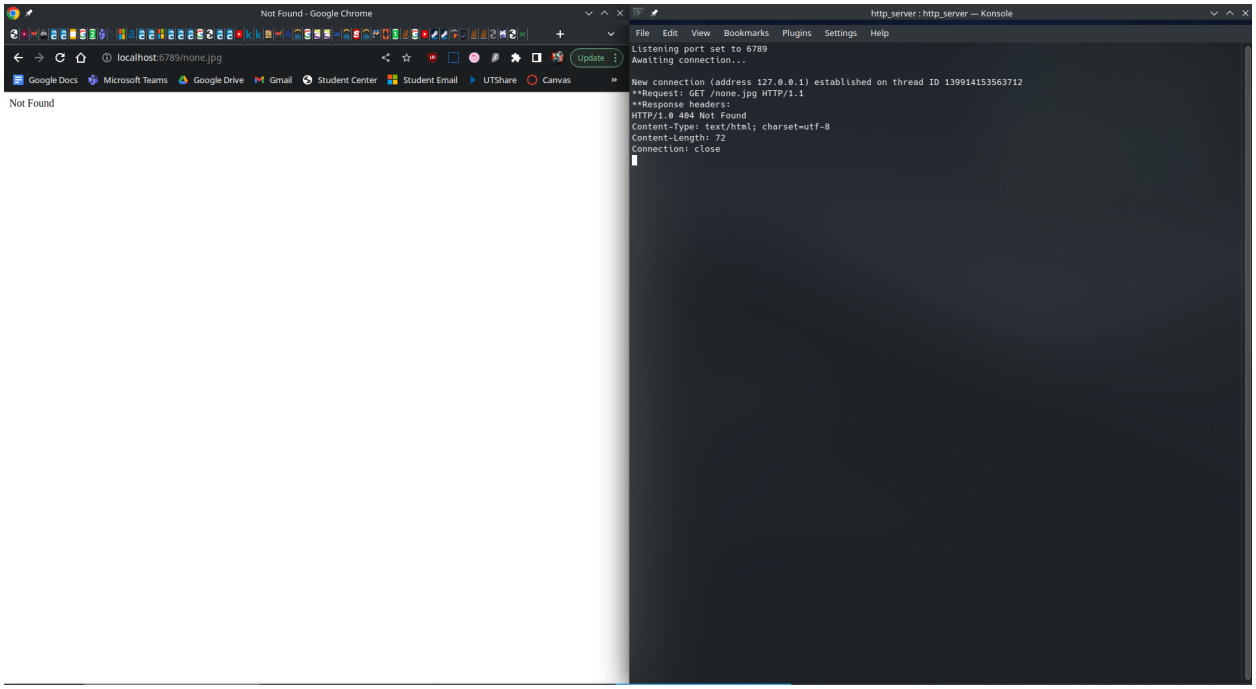
```
==254838== Memcheck, a memory error detector
==254838== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==254838== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==254838== Command: ./http_server
==254838==
Listening port set to 6789
Awaiting connection...

New connection (address 127.0.0.1) established on thread ID 102602304
**Request: GET /index.html HTTP/1.1
**Response headers:
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 12
Connection: close
^C==254838==
==254838== HEAP SUMMARY:
==254838==     in use at exit: 0 bytes in 0 blocks
==254838==   total heap usage: 24 allocs, 24 frees, 85,881 bytes allocated
==254838==
==254838== All heap blocks were freed -- no leaks are possible
==254838==
==254838== For lists of detected and suppressed errors, rerun with: -s
==254838== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Proper program termination

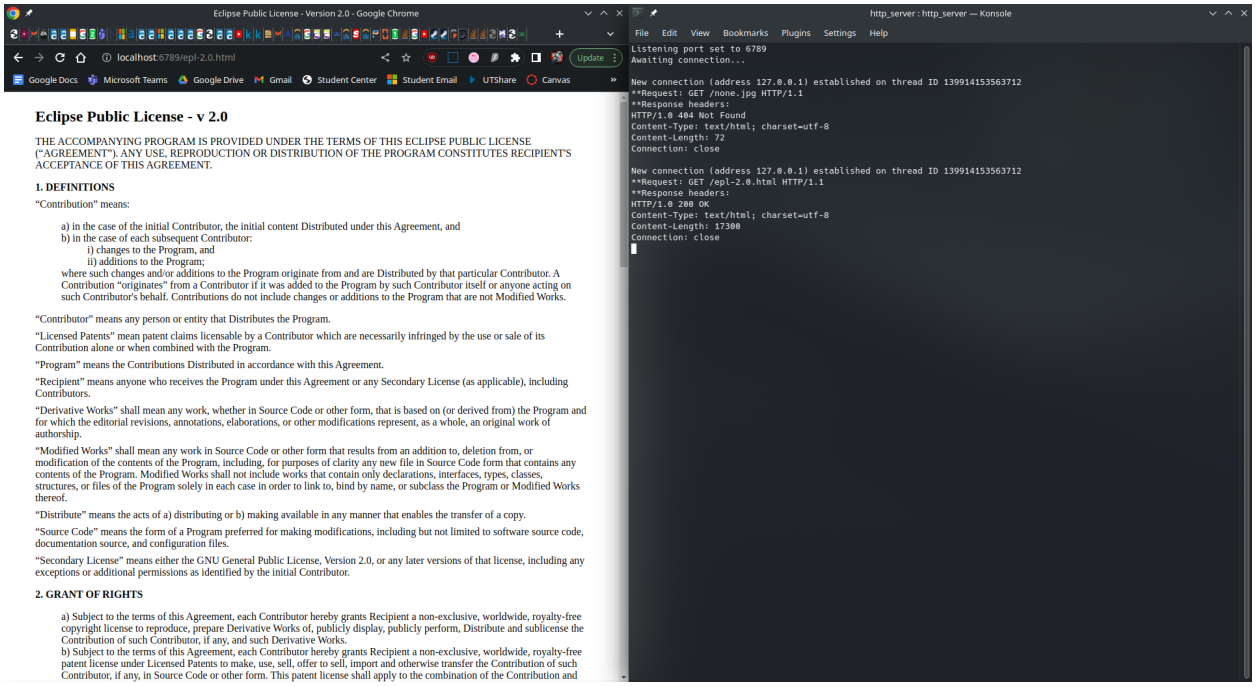
Results and Testing

This section will display program and browser output for some test cases. The first test case is a file which does not exist. Our program should send 404 and our custom 404 message to the browser.



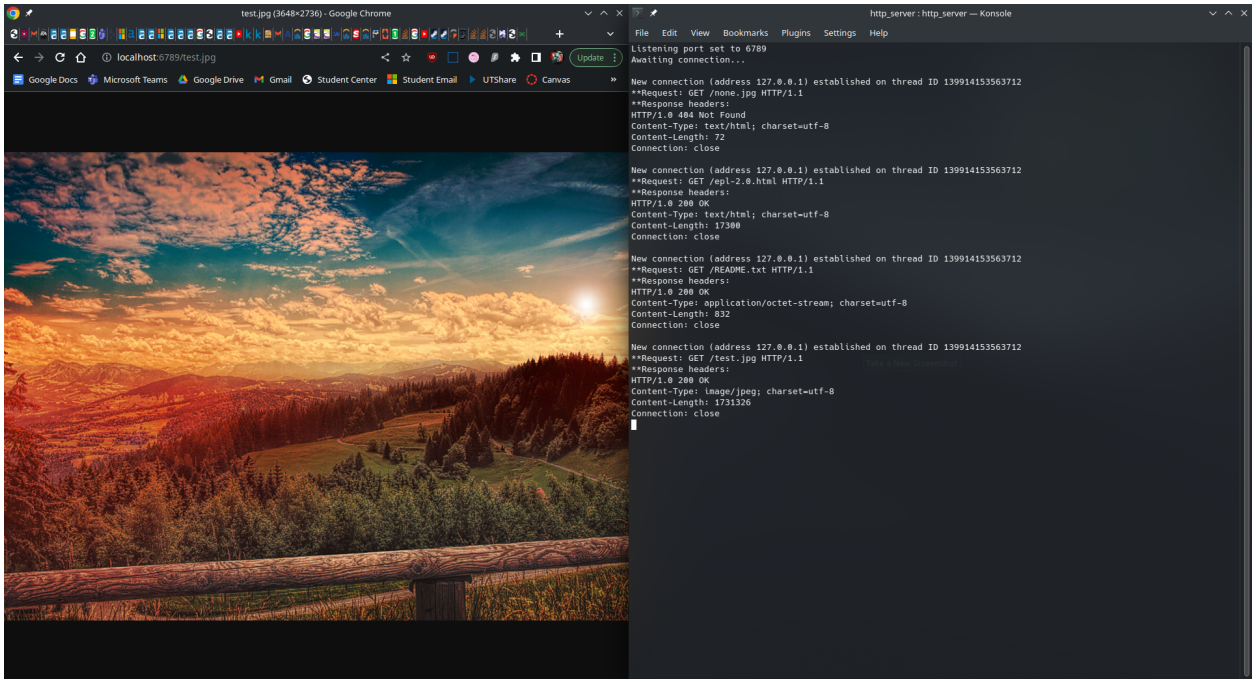
Proper 404 behavior

Next, the program should be able to send html files or other text files. Both work correctly. A test html file called epl-2.0.html is included. This is the eclipse license agreement.



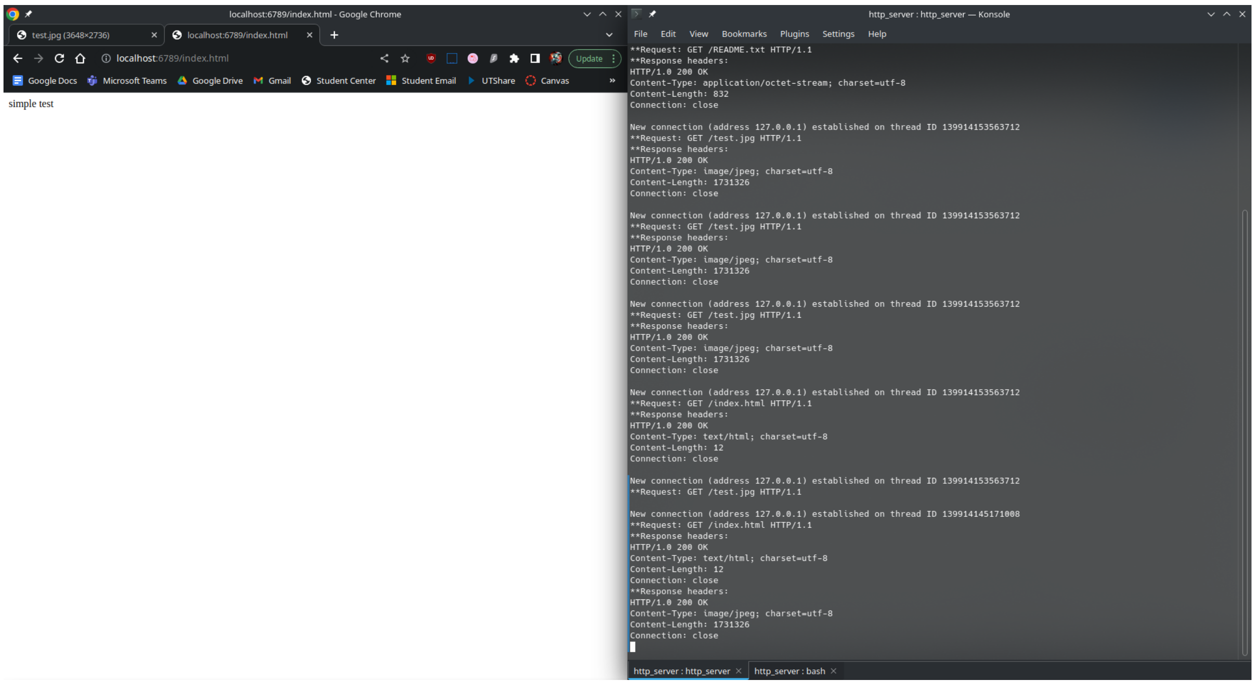
Proper HTML sending

Images can be sent as well. A test image, test.jpg, is included. This is a large 4K image, so it also tests the server for large files.



Large image sent correctly

You may have noticed that the thread IDs appear to be the same. Since the threads are running in detached mode and main does not hold any thread objects, threads run sequentially will reuse thread IDs. Sending concurrent requests reveals that two threads will be spawned concurrently, and that the server is fully multi-threaded. Select multiple Chrome tabs with CTRL and press F5.



Notice the last two requests were received at the same time, and two separate threads with different IDs were spawned. The output from cout gets a bit jumbled since multiple threads are writing to cout concurrently, but the results are correct.