Operating Systems and Systems Programming 2

Coursework

Juan Scerri

123456A

June 24, 2024

A coursework submitted in fulfilment of study unit CPS2008.

Dedicated to my mother, my poor spine and my old computer.

Contents

Report

1 | Introduction

The following is a report detailing the most prevalent aspects of the coursework. In particular, the following aspects were deemed to be of significant importance for the proper treatment of the NetSketch application.

2 | Project Dependencies, Structure and Scripts

2.1 | Dependencies

The dependencies of the project evolved as the development progressed. In particular, the initial list of dependencies was:

- cli11 – a command line parser for C++11,
- fmt – a modern formatting library, and
- raylib – a simple library for game development.

cli11 was immediately co-opted as a dependency after an extensive search for easier methods to implement command line parsing. This is because the standard `getopt` is difficult to use.

fmt is 'considered' a must have because it provides a more modern approach to string manipulation and formatting. In fact, much of the author's work has been co-opted into the STL under `std::format` as of C++20.

raylib was chosen due to previous positive experiences with the library and due to its simplicity. Because of this it allows for the quick and painless setup of basic windows, mouse events, 2D drawing etc.

Later on during development it was realised that (de)serialisation from scratch was going to be a problem. This is because it increases maintenance costs and comes with a healthy serving of endianness related bugs.

A number of possible off-the-shelf solutions where considered. Specifically, protobuf, Cap'n Proto and cereal. The problem with protobuf and Cap'n Proto is the fact that they both use a scheme language to generate source code, which can then be embedded in the project. This in effect allows you to have serialisation-friendly structures which are language agnostic. However, they 'felt' a bit too heavy for this kind of project. Hence, cereal was chosen.

Additionally, cereal does provide some other benefits. It is simple to get started, without requiring the installation of a separate command line tool, it supports serialising STL containers and it supports a portable binary format. This makes it easy to integrate since the relevant data structures do not need to be changed and it abstracts away endianness.

Finally, spdlog was the last dependency to be added. From the very early stages of development it was realised that logging is very important. However, the rudimentary implementation used during the early phases of the project was not ideal for multithreaded programs. This is because printing would often be incorrect or completely halt due to multiple threads trying to print at the same time. Most of these issues are caused by `stdout` being buffered. Instead of continuing to develop a custom solution it was replaced with spdlog.

spdlog solves the above mentioned problems whilst maintaining its own threads, allowing for a much smoother experience.

2.2 | Structure

The source code of our project is split into six subfolders: `bench` (short for benchmarking), `test_client`, `server`, `client`, `common` and `exporter` (bonus). After numerous changes to the code and its structure the above six compartmentalisations where reached.

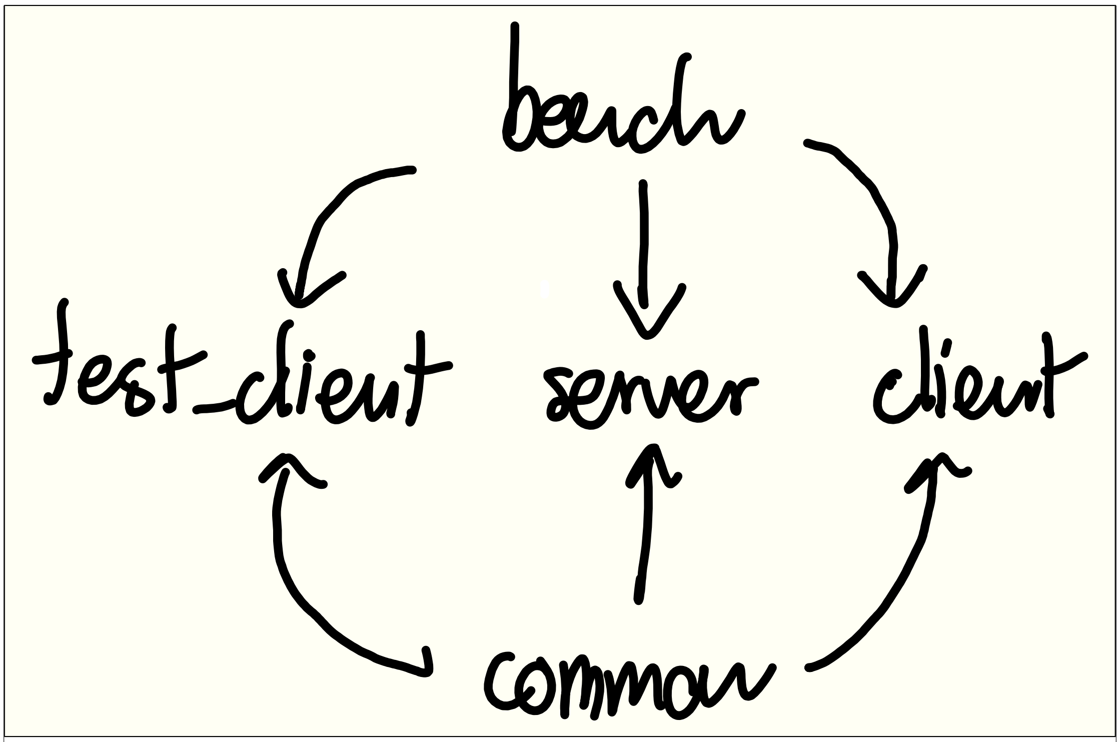


Figure 1: `test_client`, `server` and `client` use the headers in `common` and `bench`.

Now `test_client`, `server` and `client` all contain source code which builds into their respective binaries, those being the test client, the server and the client respectively.

The `bench` folder contains all the necessary code to facilitate benchmarking throughout the project. Of course, benchmarking helps ensure adequate performance. More on this later.

And finally, the `common` folder contains a number of header files. These header files contain code which was either immediately realised to be common (to all the three binaries) or it was duplicated and then later refactored into `common`.

2.3 | Scripts

Throughout development a number of build configurations were setup. As such scripts were written for using such configurations. Additionally, some more complicated scripts were also written.

In particular four different build scripts where created:

- `build-json.sh`,
- `build-rel.sh`,
- `build-tsan.sh`, and
- `build.sh`.

`build.sh` and `build-rel.sh` are default builds, but the first does not optimise (and includes debug information) whilst the other optimises (and includes debug information). Of course, in the event of an actual deployment `build-rel.sh` should be used and stripping debug information out should be considered.

`build-json.sh` is a debug build which enables the server and any test client to dump there final draw vector as a JSON file before exiting. This is of course useful to ensure that proper synchronisation is indeed taking place among the clients and the servers.

Finally, `build-tsan.sh` is again a debug build which incorporates a thread sanitizer into our program. This can be used to ensure that basic multithreading mistakes are avoided.

The next set of scripts is the following:

- `client.sh`,
- `client-debug.sh`,
- `server.sh`, and
- `server-debug.sh`.

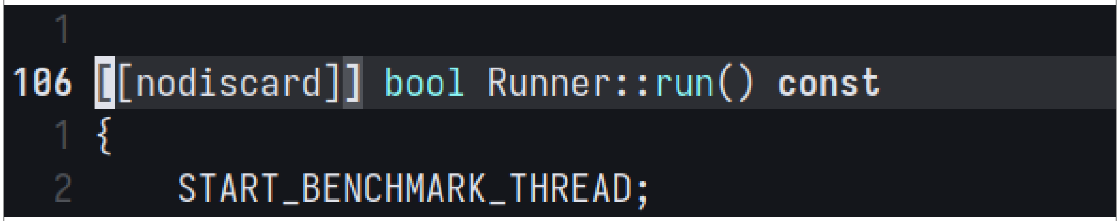
The above are all convenience scripts, they allow running the binaries in `build/` directly from the project root. Additionally, those which have the `-debug` suffix run the binary inside `gdb` to allow for debugging.

The more interesting scripts are `stress-server.sh`, `stress-server-other-actions.sh` and `collect-data.sh`. They facilitate stress testing the server and collecting data about how the server performed. For more information see Section 8.

3 | Benchmarking

A basic mechanism for benchmarking was developed. It is a single header and single source file component and it manages a single thread and a queue.

NOTE The benchmarking facility is only enabled by default for debug builds.

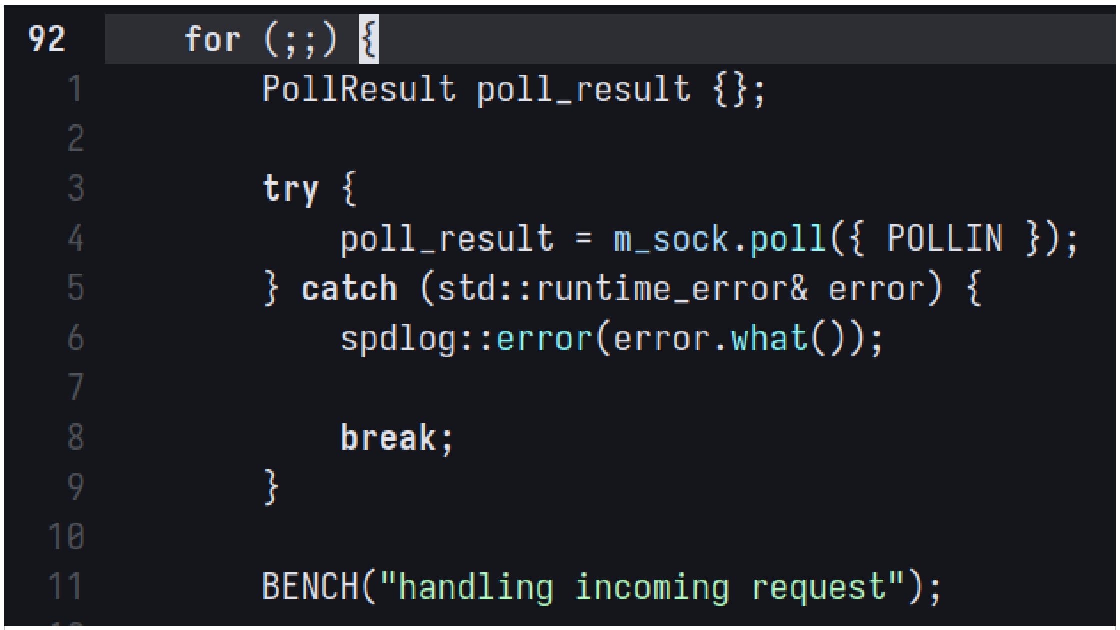


```
[[nodiscard]] bool Runner::run() const
{
    START_BENCHMARK_THREAD;
```

Figure 2: Starting the benchmark thread in `src/server/runner.cpp` at line 108.

During initialisation the benchmarking thread can be started using the `START_BENCHMARK_THREAD` macro (see Figure 2). Then in the different areas of the code which need to be tested, the macro `BENCH("benchmark name")` can be used. The scope of the benchmark is defined by RAII. `BENCH` keeps track of the time at construction, and at destruction it calculates a time delta.

ATTRIB. The usage of RAII for the creation of scoped timers was heavily inspired by Yan Chernikov's guide to simple benchmarking, titled: BENCHMARKING in C++ (how to measure performance) on YouTube.



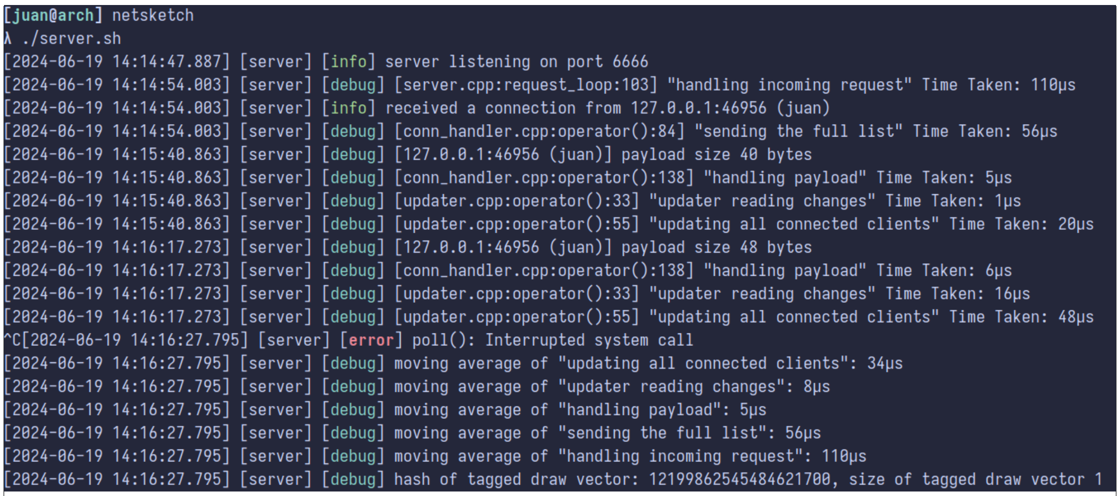
```
    for (;;) {
        PollResult poll_result {};

        try {
            poll_result = m_sock.poll({ POLLIN });
        } catch (std::runtime_error& error) {
            spdlog::error(error.what());

            break;
        }

        BENCH("handling incoming request");
```

Figure 3: Example usage of the `BENCH` macro in the file `src/server/server.cpp` at line 103.

Now calling `BENCH("benchmark name")` by default will log the result of the benchmark to the console. Additionally, it will also enqueue the result as a pair `<"benchmark name", {Time Taken}>` on the benchmark message queue. After which, it notifies the benchmark thread.

NOTE

The string used as a parameter to the `BENCH` macro uniquely identifies a specific benchmark. Additionally, calling `BENCH` multiple times with the same name will result in a moving average of the multiple readings (see Figure 4).



```
[juan@arch] netsketch
λ ./server.sh
[2024-06-19 14:14:47.887] [server] [info] server listening on port 6666
[2024-06-19 14:14:54.003] [server] [debug] [server.cpp:request_loop:103] "handling incoming request" Time Taken: 110µs
[2024-06-19 14:14:54.003] [server] [info] received a connection from 127.0.0.1:46956 (juan)
[2024-06-19 14:14:54.003] [server] [debug] [conn_handler.cpp:operator():84] "sending the full list" Time Taken: 56µs
[2024-06-19 14:15:40.863] [server] [debug] [127.0.0.1:46956 (juan)] payload size 40 bytes
[2024-06-19 14:15:40.863] [server] [debug] [conn_handler.cpp:operator():138] "handling payload" Time Taken: 5µs
[2024-06-19 14:15:40.863] [server] [debug] [updater.cpp:operator():33] "updater reading changes" Time Taken: 1µs
[2024-06-19 14:15:40.863] [server] [debug] [updater.cpp:operator():55] "updating all connected clients" Time Taken: 20µs
[2024-06-19 14:16:17.273] [server] [debug] [127.0.0.1:46956 (juan)] payload size 48 bytes
[2024-06-19 14:16:17.273] [server] [debug] [conn_handler.cpp:operator():138] "handling payload" Time Taken: 6µs
[2024-06-19 14:16:17.273] [server] [debug] [updater.cpp:operator():33] "updater reading changes" Time Taken: 16µs
[2024-06-19 14:16:17.273] [server] [debug] [updater.cpp:operator():55] "updating all connected clients" Time Taken: 48µs
^C[2024-06-19 14:16:27.795] [server] [error] poll(): Interrupted system call
[2024-06-19 14:16:27.795] [server] [debug] moving average of "updating all connected clients": 34µs
[2024-06-19 14:16:27.795] [server] [debug] moving average of "updater reading changes": 8µs
[2024-06-19 14:16:27.795] [server] [debug] moving average of "handling payload": 5µs
[2024-06-19 14:16:27.795] [server] [debug] moving average of "sending the full list": 56µs
[2024-06-19 14:16:27.795] [server] [debug] moving average of "handling incoming request": 110µs
[2024-06-19 14:16:27.795] [server] [debug] hash of tagged draw vector: 12199862545484621700, size of tagged draw vector 1
```

Figure 4: Example of server-side benchmarking (with logging).

Of course at the end of the program's life-time the benchmark thread should be stopped with the macro `END_BENCHMARK_THREAD`.

This part of the project is the backbone which facilitates the work done in Section 8.

4 | Common Headers

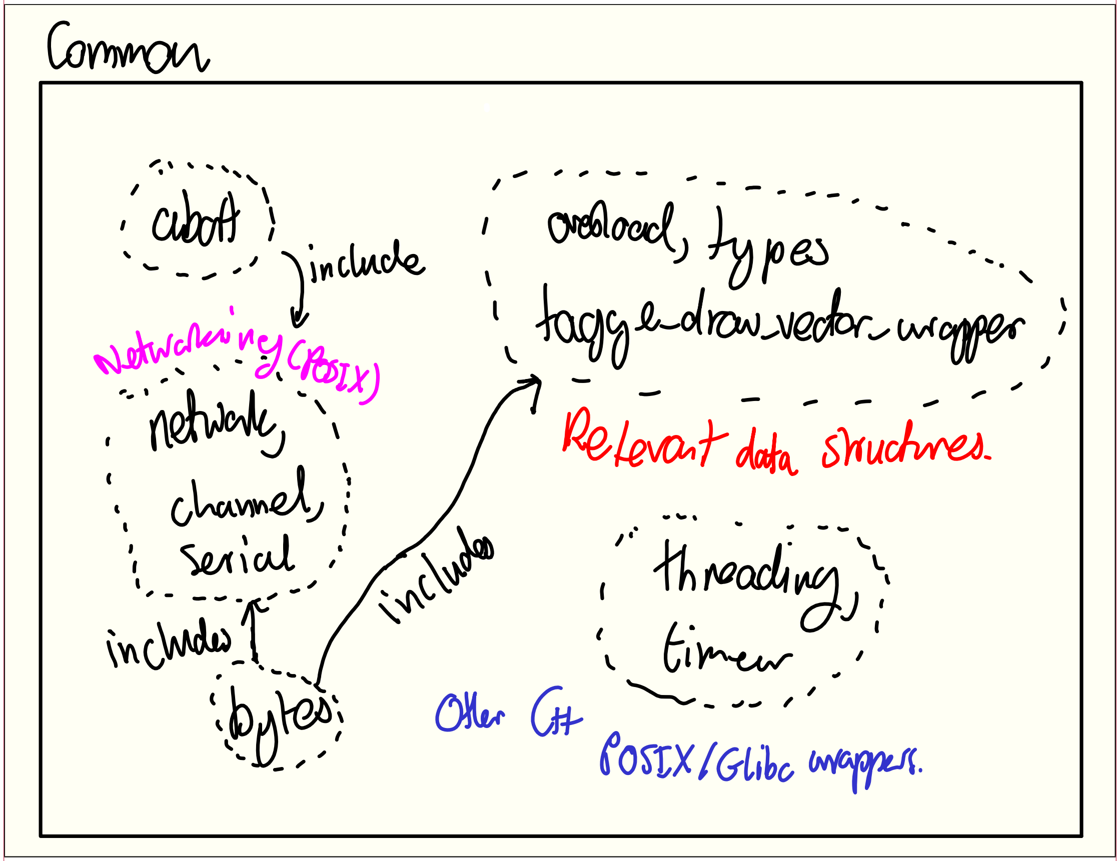
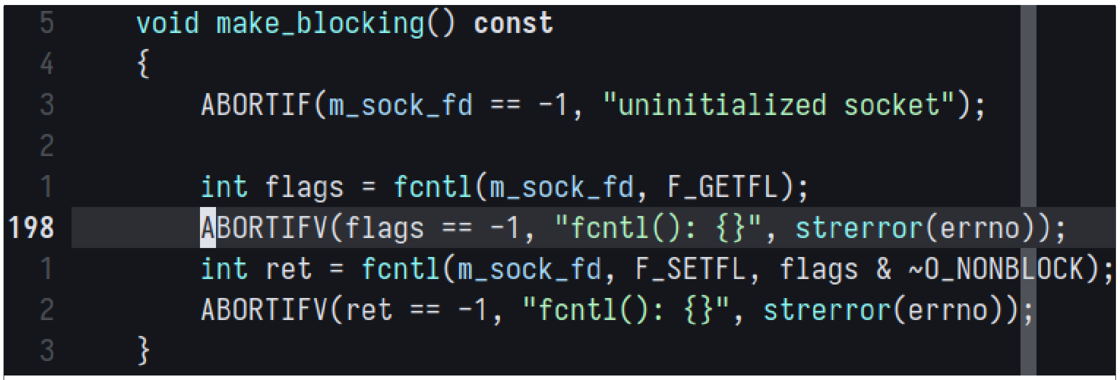


Figure 5: Logical groupings of the files in the common folder.

There are a number of logical groupings in in the common folder (see Figure 5). The smallest of which are `abort.hpp` and `bytes.hpp`. `bytes.hpp` only contains an alias to `std::string` called `ByteString`.



```
void make_blocking() const
{
    ABORTIF(m_sock_fd == -1, "uninitialized socket");

    int flags = fcntl(m_sock_fd, F_GETFL);
    ABORTIFV(flags == -1, "fcntl(): {}", strerror(errno));
    int ret = fcntl(m_sock_fd, F_SETFL, flags & ~O_NONBLOCK);
    ABORTIFV(ret == -1, "fcntl(): {}", strerror(errno));
}
```

Figure 6: Usage of the `ABORTIF` and `ABORTIFV` (variadic variant) macros in `src/common/network.hpp` at line 195.

4.1 | Abort

`abort.hpp` provides some crucial functionality. In particular, it provides mechanisms for abruptly terminating the program. This is useful because there are states that the program should never reach. It should not even be allowed to be in an error state. If such states are indeed reached the program should abort (see Figure 6).

4.2 | Relevant Data Structures (Types)

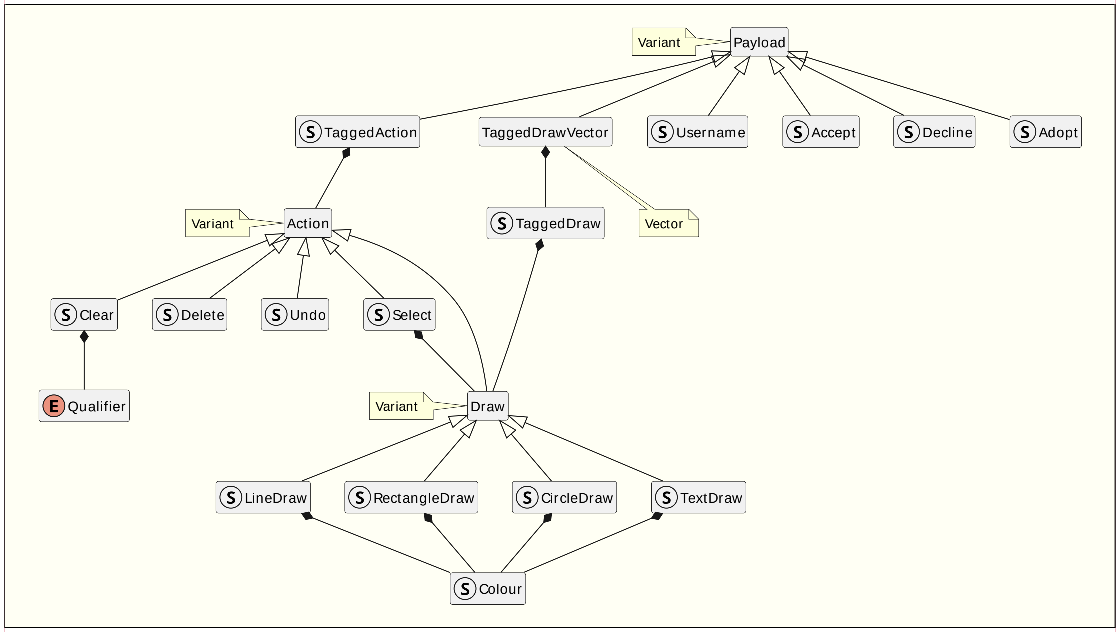


Figure 7: A hierarchy of all the types relevant to the functionality of the whiteboard (in the `src/common/types.hpp` file).

The `types.hpp` file contains all the relevant data types which the server, test client and client use (see Figure 7). The `overload.hpp` file contains some helpers which facilitate more concise and functional syntax for dealing with `std::variant` types (for an example see `src/client/gui.cpp`, line 93).

NOTE

A `Payload` can be a `TaggedAction`. This allows the system to send over deltas or partials instead of sending the full modified state every time the server receives an updated.

The `tagged_draw_vector_wrapper.hpp` contains a class which wraps a tagged draw vector, essentially augmenting the tagged draw vector with the manipulations (e.g. deleting a specific draw command) required by the project specification.

4.3 | Networking (POSIX)

The `serial.hpp` file contains easier interfaces for serialising and de-serialising than those provided by cereal.

The `network.hpp` file contains abstractions around C's POSIX networking primitives. The abstractions allow for safer code by taking advantage of RAII. In particular, the most important part of the file is the `IPv4Socket` class. It creates a significantly easier interface for interacting with sockets.

Additionally, the comment above the `IPv4SocketRef` class is important because it validates the need for such a shell class. In summary, it allows sockets to be copied and moved with being destroyed due to RAII.

The `channel.hpp` file contains the main Channel class. The class wraps around an socket reference and it allows for a simpler approach to over-the-network communication.

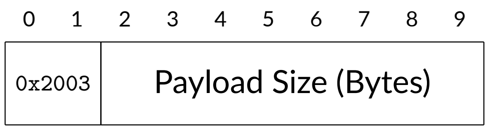


Figure 8: A diagram of the structure of a NetSketch header.

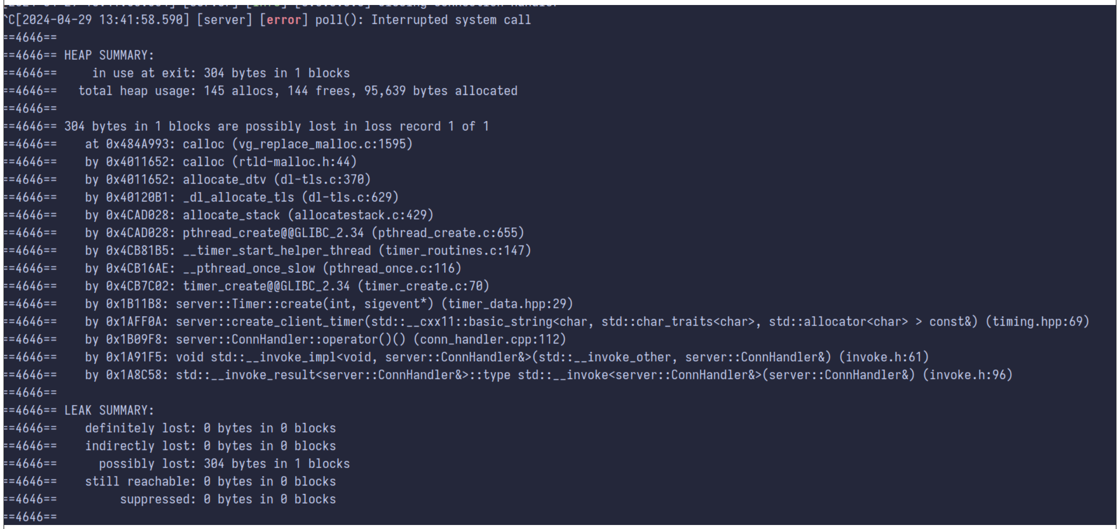
This is because the class automatically wraps any binary string into a packet which is comprised of a header and a payload. The header contains a few two initial bytes which identify the packet as a NetSketch packet and then it contains an 8-byte integer, which indicates how many more bytes should be read.

! The size field is critical for receiving the full message. This is because, the socket interface requires that the number of bytes to be read is provided. This means that the number of bytes a receiver needs to read has to be either fixed or an initial segment of a known fixed size is sent which contains the size of the actual message.

Additionally, the methods within the Channel class always return a `ChannelErrorCode` as part of there return type. This forces the users of channels to properly handle errors and do so in a neater way instead of using `errno` (and everything related to it directly).

4.4 | Other C++ POSIX/`glibc` Wrappers

The `threading.hpp` file is a large collection of C++ wrappers around the threading primitives provided by POSIX. The reason for this is detailed in the beginning of the file. However, the key take away is that the STL does not provide first class support for all the features provided by the POSIX standard. In particular, the most important feature is cancelability (the ability to direct the kernel to stop a thread). Although `std::thread` does use the underlying primitive of its platform abusing this fact and mixing POSIX threads and STL threads seems inapt. Hence, an effort was made to copy the parts of the STL necessary for work, whilst at the same time ensuring 100% compatibility with POSIX.



```
^C[2024-04-29 13:41:58.590] [server] [error] poll(): Interrupted system call
==4646==
==4646== HEAP SUMMARY:
==4646==     in use at exit: 304 bytes in 1 blocks
==4646==   total heap usage: 145 allocs, 144 frees, 95,639 bytes allocated
==4646==
==4646== 304 bytes in 1 blocks are possibly lost in loss record 1 of 1
==4646==    at 0x484A993: calloc (vg_replace_malloc.c:1595)
==4646==    by 0x4011652: calloc (rtld-malloc.h:44)
==4646==    by 0x4011652: allocate_dtv (dl-tls.c:370)
==4646==    by 0x40120B1: _dl_allocate_tls (dl-tls.c:629)
==4646==    by 0x4CAD028: allocate_stack (allocatestack.c:429)
==4646==    by 0x4CAD028: pthread_create@@GLIBC_2.34 (pthread_create.c:655)
==4646==    by 0x4CB81B5: __timer_start_helper_thread (timer_routines.c:147)
==4646==    by 0x4CB16AE: __pthread_once_slow (pthread_once.c:116)
==4646==    by 0x4CB7C02: timer_create@@GLIBC_2.34 (timer_create.c:70)
==4646==    by 0x1B11B8: server::Timer::create(int, sigevent*) (timer_data.hpp:29)
==4646==    by 0x1AFF0A: server::create_client_timer(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&) (timing.hpp:69)
==4646==    by 0x1B09F8: server::ConnHandler::operator()() (conn_handler.cpp:112)
==4646==    by 0x1A91F5: void std::__invoke_impl<void, server::ConnHandler&>(std::__invoke_other, server::ConnHandler&) (invoke.h:61)
==4646==    by 0x1A8C58: std::__invoke_result<server::ConnHandler&>::type std::__invoke<server::ConnHandler&>(server::ConnHandler&) (invoke.h:96)
==4646==
==4646== LEAK SUMMARY:
==4646==    definitely lost: 0 bytes in 0 blocks
==4646==    indirectly lost: 0 bytes in 0 blocks
==4646==      possibly lost: 304 bytes in 1 blocks
==4646==    still reachable: 0 bytes in 0 blocks
==4646==         suppressed: 0 bytes in 0 blocks
==4646==
```

Figure 9: Running `valgrind` on the server binary to test for memory leaks.

Finally, the `timer.hpp` file similarly wraps around the timer mechanisms provided by `glibc` providing a C++ interface which makes use of RAII.

NOTE

`glibc` timers are being used instead of a custom solution to reduce code complexity and reduce the need for managing additional threads (which are used as timers). Instead `glibc` manages these automatically. However, the only downside to this approach is the fact `glibc` always leaks a single allocation (see Figure 9).

5 | The Client

NOTE

A demonstration of the client can be seen in the video presentation hosted here: `https://drive.google.com/file/d/1VMXR4Z0s5iDvXcroMTLn6ddb5enpl0TF/view?usp=drive_link`.

5.1 | Shared State

The approach used for cross-thread communication was shared state. To achieve this a `share.hpp` file was used to contain all the variables which needed to exist outside and hence between threads.

This most includes the majority of used synchronisation primitives such as mutexes, condition variables and read-write locks. Additionally, it contains any threads handles, queues and vectors which need to be global or are used for communication between one thread and another.

5.2 | Client Thread Structure

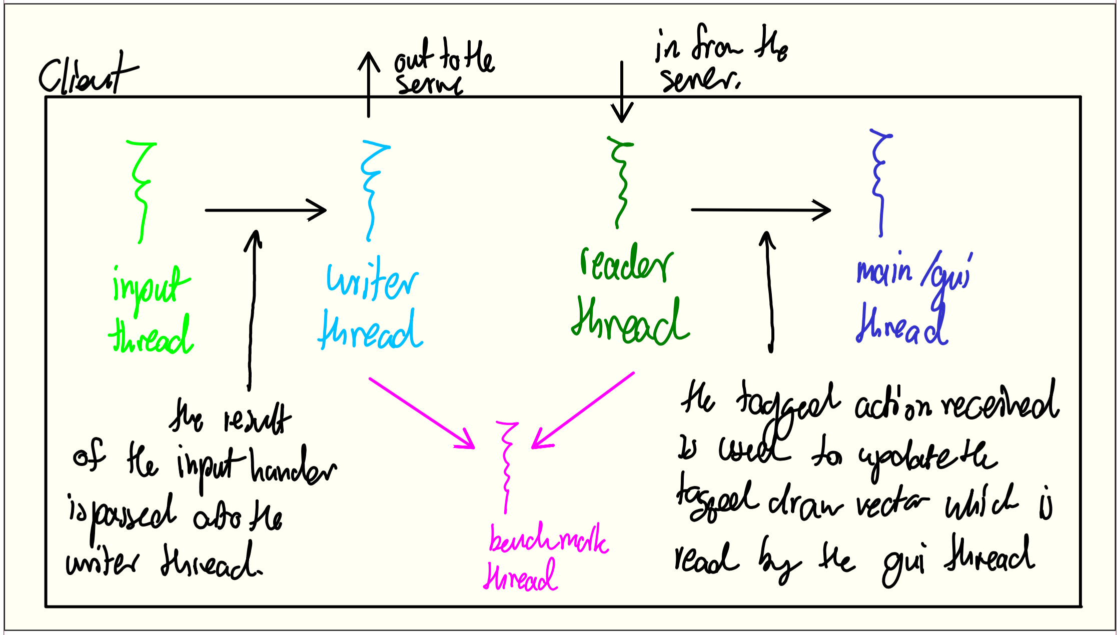

Client
out to the server
in from the server.
input thread
writer thread
reader thread
main/gui thread
the result of the input handler is passed also the writer thread.
benchmark thread
the tagged action received is used to update the tagged draw vector which is read by the gui thread

Figure 10: A drawing of the threads and their interactions in the client binary.

The above Figure 10, gives an adequate depiction of the thread interaction in the client. Expanding a bit, the input thread, generally interacts with the writer thread using the writer queue and, its associated mutex and condition variables (see `src/client/share.hpp` lines 23-25).

The Reader-GUI communication is the more complicated of the two. This is because the camera of the GUI can be managed independently from the client's command line interface. This imposes the following restriction: *the GUI should allow for a smooth user experience even if a whiteboard update is incoming*.

This means that the canvas or GUI should never freeze up during an update.

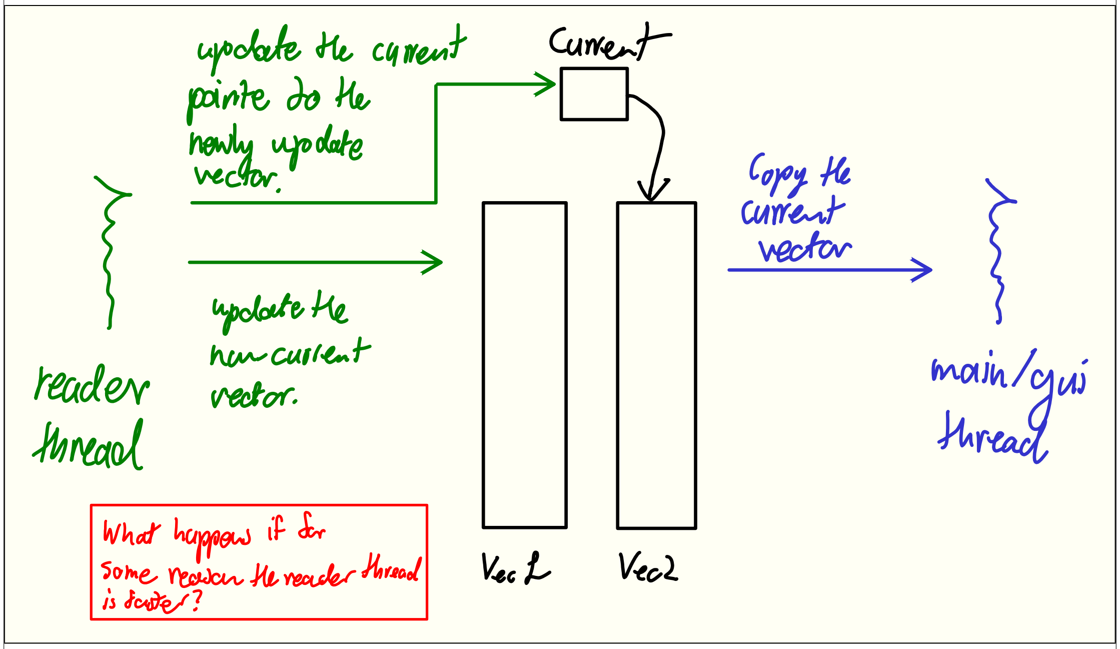

update the current pointe to the newly update vector.
Current
Copy the current vector
update the non-current vector.
reader thread
main/gui thread
What happens if for some reason the reader thread is faster?
Vec 1
Vec2

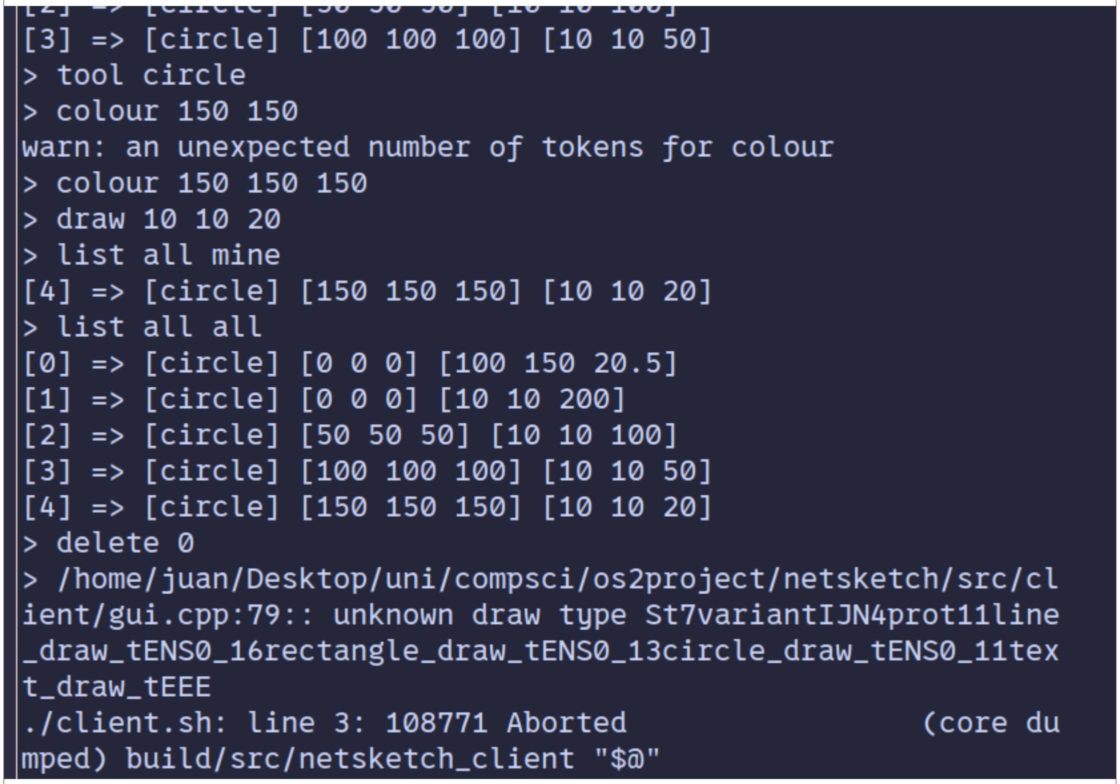
Figure 11: A drawing of the initial double-buffering technique employed to ensure responsiveness.

The main approach for tackling this problem is to use a technique knowing as double buffering. This is most commonly used in graphics related scenarios. A simple implementation should have two vectors which are always in parity but are maintained independently. Whilst a pointer called ‘current’ is used to indicate which of the vectors the GUI thread should read from.

At any point in time the reader thread can update the unused vector. Then after finishing the ‘current’ pointer is updated by the reader and the other vector is also updated, at which point the ‘current’ pointer is swapped again.

However, this approach and specifically trying to avoid the use of synchronisation primitives (as they are very expensive) led to a race condition.

5.3 | Reader-GUI Race Condition (+ Solution)

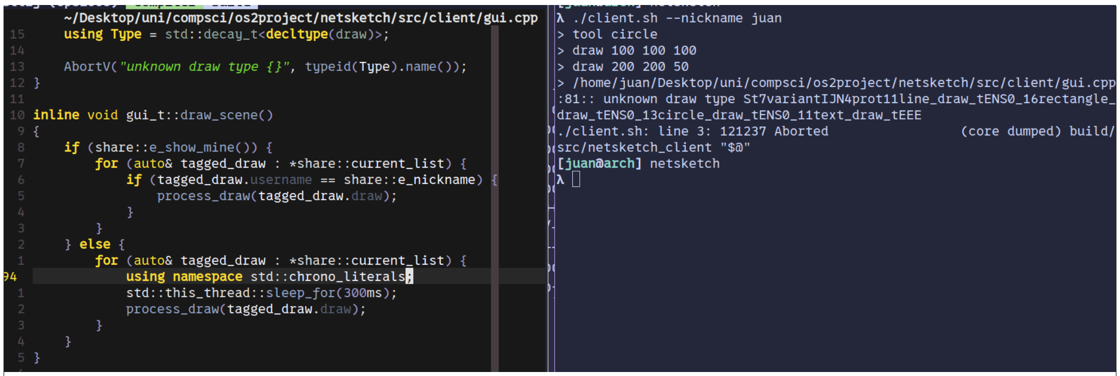


```
[3] => [circle] [100 100 100] [10 10 50]
> tool circle
> colour 150 150
warn: an unexpected number of tokens for colour
> colour 150 150 150
> draw 10 10 20
> list all mine
[4] => [circle] [150 150 150] [10 10 20]
> list all all
[0] => [circle] [0 0 0] [100 150 20.5]
[1] => [circle] [0 0 0] [10 10 200]
[2] => [circle] [50 50 50] [10 10 100]
[3] => [circle] [100 100 100] [10 10 50]
[4] => [circle] [150 150 150] [10 10 20]
> delete 0
> /home/juan/Desktop/uni/compsci/os2project/netsketch/src/cl
ient/gui.cpp:79:: unknown draw type St7variantIJN4prot11line
_draw_tENS0_16rectangle_draw_tENS0_13circle_draw_tENS0_11tex
t_draw_tEEE
./client.sh: line 3: 108771 Aborted                 (core du
mped) build/src/netsketch_client "$@"
```

Figure 12: An abnormal exit of the program caused by an unknown type, later verified to be caused by a data race.

After some digging and research it was later verified to be an issue caused by a data race as indicated by Figure 11. In particular, it was happening because the reader was sometimes faster than the GUI and whilst the GUI was reading, the reader would update the underlying structure. Because of this, the GUI would momentarily read malformed data leading to an abnormal exit (as seen in Figure 12).

Additionally, the issue was appearing irregularly since it is not very common for the GUI thread to be slower than the reader thread. However, to confirm the issue the GUI was artificially slowed downed using a sleep. This confirmed the issue and that it was caused due a timing issue between the reader and GUI threads.



```
~/Desktop/uni/compsci/os2project/netsketch/src/client/gui.cpp
    using Type = std::decay_t<decltype(draw)>;

    AbortV("unknown draw type {}", typeid(Type).name());
}

inline void gui_t::draw_scene()
{
    if (share::e_show_mine()) {
        for (auto& tagged_draw : *share::current_list) {
            if (tagged_draw.username == share::e_nickname) {
                process_draw(tagged_draw.draw);
            }
        }
    } else {
        for (auto& tagged_draw : *share::current_list) {
            using namespace std::chrono_literals;
            std::this_thread::sleep_for(300ms);
            process_draw(tagged_draw.draw);
        }
    }
}
```

```
λ ./client.sh --nickname juan
> tool circle
> draw 100 100 100
> draw 200 200 50
> /home/juan/Desktop/uni/compsci/os2project/netsketch/src/client/gui.cpp
:81:: unknown draw type St7variantIJN4prot11line_draw_tENS0_16rectangle_
draw_tENS0_13circle_draw_tENS0_11text_draw_tEEE
./client.sh: line 3: 121237 Aborted                 (core dumped) build/
src/netsketch_client "$@"
[juan@arch] netsketch
λ
```

Figure 13: Artificially inducing the data race exhibited in Figure 12 using a sleep function.

After some research the following article was found: `http://concurrencyfreaks.blogspot.com/2013/11/double-instance-locking.html`. The author(s) claim that the technique is "lock-free" for (in our case) the GUI thread, which is exactly what was needed to still ensure responsiveness. Implementing there described approach solved the issue.

> **!** Performance before and after the change was not measured. This is because of a number of reasons. In particular, the infrastructure for benchmarking was not yet present in the project and correctness takes precedence over speed.

6 | The Server

6.1 | Shared State

With regards to the sharing of state between threads a similar approach to what was used within the client is also being used within the server. The only difference is that within the server there are more globally shared variables due to the server being overall more complex (see `src/server/share.hpp`).

6.2 | Server Thread Structure

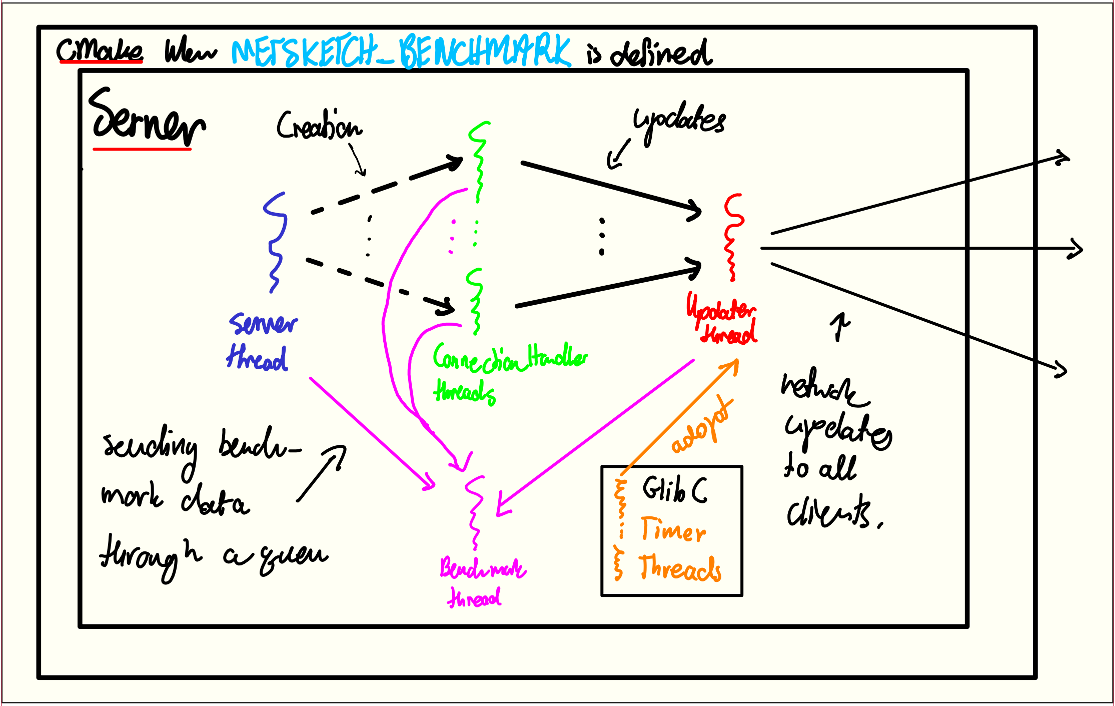

CMake when NETSKETCH_BENCHMARK is defined
Server
Creation
updates
Server thread
Connection Handler threads
Updater thread
network updates to all clients.
sending bench-mark data through a queue
Benchmark thread
adopt
Glib C Timer Threads

Figure 14: A drawing of all the thread interactions in the server.

The threads present within the server are the server thread, the updater, thread the connection handler threads and the timer threads.

In particular, the server thread sets up the necessary networking and handles new incoming connections. When a new connection arrives a connection handler thread is created to handle that specific user. Further communications from the client are handled by its respective connection handler.

Each connection handler then interacts with the updater thread which updates the local state of the server and sends the same update to all the connected clients.

The timer threads are managed by `glibc` and there main purpose is to give a disconnected user a time gap before his draw commands are adopted by the server.

6.3 | Limitations of the Updater Model

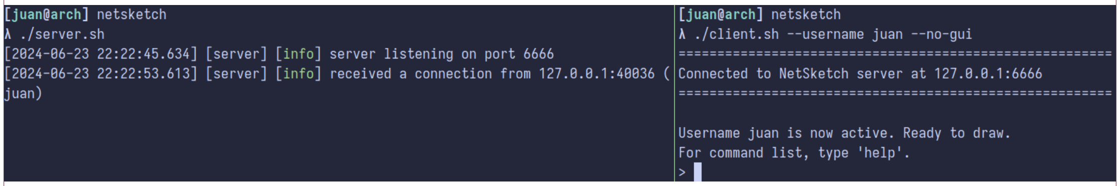
The main problem with the current model is the fact the that the updater thread is a bottleneck. If sufficiently many clients are connected to the server, it will take the updater more time to finish every next update cycle, since it has to send the update to all the connected users in a for-loop.

However, solving this problem is not easy. Using a Publisher-Subscriber model seems adequate however, it will still have the same bottleneck as the current model if internally somewhere a for-loop is notifying each thread.

To solve a problem like this a design which inverts the responsibility of staying notified is required. However, this also has its own drawbacks. Will the threads all continuously poll to check if an update is present? Unfortunately, there does not seem to be an elegant solution which maintains code complexity to a minimum. Because of this fact the initial solution was kept.

6.4 | Other Server Behaviours

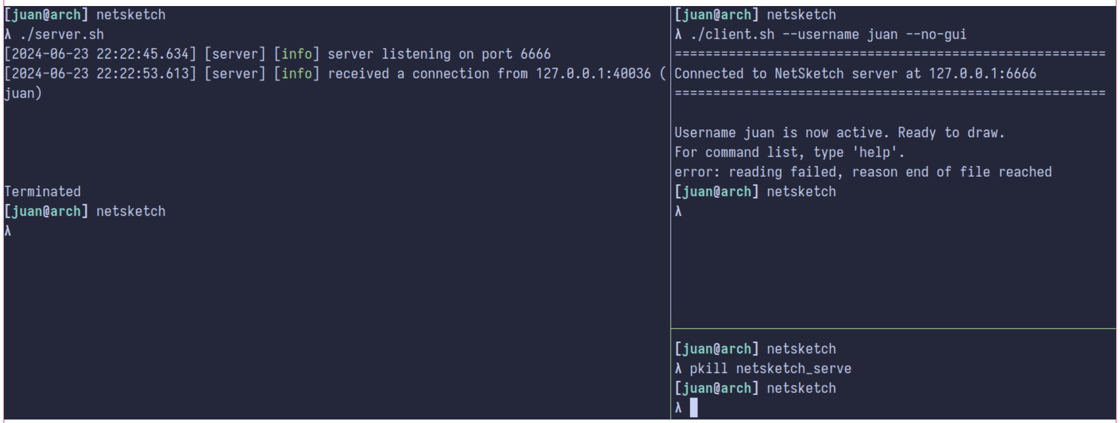
Abrupt Server Termination



```
[juan@arch] netsketch
λ ./server.sh
[2024-06-23 22:22:45.634] [server] [info] server listening on port 6666
[2024-06-23 22:22:53.613] [server] [info] received a connection from 127.0.0.1:40036 (
juan)
```

```
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
========================================================
Connected to NetSketch server at 127.0.0.1:6666
========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
>
```

Figure 15: A screenshot of a operational server and client.



```
[juan@arch] netsketch
λ ./server.sh
[2024-06-23 22:22:45.634] [server] [info] server listening on port 6666
[2024-06-23 22:22:53.613] [server] [info] received a connection from 127.0.0.1:40036 (
juan)




Terminated
[juan@arch] netsketch
λ
```

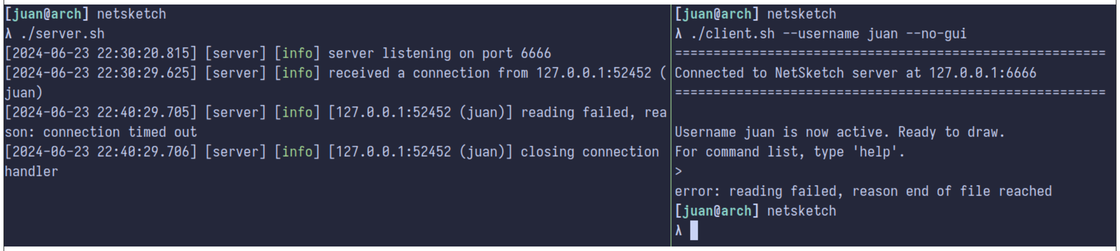
```
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
========================================================
Connected to NetSketch server at 127.0.0.1:6666
========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
error: reading failed, reason end of file reached
[juan@arch] netsketch
λ
```

```
[juan@arch] netsketch
λ pkill netsketch_serve
[juan@arch] netsketch
λ
```

Figure 16: A screenshot of a artificially induced termination (using `pkill`).

In the event that the server abruptly terminates, all connected clients also terminate (see Figure 16).

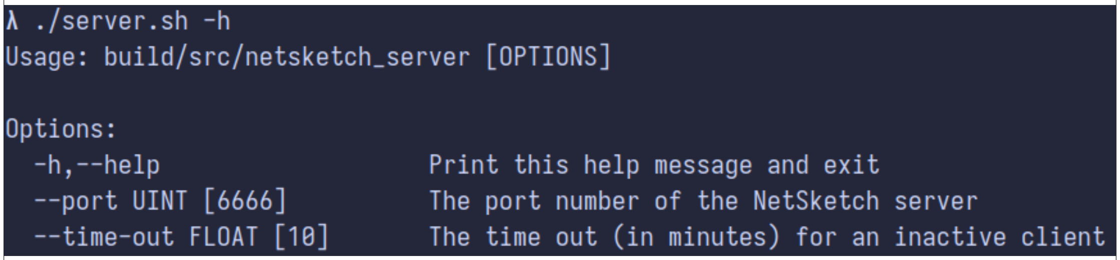
Disconnecting Idle Users



```
[juan@arch] netsketch
λ ./server.sh
[2024-06-23 22:30:20.815] [server] [info] server listening on port 6666
[2024-06-23 22:30:29.625] [server] [info] received a connection from 127.0.0.1:52452 (
juan)
[2024-06-23 22:40:29.705] [server] [info] [127.0.0.1:52452 (juan)] reading failed, rea
son: connection timed out
[2024-06-23 22:40:29.706] [server] [info] [127.0.0.1:52452 (juan)] closing connection
handler
```

```
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
=======================================================
Connected to NetSketch server at 127.0.0.1:6666
=======================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
>
error: reading failed, reason end of file reached
[juan@arch] netsketch
λ
```

Figure 17: A screenshot of a server disconnecting a user that has been idle for ten minutes.

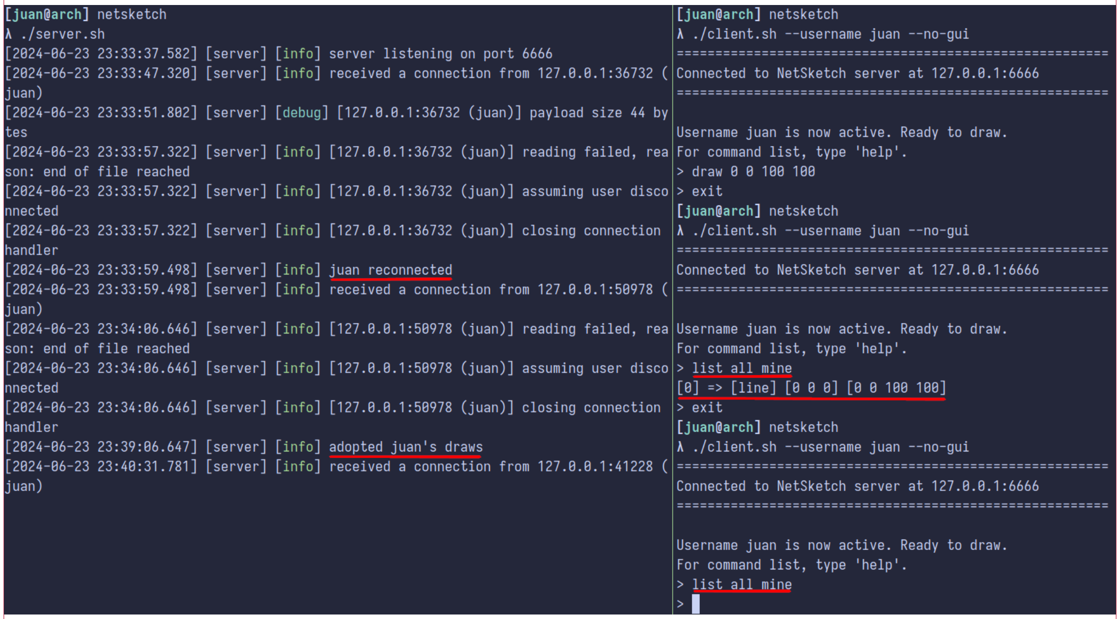


```
λ ./server.sh -h
Usage: build/src/netsketch_server [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  --port UINT [6666]          The port number of the NetSketch server
  --time-out FLOAT [10]       The time out (in minutes) for an inactive client
```

Figure 18: A screenshot of the server's help menu.

The server can disconnect idle clients. The default timeout is set to ten minutes. However, this can be changed to any number of minutes (see Figure 18).

Draw Call Adoption



```
[juan@arch] netsketch
λ ./server.sh
[2024-06-23 23:33:37.582] [server] [info] server listening on port 6666
[2024-06-23 23:33:47.320] [server] [info] received a connection from 127.0.0.1:36732 (
juan)
[2024-06-23 23:33:51.802] [server] [debug] [127.0.0.1:36732 (juan)] payload size 44 by
tes
[2024-06-23 23:33:57.322] [server] [info] [127.0.0.1:36732 (juan)] reading failed, rea
son: end of file reached
[2024-06-23 23:33:57.322] [server] [info] [127.0.0.1:36732 (juan)] assuming user disco
nnected
[2024-06-23 23:33:57.322] [server] [info] [127.0.0.1:36732 (juan)] closing connection
handler
[2024-06-23 23:33:59.498] [server] [info] juan reconnected
[2024-06-23 23:33:59.498] [server] [info] received a connection from 127.0.0.1:50978 (
juan)
[2024-06-23 23:34:06.646] [server] [info] [127.0.0.1:50978 (juan)] reading failed, rea
son: end of file reached
[2024-06-23 23:34:06.646] [server] [info] [127.0.0.1:50978 (juan)] assuming user disco
nnected
[2024-06-23 23:34:06.646] [server] [info] [127.0.0.1:50978 (juan)] closing connection
handler
[2024-06-23 23:39:06.647] [server] [info] adopted juan's draws
[2024-06-23 23:40:31.781] [server] [info] received a connection from 127.0.0.1:41228 (
juan)
```

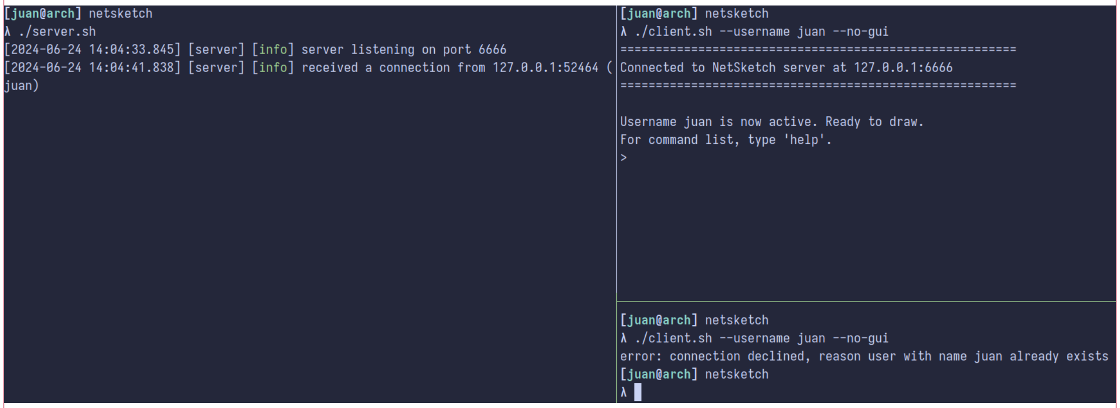
```
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
=========================================================
Connected to NetSketch server at 127.0.0.1:6666
=========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
> draw 0 0 100 100
> exit
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
=========================================================
Connected to NetSketch server at 127.0.0.1:6666
=========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
> list all mine
[0] => [line] [0 0 0] [0 0 100 100]
> exit
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
=========================================================
Connected to NetSketch server at 127.0.0.1:6666
=========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
> list all mine
>
```

Figure 19: A screenshot demonstrating draw command adoption by the server (the relevant server logs and client info if highlighted).

When a client disconnects the server preserves the user's ownership of his/her draw commands for five minutes. After that the server adopts the commands and if a user connects they'll no longer have ownership of those draw commands.

In the event that a client unexpectedly disconnects, this approach gives the client a buffer to reconnect without losing their progress and hence continue where they left off.

Clashing Usernames



```
[juan@arch] netsketch
λ ./server.sh
[2024-06-24 14:04:33.845] [server] [info] server listening on port 6666
[2024-06-24 14:04:41.838] [server] [info] received a connection from 127.0.0.1:52464 (
juan)
```

```
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
=========================================================
Connected to NetSketch server at 127.0.0.1:6666
=========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
>
```

```
[juan@arch] netsketch
λ ./client.sh --username juan --no-gui
error: connection declined, reason user with name juan already exists
[juan@arch] netsketch
λ
```

Figure 20: A screenshot two clients requesting the same name. One of the clients is refused by the server.

For every possible username the server only supports a single connection.

7 | The Test Client

The test client is a variation of the actual client. It takes in as input the following extra command line arguments:

- `--iterations` which is the number of requests the test client will send to the server,
- `--interval` which is the interval in between each request in seconds,
- `--expected_responses` which is **suppose** to be the total number of responses the test client will receive (however it refers to the total number of expected draw commands), and
- `--other_actions` which indicates whether generating undo, select and clear should be allowed or not.

! The `--expected_responses` flag's behaviour is prefixed by a '**suppose**'. This is because it was realised that calculating the number of expected responses especially, if multiple test clients are present is extremely difficult. This is the case because if multiple tests clients are present some will actually connect to the server later than others. This means that they will receive all the changes events up to that point as one whole tagged draw vector i.e. one response. Hence, calculating the number of expected responses is impossible. Instead the number of expected responses actually refers to the number of draw calls or the final expected size of the tagged draw vector. Additionally, note that if the other actions were also enabled it would also make the calculation impossible.

The `--expected_responses` flag is important because it tells the test client when to stop and terminate. In the event that the quota set by the `--expected_responses` flag is never met, the test client will wait until the server automatically, disconnects the client due to inactivity (circa ten minutes).

The most critical behaviour of a test client is its generation of random actions which is contained in the file `src/test_client/simulate_user.cpp`. It is a simple for-loop which uniformly generates all the available drawing commands (if the `--other_actions` flag is set it will generate all actions uniformly).

Thread Structure

The thread structure of the test client is similar to the normal client except it does not have a dedicated input thread and the GUI/main is used for running the above mentioned random action/draw generator.

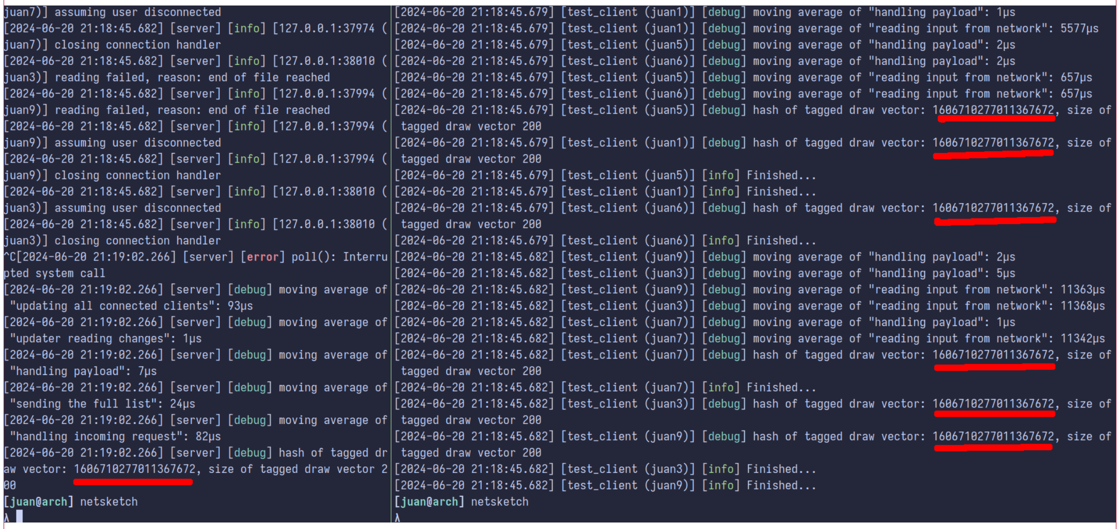
8 | Stress/Performance Testing

8.1 | Testing the Server

Now having established a test client, the scripts `stress-server.sh` and `stress-server-other-actions.sh` can be used to , of course, stress the server and test for correctness.

Additionally, the `collect-data.sh` script can be used to gather information about the server, in particular, its performance under heavy load.

Testing Correctness



```
juan7)] assuming user disconnected
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:37974 (
juan7)] closing connection handler
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:38010 (
juan3)] reading failed, reason: end of file reached
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:37994 (
juan9)] reading failed, reason: end of file reached
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:37994 (
juan9)] assuming user disconnected
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:37994 (
juan9)] closing connection handler
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:38010 (
juan3)] assuming user disconnected
[2024-06-20 21:18:45.682] [server] [info] [127.0.0.1:38010 (
juan3)] closing connection handler
^C[2024-06-20 21:19:02.266] [server] [error] poll(): Interru
pted system call
[2024-06-20 21:19:02.266] [server] [debug] moving average of
 "updating all connected clients": 93µs
[2024-06-20 21:19:02.266] [server] [debug] moving average of
 "updater reading changes": 1µs
[2024-06-20 21:19:02.266] [server] [debug] moving average of
 "handling payload": 7µs
[2024-06-20 21:19:02.266] [server] [debug] moving average of
 "sending the full list": 24µs
[2024-06-20 21:19:02.266] [server] [debug] moving average of
 "handling incoming request": 82µs
[2024-06-20 21:19:02.266] [server] [debug] hash of tagged dr
aw vector: 1606710277011367672, size of tagged draw vector 2
00
[juan@arch] netsketch
λ
```

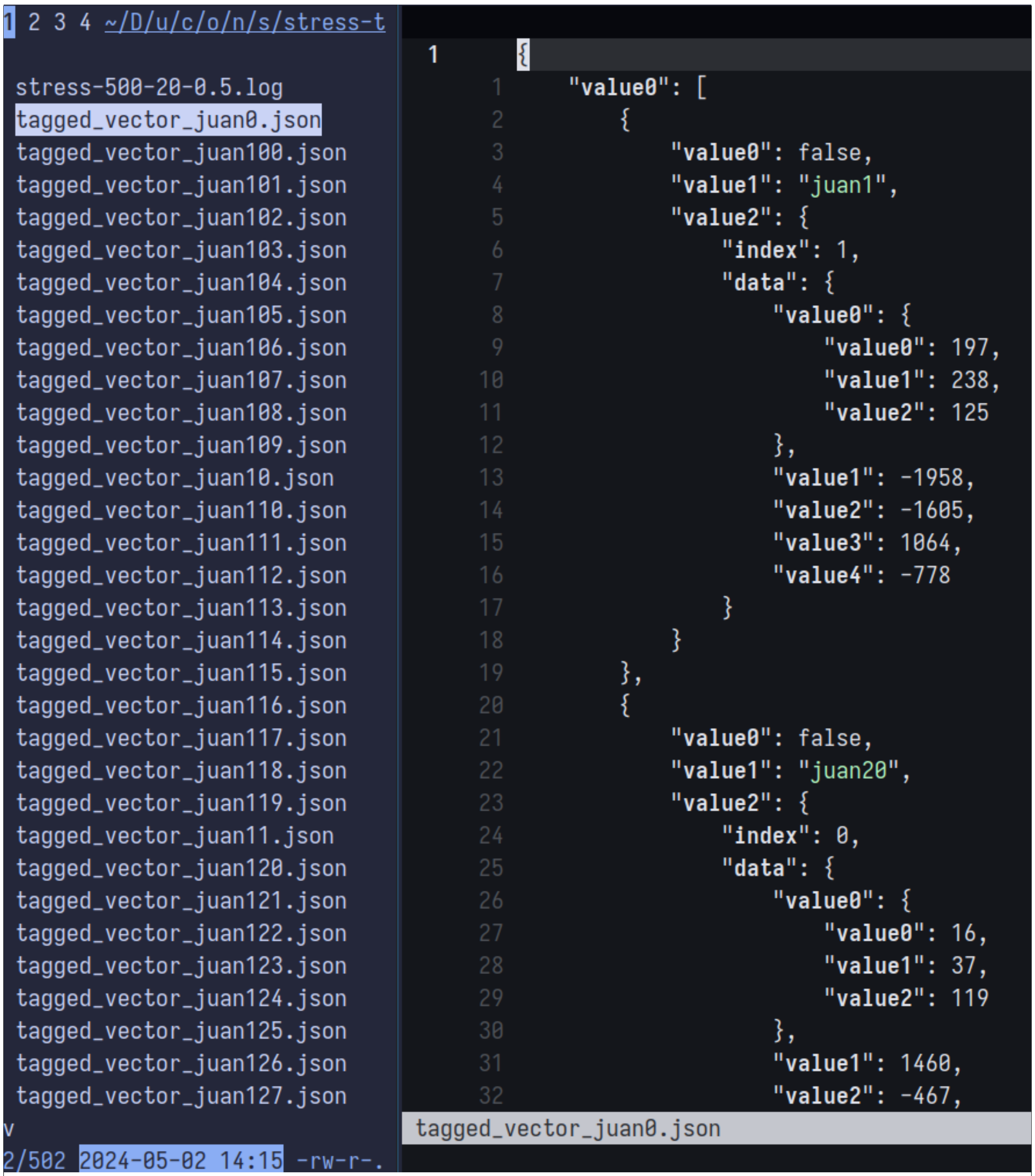
```
[2024-06-20 21:18:45.679] [test_client (juan1)] [debug] moving average of "handling payload": 1µs
[2024-06-20 21:18:45.679] [test_client (juan1)] [debug] moving average of "reading input from network": 5577µs
[2024-06-20 21:18:45.679] [test_client (juan5)] [debug] moving average of "handling payload": 2µs
[2024-06-20 21:18:45.679] [test_client (juan6)] [debug] moving average of "handling payload": 2µs
[2024-06-20 21:18:45.679] [test_client (juan5)] [debug] moving average of "reading input from network": 657µs
[2024-06-20 21:18:45.679] [test_client (juan6)] [debug] moving average of "reading input from network": 657µs
[2024-06-20 21:18:45.679] [test_client (juan5)] [debug] hash of tagged draw vector: 1606710277011367672, size of
 tagged draw vector 200
[2024-06-20 21:18:45.679] [test_client (juan1)] [debug] hash of tagged draw vector: 1606710277011367672, size of
 tagged draw vector 200
[2024-06-20 21:18:45.679] [test_client (juan5)] [info] Finished...
[2024-06-20 21:18:45.679] [test_client (juan1)] [info] Finished...
[2024-06-20 21:18:45.679] [test_client (juan6)] [debug] hash of tagged draw vector: 1606710277011367672, size of
 tagged draw vector 200
[2024-06-20 21:18:45.679] [test_client (juan6)] [info] Finished...
[2024-06-20 21:18:45.682] [test_client (juan9)] [debug] moving average of "handling payload": 2µs
[2024-06-20 21:18:45.682] [test_client (juan3)] [debug] moving average of "handling payload": 5µs
[2024-06-20 21:18:45.682] [test_client (juan9)] [debug] moving average of "reading input from network": 11363µs
[2024-06-20 21:18:45.682] [test_client (juan3)] [debug] moving average of "reading input from network": 11368µs
[2024-06-20 21:18:45.682] [test_client (juan7)] [debug] moving average of "handling payload": 1µs
[2024-06-20 21:18:45.682] [test_client (juan7)] [debug] moving average of "reading input from network": 11342µs
[2024-06-20 21:18:45.682] [test_client (juan7)] [debug] hash of tagged draw vector: 1606710277011367672, size of
 tagged draw vector 200
[2024-06-20 21:18:45.682] [test_client (juan7)] [info] Finished...
[2024-06-20 21:18:45.682] [test_client (juan3)] [debug] hash of tagged draw vector: 1606710277011367672, size of
 tagged draw vector 200
[2024-06-20 21:18:45.682] [test_client (juan9)] [debug] hash of tagged draw vector: 1606710277011367672, size of
 tagged draw vector 200
[2024-06-20 21:18:45.682] [test_client (juan3)] [info] Finished...
[2024-06-20 21:18:45.682] [test_client (juan9)] [info] Finished...
[juan@arch] netsketch
λ
```

Figure 21: Stressing the server with ten test clients and twenty draw commands at an interval of 0.5 seconds (with the tagged draw vector hashes highlighted).

NOTE Under debug builds the tagged draw vector can be hashed. This allows us to compare the hashes generated by the different test clients and the server.

When testing for correctness an initial stress test using the debug build is performed. This stress tests yields the hashes of the tagged draw vectors held by each test client and the server (see Figure 21).

These hashes can then be compared. Any discrepancies are indicative of issues with correctness, specifically synchronisation.



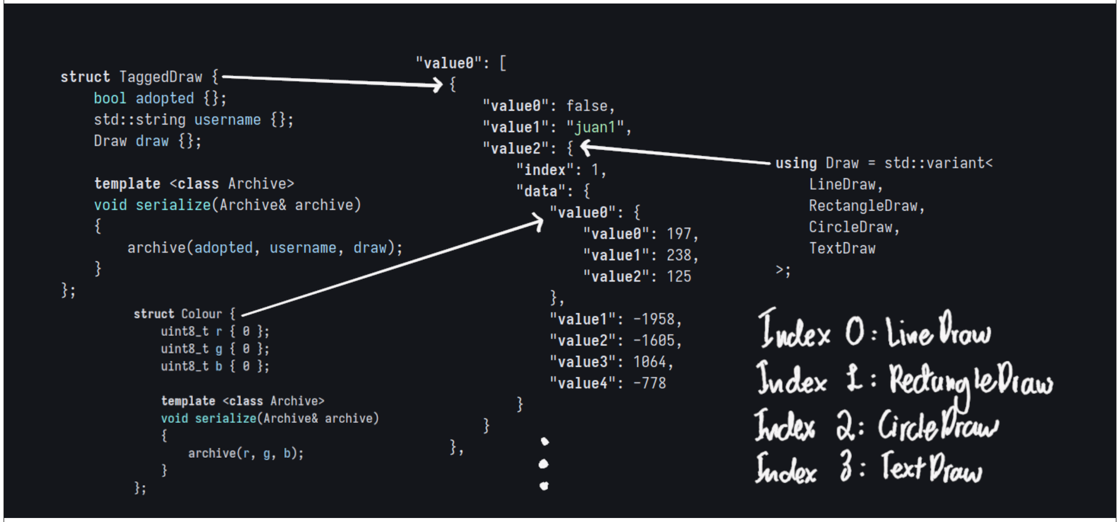
```
1 2 3 4 ~/D/u/c/o/n/s/stress-t

stress-500-20-0.5.log
tagged_vector_juan0.json
tagged_vector_juan100.json
tagged_vector_juan101.json
tagged_vector_juan102.json
tagged_vector_juan103.json
tagged_vector_juan104.json
tagged_vector_juan105.json
tagged_vector_juan106.json
tagged_vector_juan107.json
tagged_vector_juan108.json
tagged_vector_juan109.json
tagged_vector_juan10.json
tagged_vector_juan110.json
tagged_vector_juan111.json
tagged_vector_juan112.json
tagged_vector_juan113.json
tagged_vector_juan114.json
tagged_vector_juan115.json
tagged_vector_juan116.json
tagged_vector_juan117.json
tagged_vector_juan118.json
tagged_vector_juan119.json
tagged_vector_juan11.json
tagged_vector_juan120.json
tagged_vector_juan121.json
tagged_vector_juan122.json
tagged_vector_juan123.json
tagged_vector_juan124.json
tagged_vector_juan125.json
tagged_vector_juan126.json
tagged_vector_juan127.json
v
2/502 2024-05-02 14:15 -rw-r-.
```

```
{
    "value0": [
        {
            "value0": false,
            "value1": "juan1",
            "value2": {
                "index": 1,
                "data": {
                    "value0": {
                        "value0": 197,
                        "value1": 238,
                        "value2": 125
                    },
                    "value1": -1958,
                    "value2": -1605,
                    "value3": 1064,
                    "value4": -778
                }
            }
        },
        {
            "value0": false,
            "value1": "juan20",
            "value2": {
                "index": 0,
                "data": {
                    "value0": {
                        "value0": 16,
                        "value1": 37,
                        "value2": 119
                    },
                    "value1": 1460,
                    "value2": -467,
tagged_vector_juan0.json
```

Figure 22: A collection of JSON dumps along with an open JSON file.

In the event that two or more of these hashes are different, the binaries can be recompiled with `build-json.sh`. This allows each of the test clients and the server to dump their local tagged draw vector as a JSON file. These files can then be manually inspected.

Of course, this does not guarantee an easier time debugging however, it can provide insight as to why the observed behaviour is being recorded.



```
struct TaggedDraw {
    bool adopted {};
    std::string username {};
    Draw draw {};

    template <class Archive>
    void serialize(Archive& archive)
    {
        archive(adopted, username, draw);
    }
};

        struct Colour {
            uint8_t r { 0 };
            uint8_t g { 0 };
            uint8_t b { 0 };

            template <class Archive>
            void serialize(Archive& archive)
            {
                archive(r, g, b);
            }
        };
```

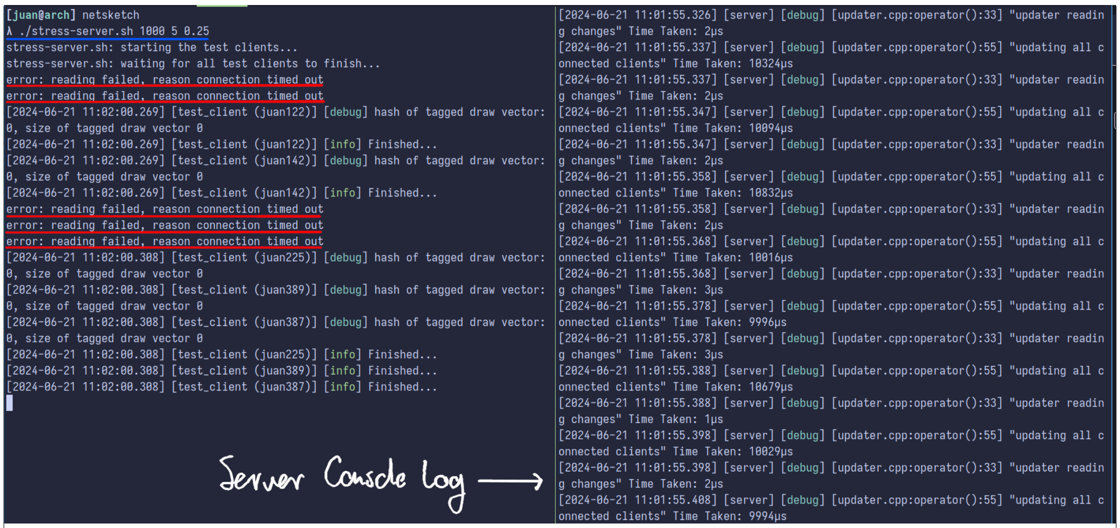
```
"value0": [
    {
        "value0": false,
        "value1": "juan1",
        "value2": {
            "index": 1,
            "data": {
                "value0": {
                    "value0": 197,
                    "value1": 238,
                    "value2": 125
                },
                "value1": -1958,
                "value2": -1605,
                "value3": 1064,
                "value4": -778
            }
        }
    },
    .
    .
    .
```

```
using Draw = std::variant<
    LineDraw,
    RectangleDraw,
    CircleDraw,
    TextDraw
>;
```

Figure 23: A graphical guide to understanding a dumped JSON file.

NOTE

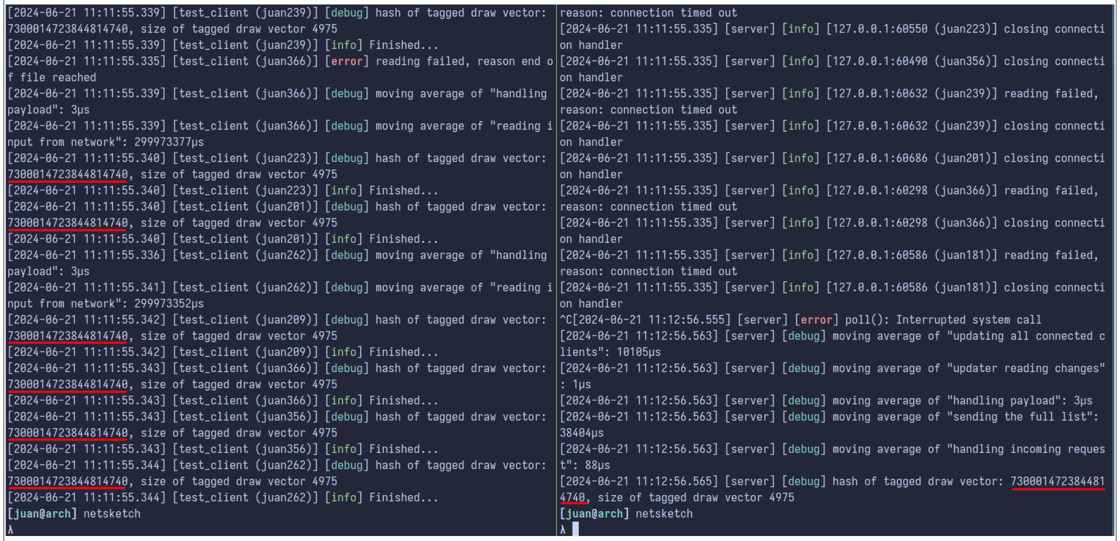
The dumped JSON files (see Figure 22) lack proper names for the fields. This is because the cereal serialisation library only supports the addition of such names when serialising into XML or JSON. However, the program is also serialising into portable binary making it a bit more difficult to annotate fields with names. Please see Figure 23 for a guide on reading the JSON files.



```
[juan@arch] netsketch
λ ./stress-server.sh 1000 5 0.25
stress-server.sh: starting the test clients...
stress-server.sh: waiting for all test clients to finish...
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
[2024-06-21 11:02:00.269] [test_client (juan122)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 11:02:00.269] [test_client (juan122)] [info] Finished...
[2024-06-21 11:02:00.269] [test_client (juan142)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 11:02:00.269] [test_client (juan142)] [info] Finished...
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
[2024-06-21 11:02:00.308] [test_client (juan225)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 11:02:00.308] [test_client (juan389)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 11:02:00.308] [test_client (juan387)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 11:02:00.308] [test_client (juan225)] [info] Finished...
[2024-06-21 11:02:00.308] [test_client (juan389)] [info] Finished...
[2024-06-21 11:02:00.308] [test_client (juan387)] [info] Finished...
```

```
[2024-06-21 11:01:55.326] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 2μs
[2024-06-21 11:01:55.337] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 10324μs
[2024-06-21 11:01:55.337] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 2μs
[2024-06-21 11:01:55.347] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 10094μs
[2024-06-21 11:01:55.347] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 2μs
[2024-06-21 11:01:55.358] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 10832μs
[2024-06-21 11:01:55.358] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 2μs
[2024-06-21 11:01:55.368] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 10016μs
[2024-06-21 11:01:55.368] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 3μs
[2024-06-21 11:01:55.378] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 9996μs
[2024-06-21 11:01:55.378] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 3μs
[2024-06-21 11:01:55.388] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 10679μs
[2024-06-21 11:01:55.388] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 1μs
[2024-06-21 11:01:55.398] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 10029μs
[2024-06-21 11:01:55.398] [server] [debug] [updater.cpp:operator():33] "updater readin
g changes" Time Taken: 2μs
[2024-06-21 11:01:55.408] [server] [debug] [updater.cpp:operator():55] "updating all c
onnected clients" Time Taken: 9994μs
```

Figure 24: Initial phase of a stress test with 1000 test clients (the command and timeout errors are highlighted).



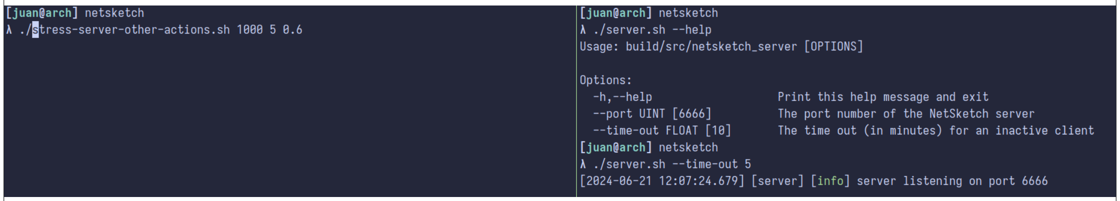
```
[2024-06-21 11:11:55.339] [test_client (juan239)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.339] [test_client (juan239)] [info] Finished...
[2024-06-21 11:11:55.335] [test_client (juan366)] [error] reading failed, reason end o
f file reached
[2024-06-21 11:11:55.339] [test_client (juan366)] [debug] moving average of "handling
payload": 3μs
[2024-06-21 11:11:55.339] [test_client (juan366)] [debug] moving average of "reading i
nput from network": 299973377μs
[2024-06-21 11:11:55.340] [test_client (juan223)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.340] [test_client (juan223)] [info] Finished...
[2024-06-21 11:11:55.340] [test_client (juan201)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.340] [test_client (juan201)] [info] Finished...
[2024-06-21 11:11:55.336] [test_client (juan262)] [debug] moving average of "handling
payload": 3μs
[2024-06-21 11:11:55.341] [test_client (juan262)] [debug] moving average of "reading i
nput from network": 299973352μs
[2024-06-21 11:11:55.342] [test_client (juan209)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.342] [test_client (juan209)] [info] Finished...
[2024-06-21 11:11:55.343] [test_client (juan366)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.343] [test_client (juan366)] [info] Finished...
[2024-06-21 11:11:55.343] [test_client (juan356)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.343] [test_client (juan356)] [info] Finished...
[2024-06-21 11:11:55.344] [test_client (juan262)] [debug] hash of tagged draw vector:
7300014723844814740, size of tagged draw vector 4975
[2024-06-21 11:11:55.344] [test_client (juan262)] [info] Finished...
[juan@arch] netsketch
λ
```

```
reason: connection timed out
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60550 (juan223)] closing connecti
on handler
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60490 (juan356)] closing connecti
on handler
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60632 (juan239)] reading failed,
reason: connection timed out
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60632 (juan239)] closing connecti
on handler
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60686 (juan201)] closing connecti
on handler
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60298 (juan366)] reading failed,
reason: connection timed out
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60298 (juan366)] closing connecti
on handler
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60586 (juan181)] reading failed,
reason: connection timed out
[2024-06-21 11:11:55.335] [server] [info] [127.0.0.1:60586 (juan181)] closing connecti
on handler
^C[2024-06-21 11:12:56.555] [server] [error] poll(): Interrupted system call
[2024-06-21 11:12:56.563] [server] [debug] moving average of "updating all connected c
lients": 10105μs
[2024-06-21 11:12:56.563] [server] [debug] moving average of "updater reading changes"
: 1μs
[2024-06-21 11:12:56.563] [server] [debug] moving average of "handling payload": 3μs
[2024-06-21 11:12:56.563] [server] [debug] moving average of "sending the full list":
38404μs
[2024-06-21 11:12:56.563] [server] [debug] moving average of "handling incoming reques
t": 88μs
[2024-06-21 11:12:56.565] [server] [debug] hash of tagged draw vector: 730001472384481
4740, size of tagged draw vector 4975
[juan@arch] netsketch
λ
```

Figure 25: End phase of a stress test with 1000 test clients (the tagged draw vector hashes are highlighted).

Currently, the project only starts to exhibit issues with correctness when 1000 simultaneous test clients try to connect to the server. However, this only seems to be a networking limitation, since some of the connections (see Figure 24) are dropped. Because of this the test clients do not automatically terminate, since they do not reach their expected draw count. However, after ten minutes of idling the server automatically, disconnects the clients.

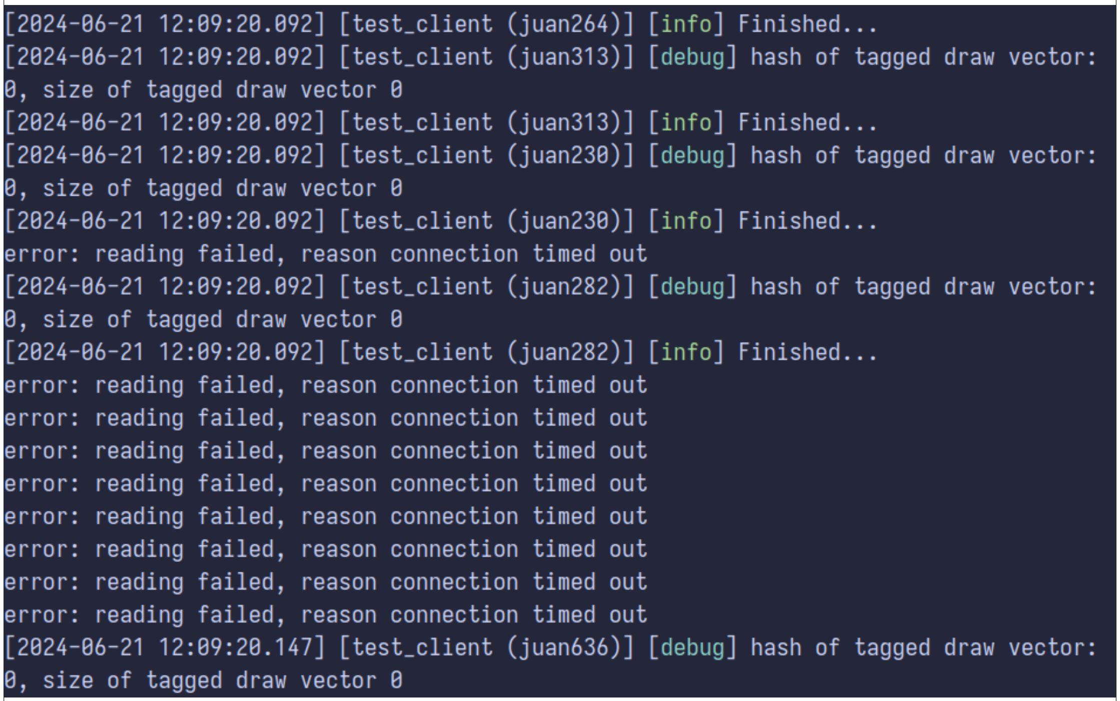
At the end (see Figure 25) it still seems that overall the server's mechanisms for synchronisation are robust. This is because the hashes produced from the tagged draw vectors still seem to be equal to one another.



```
[juan@arch] netsketch
λ ./stress-server-other-actions.sh 1000 5 0.6
```

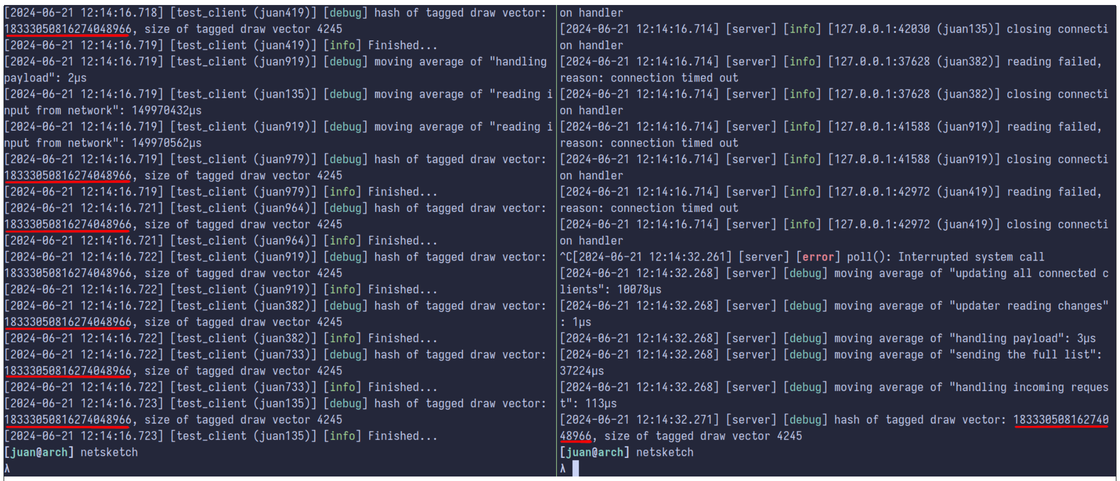
```
[juan@arch] netsketch
λ ./server.sh --help
Usage: build/src/netsketch_server [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  --port UINT [6666]          The port number of the NetSketch server
  --time-out FLOAT [10]       The time out (in minutes) for an inactive client
[juan@arch] netsketch
λ ./server.sh --time-out 5
[2024-06-21 12:07:24.679] [server] [info] server listening on port 6666
```

Figure 26: Start of a stress test with 1000 test clients (with other actions enabled and the tagged vector hashes highlighted).



```
[2024-06-21 12:09:20.092] [test_client (juan264)] [info] Finished...
[2024-06-21 12:09:20.092] [test_client (juan313)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 12:09:20.092] [test_client (juan313)] [info] Finished...
[2024-06-21 12:09:20.092] [test_client (juan230)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 12:09:20.092] [test_client (juan230)] [info] Finished...
error: reading failed, reason connection timed out
[2024-06-21 12:09:20.092] [test_client (juan282)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
[2024-06-21 12:09:20.092] [test_client (juan282)] [info] Finished...
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
error: reading failed, reason connection timed out
[2024-06-21 12:09:20.147] [test_client (juan636)] [debug] hash of tagged draw vector:
0, size of tagged draw vector 0
```

Figure 27: Some test clients again experience timeout errors.



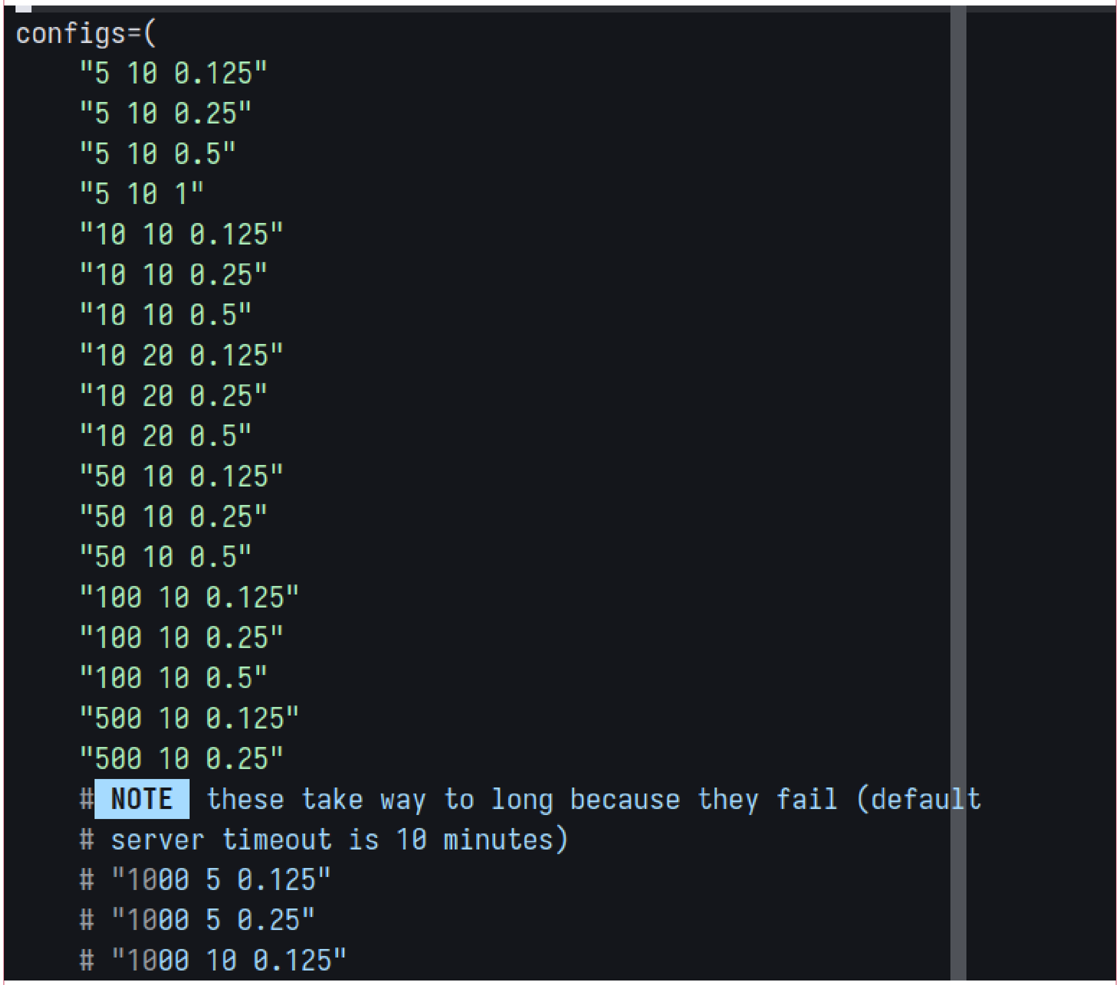
```
[2024-06-21 12:14:16.718] [test_client (juan419)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.719] [test_client (juan419)] [info] Finished...
[2024-06-21 12:14:16.719] [test_client (juan919)] [debug] moving average of "handling
payload": 2µs
[2024-06-21 12:14:16.719] [test_client (juan135)] [debug] moving average of "reading i
nput from network": 149970432µs
[2024-06-21 12:14:16.719] [test_client (juan919)] [debug] moving average of "reading i
nput from network": 149970562µs
[2024-06-21 12:14:16.719] [test_client (juan979)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.719] [test_client (juan979)] [info] Finished...
[2024-06-21 12:14:16.721] [test_client (juan964)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.721] [test_client (juan964)] [info] Finished...
[2024-06-21 12:14:16.722] [test_client (juan919)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.722] [test_client (juan919)] [info] Finished...
[2024-06-21 12:14:16.722] [test_client (juan382)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.722] [test_client (juan382)] [info] Finished...
[2024-06-21 12:14:16.722] [test_client (juan733)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.722] [test_client (juan733)] [info] Finished...
[2024-06-21 12:14:16.723] [test_client (juan135)] [debug] hash of tagged draw vector:
18333050816274048966, size of tagged draw vector 4245
[2024-06-21 12:14:16.723] [test_client (juan135)] [info] Finished...
[juan@arch] netsketch
λ
```

```
on handler
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:42030 (juan135)] closing connecti
on handler
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:37628 (juan382)] reading failed,
reason: connection timed out
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:37628 (juan382)] closing connecti
on handler
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:41588 (juan919)] reading failed,
reason: connection timed out
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:41588 (juan919)] closing connecti
on handler
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:42972 (juan419)] reading failed,
reason: connection timed out
[2024-06-21 12:14:16.714] [server] [info] [127.0.0.1:42972 (juan419)] closing connecti
on handler
^C[2024-06-21 12:14:32.261] [server] [error] poll(): Interrupted system call
[2024-06-21 12:14:32.268] [server] [debug] moving average of "updating all connected c
lients": 10078µs
[2024-06-21 12:14:32.268] [server] [debug] moving average of "updater reading changes"
: 1µs
[2024-06-21 12:14:32.268] [server] [debug] moving average of "handling payload": 3µs
[2024-06-21 12:14:32.268] [server] [debug] moving average of "sending the full list":
37224µs
[2024-06-21 12:14:32.268] [server] [debug] moving average of "handling incoming reques
t": 113µs
[2024-06-21 12:14:32.271] [server] [debug] hash of tagged draw vector: 183330508162740
48966, size of tagged draw vector 4245
[juan@arch] netsketch
λ
```

Figure 28: End phase of a stress test with 1000 test clients (with other actions enabled and highlighted tagged draw vector hashes).

Finally, a test with other actions for the test clients enabled was conducted. A similar conlusion was reached. The server's mechanisms for synchronisation appear to be robust (see Figure 26, Figure 27 and Figure 28).

Testing Performance

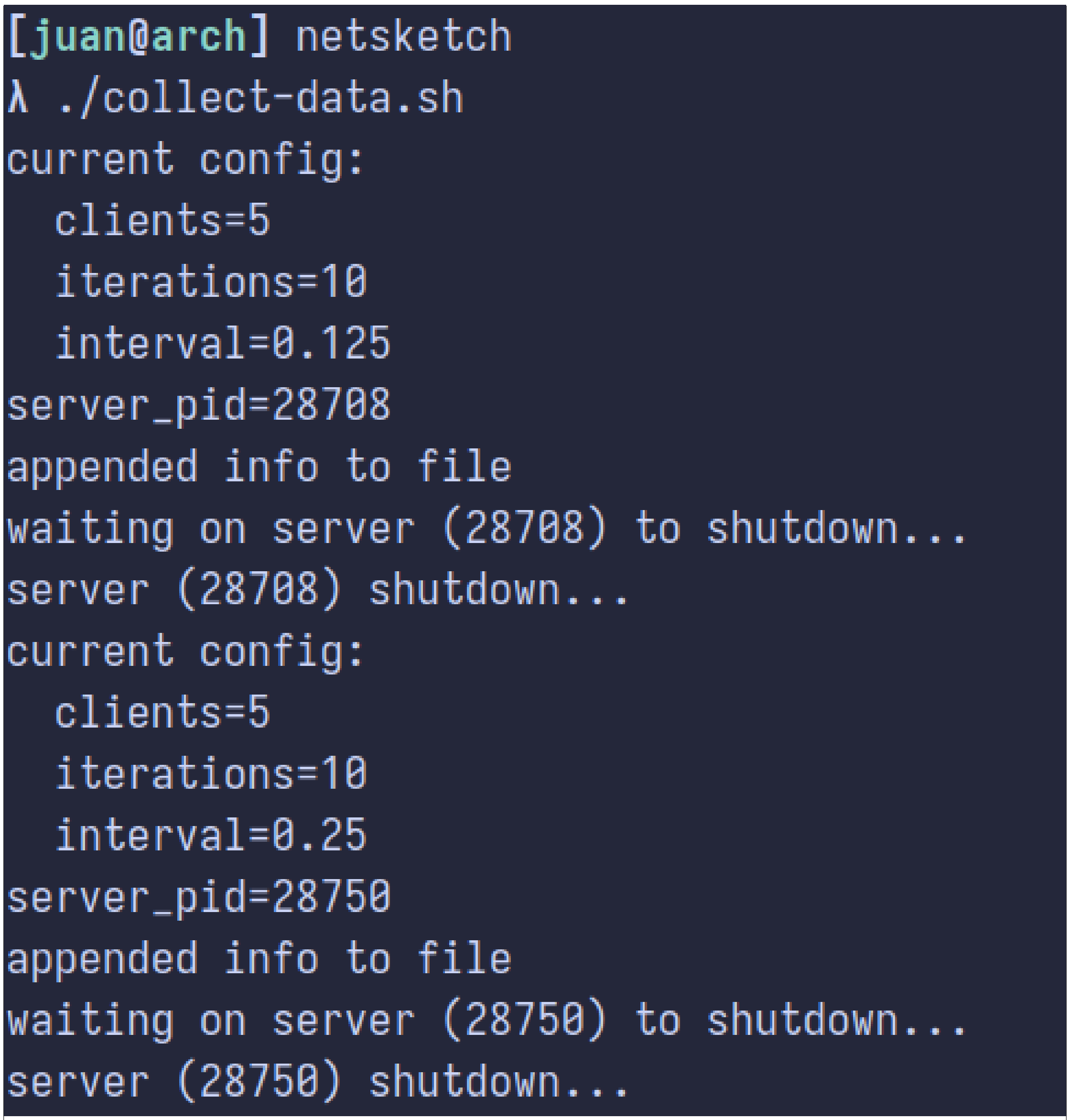


```
configs=(
    "5 10 0.125"
    "5 10 0.25"
    "5 10 0.5"
    "5 10 1"
    "10 10 0.125"
    "10 10 0.25"
    "10 10 0.5"
    "10 20 0.125"
    "10 20 0.25"
    "10 20 0.5"
    "50 10 0.125"
    "50 10 0.25"
    "50 10 0.5"
    "100 10 0.125"
    "100 10 0.25"
    "100 10 0.5"
    "500 10 0.125"
    "500 10 0.25"
    # NOTE these take way to long because they fail (default
    # server timeout is 10 minutes)
    # "1000 5 0.125"
    # "1000 5 0.25"
    # "1000 10 0.125"
```

Figure 29: All the different configurations used for testing and data collection (each string “c i t” is understood as the number of clients (= ‘c’), the number of iterations (= ‘i’) and the time interval between each iteration (= ‘t’).

When testing for performance the `collect-data.sh` script was used. In particular, it contains a selection of configurations (see Figure 29) for which it is known that all the simultaneously generated connections can be handled.

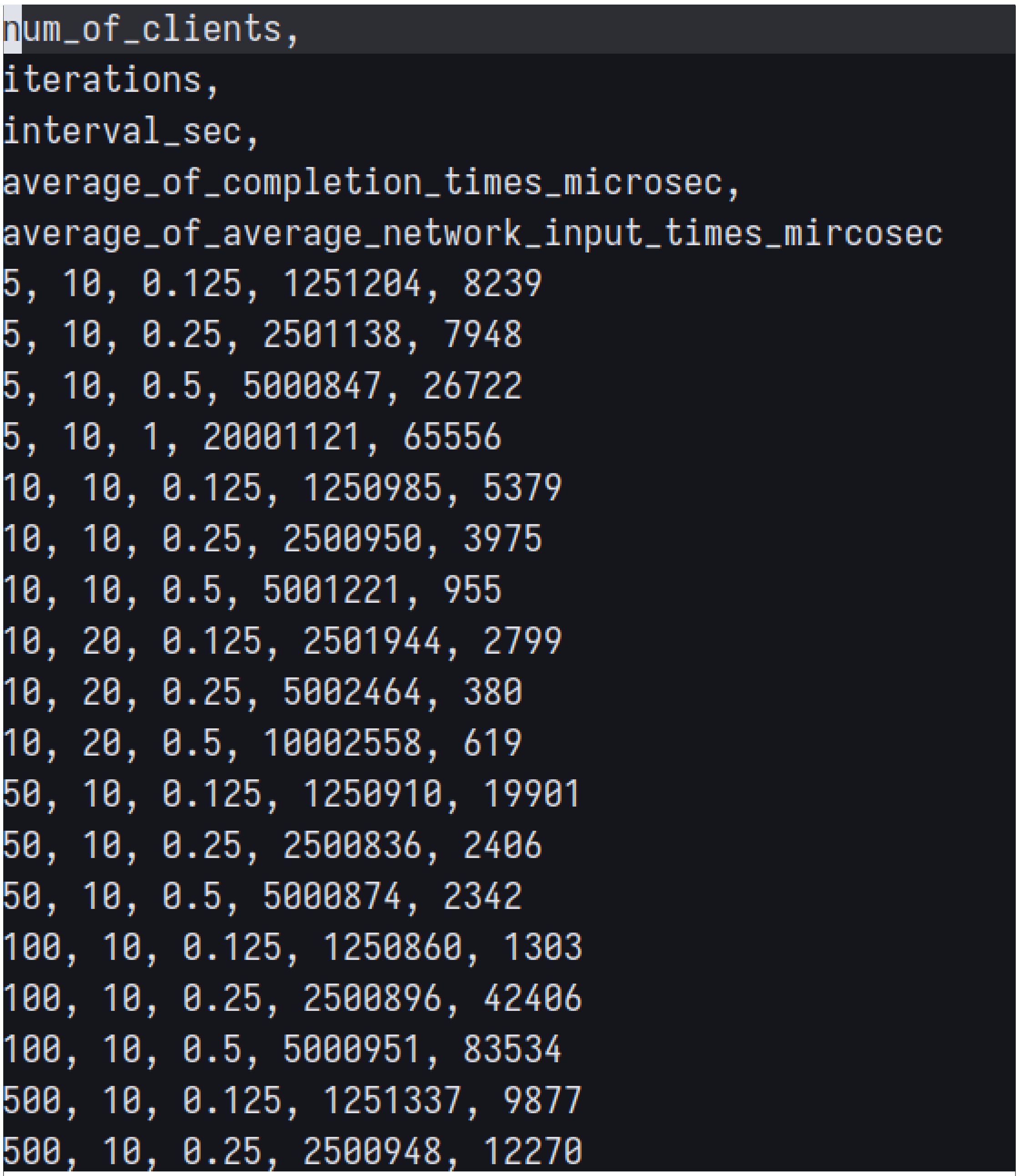
! Such a claim only hold on the machine which the project was developed on.



```
[juan@arch] netsketch
λ ./collect-data.sh
current config:
  clients=5
  iterations=10
  interval=0.125
server_pid=28708
appended info to file
waiting on server (28708) to shutdown...
server (28708) shutdown...
current config:
  clients=5
  iterations=10
  interval=0.25
server_pid=28750
appended info to file
waiting on server (28750) to shutdown...
server (28750) shutdown...
```

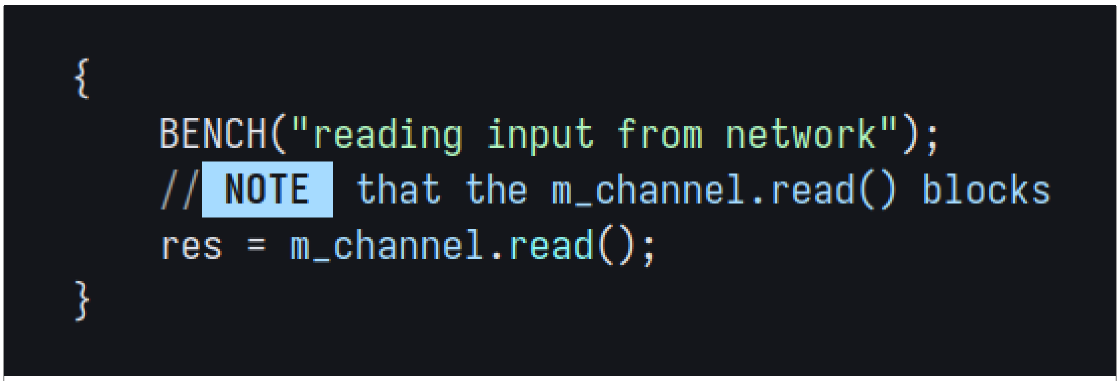
Figure 30: An example of the `collect-data.sh` script running.

! The script `collect-data.sh` connects to any currently running server. If none exist it will manage starting and shutting down instances itself (as it does in Figure 30).



```
num_of_clients,
iterations,
interval_sec,
average_of_completion_times_microsec,
average_of_average_network_input_times_mircosec
5, 10, 0.125, 1251204, 8239
5, 10, 0.25, 2501138, 7948
5, 10, 0.5, 5000847, 26722
5, 10, 1, 20001121, 65556
10, 10, 0.125, 1250985, 5379
10, 10, 0.25, 2500950, 3975
10, 10, 0.5, 5001221, 955
10, 20, 0.125, 2501944, 2799
10, 20, 0.25, 5002464, 380
10, 20, 0.5, 10002558, 619
50, 10, 0.125, 1250910, 19901
50, 10, 0.25, 2500836, 2406
50, 10, 0.5, 5000874, 2342
100, 10, 0.125, 1250860, 1303
100, 10, 0.25, 2500896, 42406
100, 10, 0.5, 5000951, 83534
500, 10, 0.125, 1251337, 9877
500, 10, 0.25, 2500948, 12270
```

Figure 31: The collected data from running `collect-data.sh` in the produces `data.csv` file.



```
{
    BENCH("reading input from network");
    // NOTE that the m_channel.read() blocks
    res = m_channel.read();
}
```

Figure 32: The `"reading input from network"` benchmark in `src/test_client/reader.cpp` at line 45.

In particular, looking at the outcome of the script (see Figure 31), the time it takes for each test client in the scope of a specific configuration is very much linked to the specified interval and the number of iterations.

In fact, if for each configuration the calculation $\texttt{iterations} * \texttt{interval\_sec} * 1000000$ is performed it is roughly equal to the completion time for each test client (in the context of that configuration).

The average amount of time it takes to read a response is quite inconsistent across the board. It seems to be really dependent on the load of current machine hosting the server. However, there does seem a slight up tick for longer intervals.

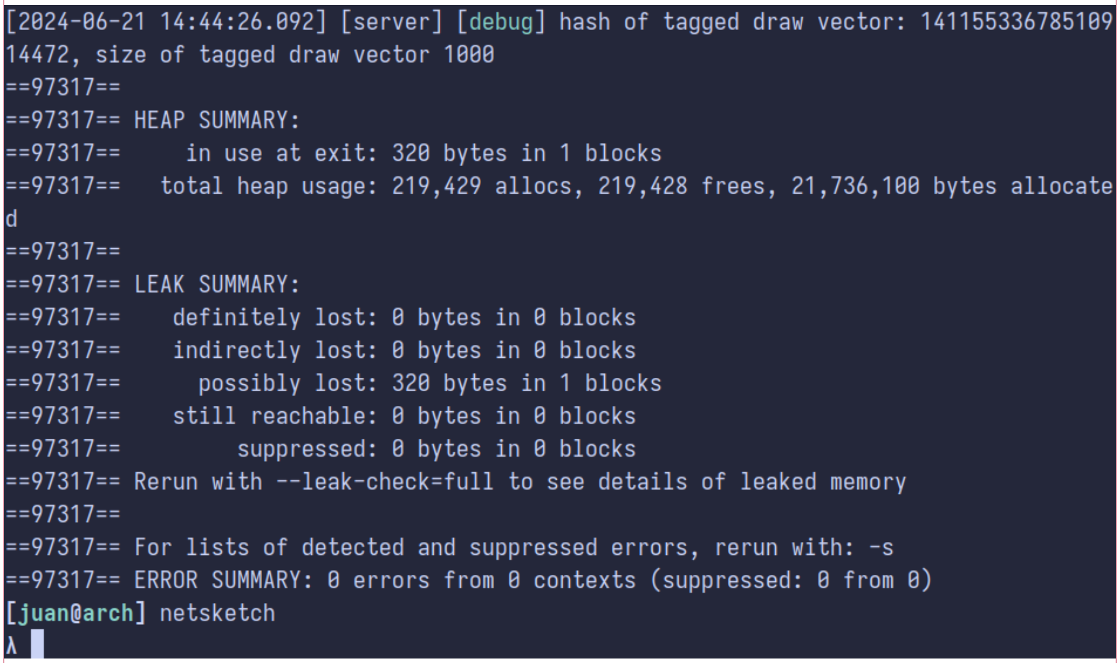
From the above observations one can conclude the following:

The amount of time it takes for a client (call it C) to terminate is equivalent to the amount of time it would take for all the clients (including C) to send all the draw commands sequentially, to C.

This is indeed what was warned in Subsection 6.3. The responsiveness of the application is indeed dependent on the single-threaded performance of the server. However, the amount of time the server spends doing ‘other things’, outside of the ideal scenario (where the time would be exactly that computed using the above formula), is minimal. Hence, this indicates a overall reasonable performance.

! One has to keep in mind that all these test were performed locally, where the performance penalty imposed by network congestion is close to zero.

8.2 | Running `valgrind` on the Server



```
[2024-06-21 14:44:26.092] [server] [debug] hash of tagged draw vector: 141155336785109
14472, size of tagged draw vector 1000
==97317==
==97317== HEAP SUMMARY:
==97317==     in use at exit: 320 bytes in 1 blocks
==97317==   total heap usage: 219,429 allocs, 219,428 frees, 21,736,100 bytes allocate
d
==97317==
==97317== LEAK SUMMARY:
==97317==    definitely lost: 0 bytes in 0 blocks
==97317==    indirectly lost: 0 bytes in 0 blocks
==97317==      possibly lost: 320 bytes in 1 blocks
==97317==    still reachable: 0 bytes in 0 blocks
==97317==         suppressed: 0 bytes in 0 blocks
==97317== Rerun with --leak-check=full to see details of leaked memory
==97317==
==97317== For lists of detected and suppressed errors, rerun with: -s
==97317== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[juan@arch] netsketch
λ
```

Figure 33: The result of running `./stress-server 100 10 0.25` with the server being instrumented with `valgrind` (the single lost allocation has been clarified in Section 4.4).

Finally, the server was tested with `valgrind` to ensure that over time the server does not consume more and more memory, which could lead to the system failing (see Figure 33).

8.3 | Testing the Client

Performance Testing

To test the actual client, CLion was used. CLion provides facilities for generating flamegraphs and the client was tested using these facilities.

ATTRIB. The clear summary about flamegraphs provided by Brendan Gregg on his website `https://www.brendangregg.com/flamegraphs.html` was consulted.

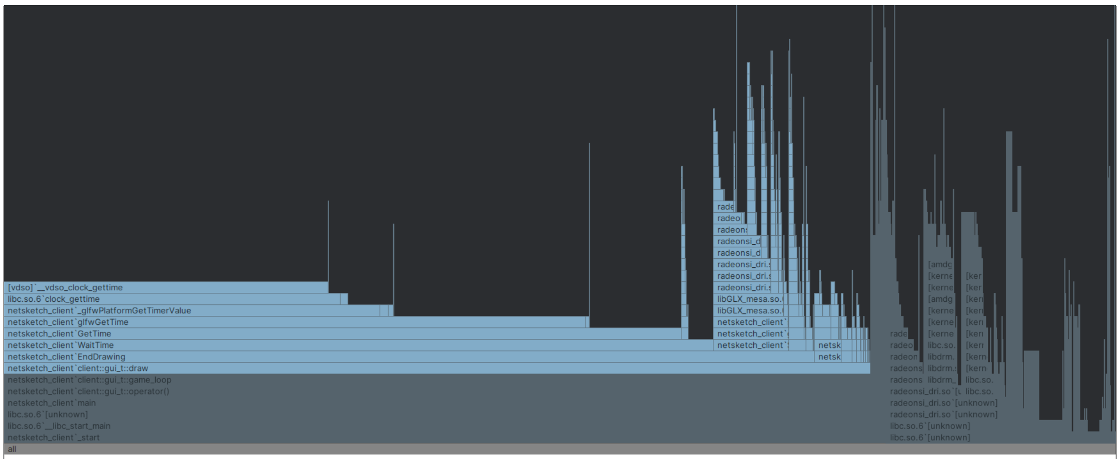


Figure 34: A flamegraph of the client running captured by CLion.

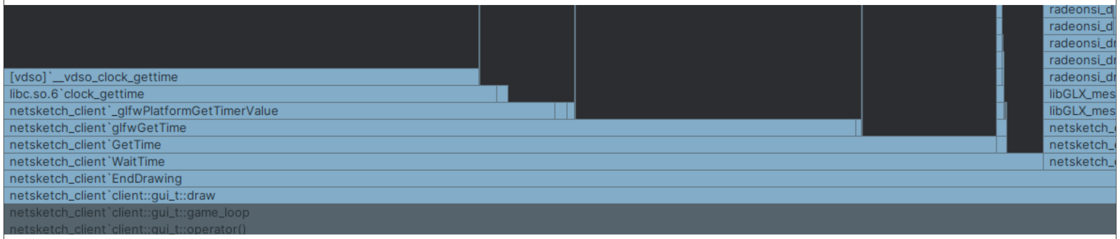
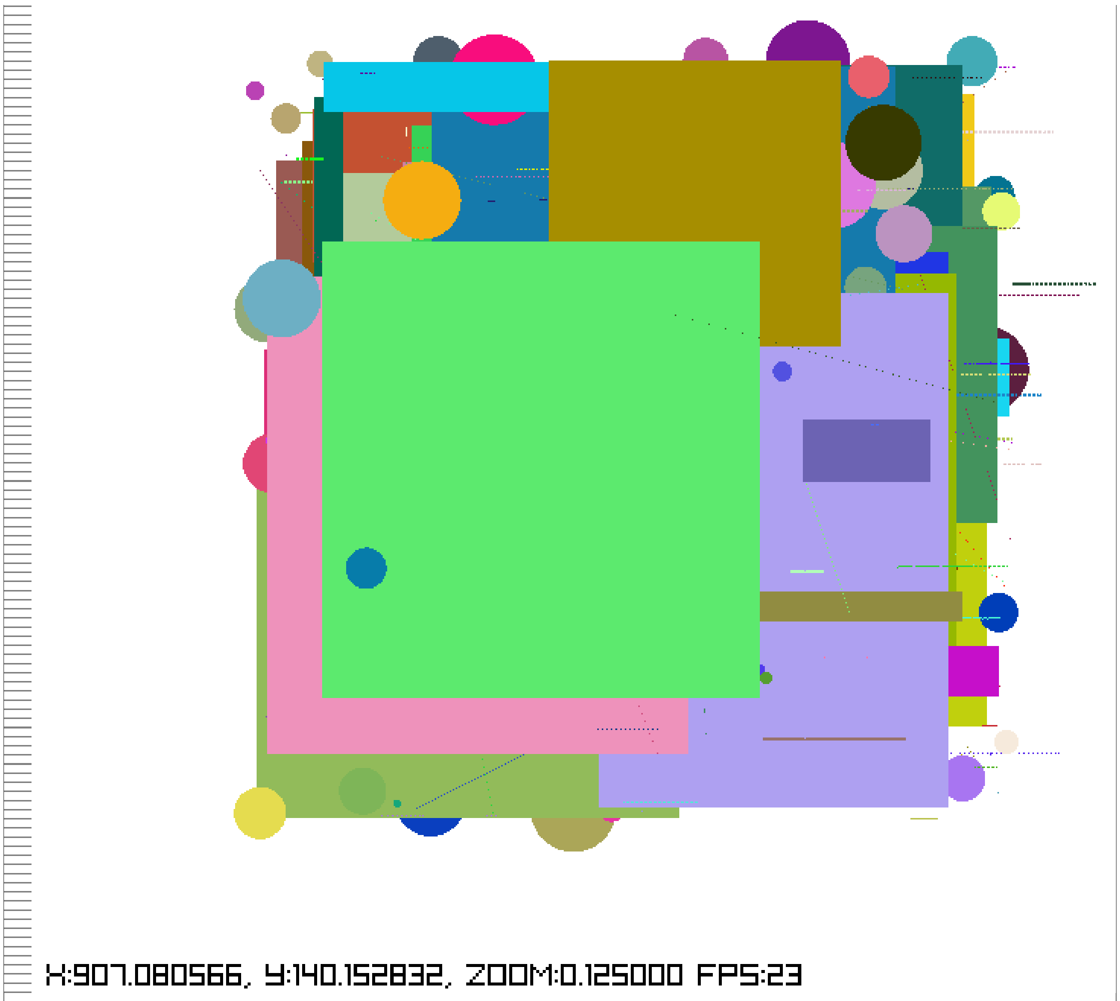


Figure 35: The same flamegraph as Figure 34 but zoomed (for clarity).

As can be seen from Figure 35, the hottest path i.e. the path which has the most calls is the draw function in the client. This means that the largest bottle neck in the client is actually the drawing.



X:907.080566, Y:140.152832, ZOOM:0.125000 FPS:23

Figure 36: A screenshot of a single client connected to the server with many draw calls (notice that the FPS counter is sitting at around 23).

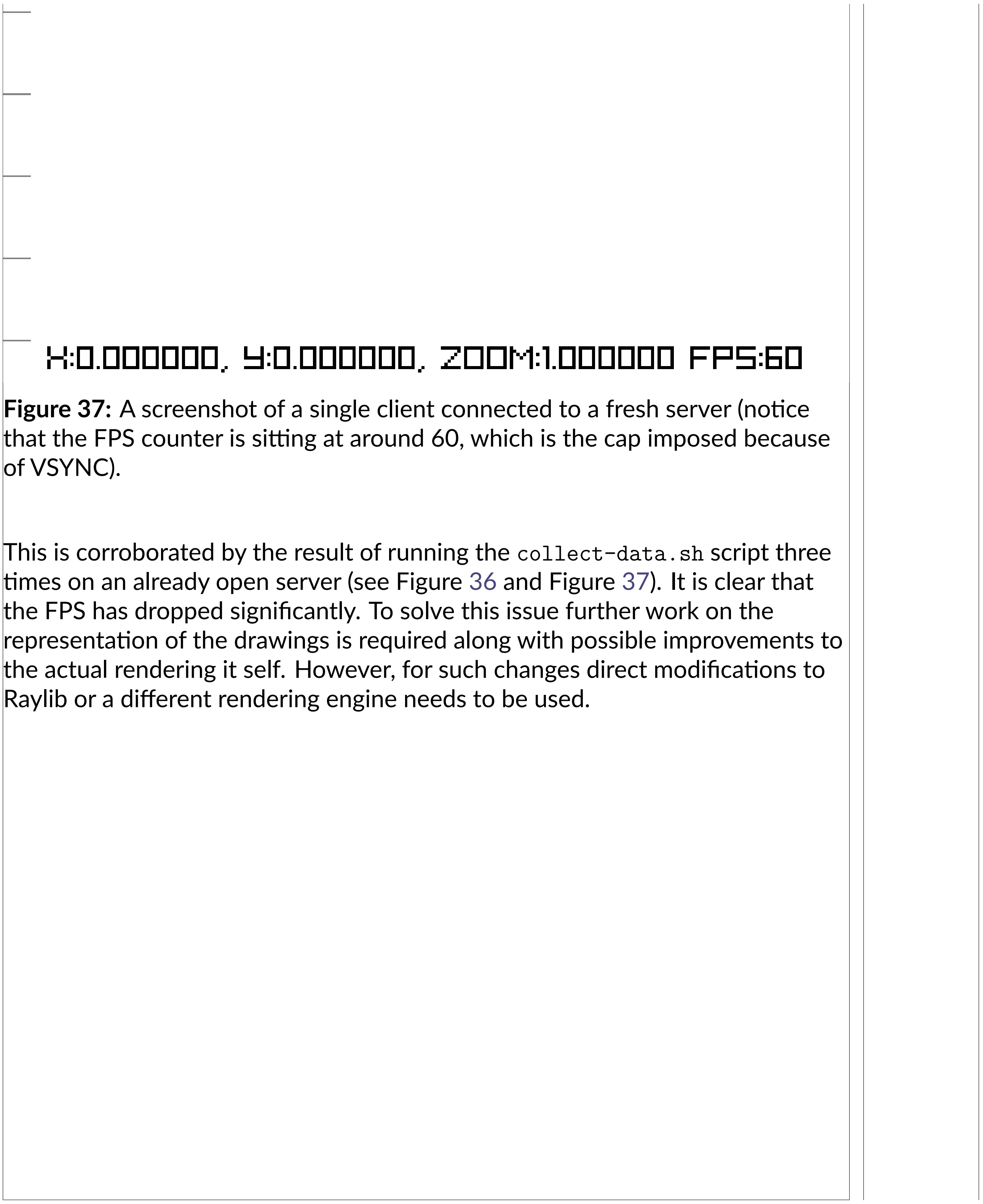
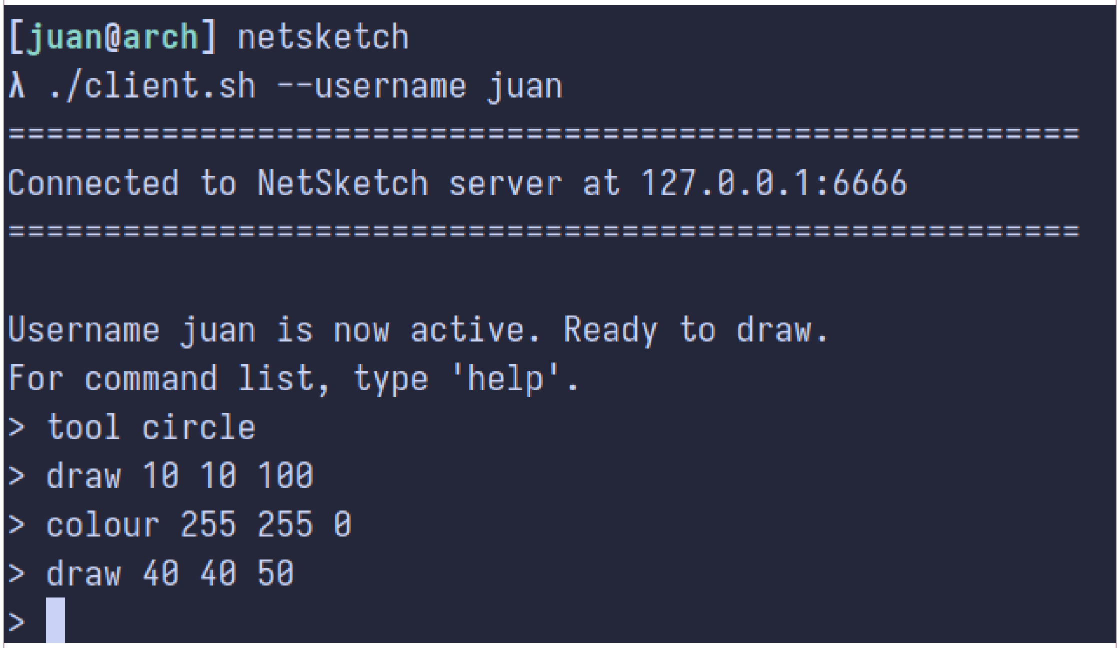

X:0.000000, Y:0.000000, ZOOM:1.000000 FPS:60

Figure 37: A screenshot of a single client connected to a fresh server (notice that the FPS counter is sitting at around 60, which is the cap imposed because of VSYNC).

This is corroborated by the result of running the `collect-data.sh` script three times on an already open server (see Figure 36 and Figure 37). It is clear that the FPS has dropped significantly. To solve this issue further work on the representation of the drawings is required along with possible improvements to the actual rendering it self. However, for such changes direct modifications to Raylib or a different rendering engine needs to be used.

Input Testing



```
[juan@arch] netsketch
λ ./client.sh --username juan
=========================================================
Connected to NetSketch server at 127.0.0.1:6666
=========================================================

Username juan is now active. Ready to draw.
For command list, type 'help'.
> tool circle
> draw 10 10 100
> colour 255 255 0
> draw 40 40 50
>
```

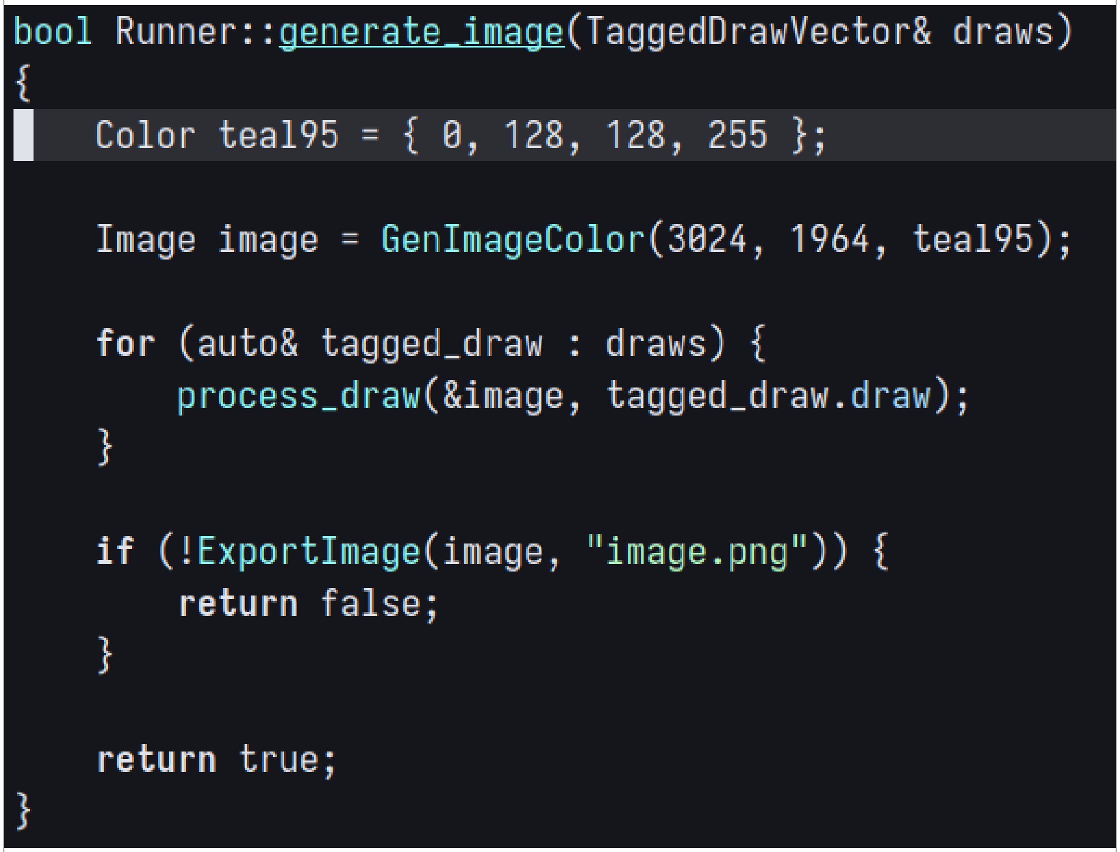
Figure 38: A screenshot of some basic input into a client.

A very rudimentary approach to testing the input mechanism was used. In particular, the command line interface was manually tested to ensure that at its most basic the inputs were properly parsed.

9 | Recording

For a more thorough walk-through of how the interface can be used and how the clients and the server behave please have a look at the video presentation which is being hosted at `https://drive.google.com/file/d/1VMXR4ZOs5iDvXcroMTLn6ddb5enplOTF/view?usp=drive_link`.

10 | **Bonus**: The Exporter



```cpp
bool Runner::generate_image(TaggedDrawVector& draws)
{
    Color teal95 = { 0, 128, 128, 255 };

    Image image = GenImageColor(3024, 1964, teal95);

    for (auto& tagged_draw : draws) {
        process_draw(&image, tagged_draw.draw);
    }

    if (!ExportImage(image, "image.png")) {
        return false;
    }

    return true;
}
```

Figure 39: A screenshot of the generate image method in `src/exporter/runner.cpp` at line 244.

The exporter is a simple binary which connects to an already running server. It then uses Raylib's built in software renderer to generate an image of the server's local tagged draw vector with a specific background colour.

Of course, later on this can be directly embedded into the client, using the GPU buffer instead of software rendering. But for now it is sufficient for some cool backgrounds.

Figure 40: A cool background generated by using the exporter.