Real Time Embedded Systems – Final Project (2025)

Name: Chouklis Athanasios // AEM: 10396 // GitHub Repository

The goal of this project is to write a program that collects and processes cryptocurrency trade data from OKX in real time, using the platform's public WebSocket channels.

As the program is meant to run on low-power devices (in this case, a Raspberry Pi Zero 2 W) for long periods of time, it must make efficient use of system resources. Consequently, the language chosen was C and care was taken to ensure multi-threading is safely utilized, via the **pthread.h** library. The **libwebsockets.h** library was used to interface with OKX's WebSocket API. The program is comprised of 4 threads:

1) The main thread takes all the steps necessary to initiate the program's working data (file pointers, FIFO queue for incoming trade data, etc.) before starting the WebSocket connection process. It sets libwebsockets' protocols struct, creates the context and sets all necessary parameters (address, port, path, SSL usage, retry/idle policy, etc.) before initiating the connection using **lws_sul_schedule()**. It then creates the other 3 threads; structs are used to pass bulk data to them, as **pthread_create()** can only pass one parameter to the thread it creates. It then sits idle until it receives either of the SIGINT or SIGTERM signals, at which point it begins the cleanup and shutdown process by setting certain flags, broadcasting pthread conditions and using **pthread_join()** on each thread, as well as destroying the libwebsockets context, closing file pointers, etc.

2) The "producer" thread receives the libwebsockets context (sent at creation by the main thread) and uses it to continuously call **lws_service()** until a termination flag is set. When a WebSocket event occurs, **lws_service()** calls **callback_okx()** as defined in the protocols struct. This callback function responds to the event accordingly:

- Upon establishing the connection, the function subscribes to the 8 required symbols using **lws_callback_on_writable()**

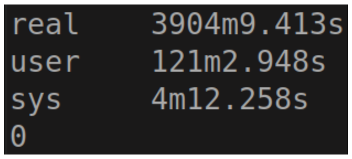
- When a message is received, the function checks what type it is. "Subscribe" messages are printed to console, "Unsubscribe" messages prompt another **lws_callback_on_writable()** call to subscribe again and "Data" messages result in the addition of new trade data to the FIFO queue. As the data is received in JSON format, its parsing is achieved via the **jansson.h** library

- When a "Connection Closed" or "Connection Error" is received, reconnection is attempted indefinitely until it succeeds, using **lws_retry_sul_schedule_retry_wsi()**

3) The “consumer” thread removes the trade data from the FIFO queue and saves it to its respective file (each symbol has its own file). It also increments 3 variables: numTrades, totalSize and sumPrices, to keep track of the symbol’s number of trades conducted, total size traded and sum of all trade prices paid in the last minute. The trio of numTrades, totalSize and sumPrices, along with all the file pointers, comprise a struct dubbed “database” which is given its own pthread mutex (contained within it) to prevent race conditions, similar to the FIFO queue’s mutex.

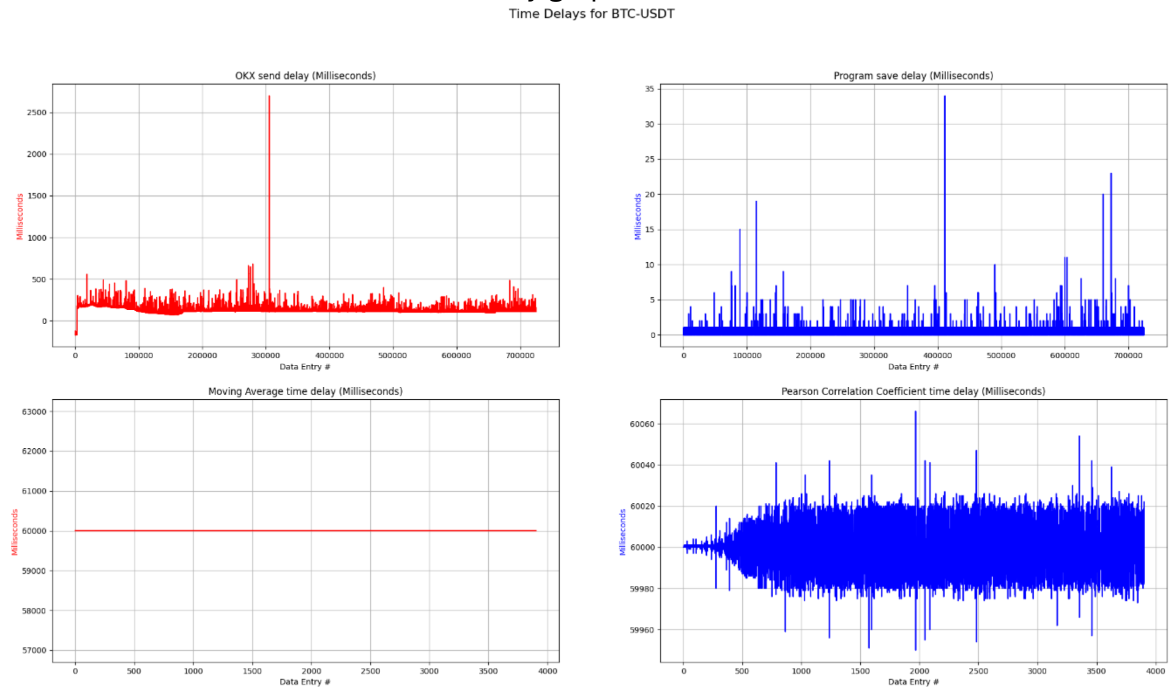
4) The ”everyInterval” thread wakes once every minute to save the data in the “database” struct to its own arrays; it then resets the struct’s data to 0 in order to prepare the recording of the next minute’s trade data. For each symbol, it eventually (after 15 minutes) accumulates and keeps the “results” of the most recent 15 minutes in a circular buffer whose oldest value gets replaced every minute. This data is then on-the-spot used to calculate and save (to files) the required moving average and pearson correlation coefficient values of the minute. The coefficient calculation takes into account the program’s entire runtime, parsing the moving average files value by value, one by one, to find the absolute best correlation and the timestamp it corresponds to.

The implementation of the FIFO queue was taken from Andrae Muys’ 1997 Producer/Consumer Demo, with slight modification. The program was run using *time ./rtes-okx-raspi; echo $?* to get the information shown below (full runtime log on GitHub):

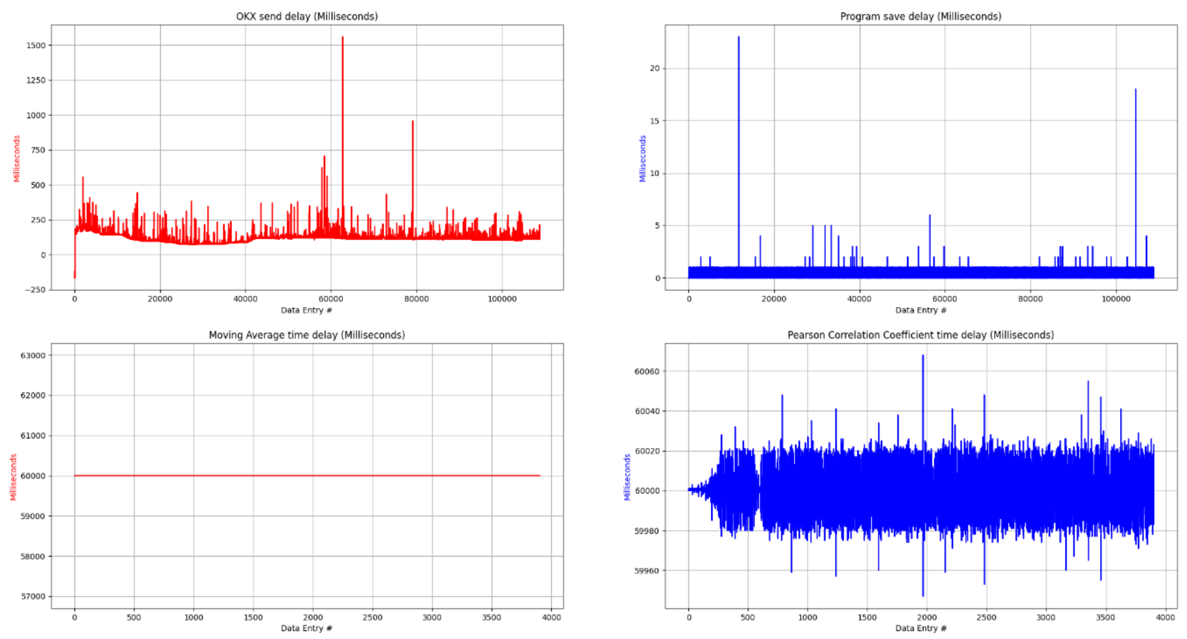


```
real	3904m9.413s
user	121m2.948s
sys	4m12.258s
0
```

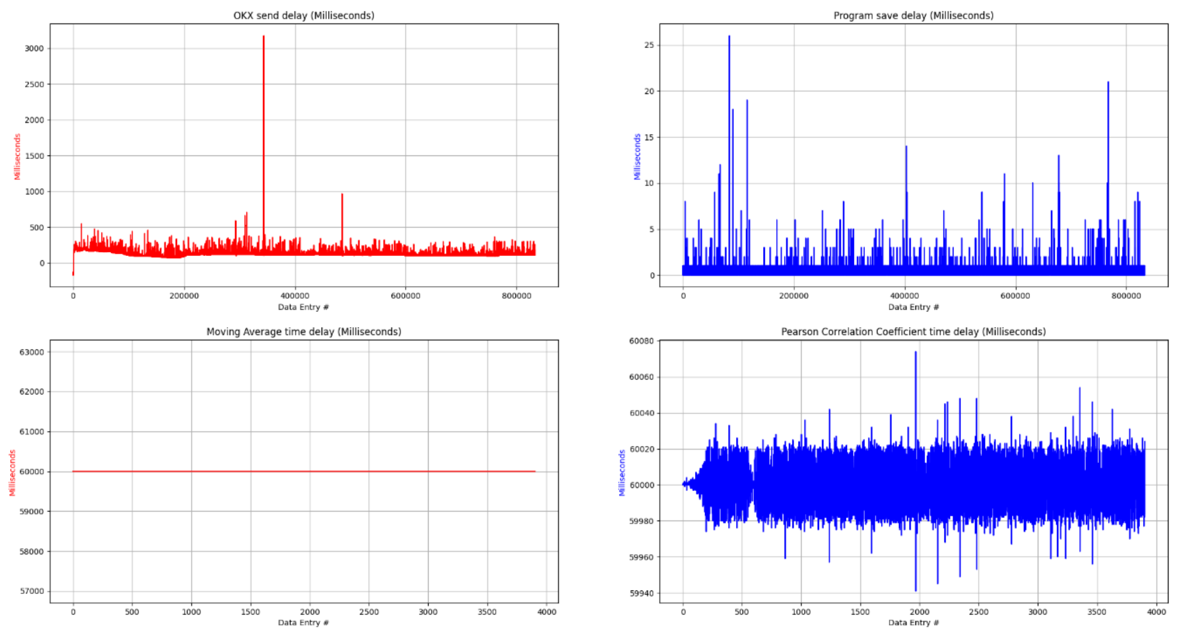
The total runtime was 3904 minutes and 9.413 seconds, or approximately **65 hours** without issue. The sum of “user” and “sys” comprises the total amount of time the CPU was busy for; dividing it by “real” gives the approximate result of **0.032**, which is the fraction of the total runtime for which the CPU was busy. This means that the CPU was busy approximately **3.2%** of the time, and therefore it was idle **96.8%** of the time. “0” is the exit status. Below are the time delay graphs that resulted from this run:



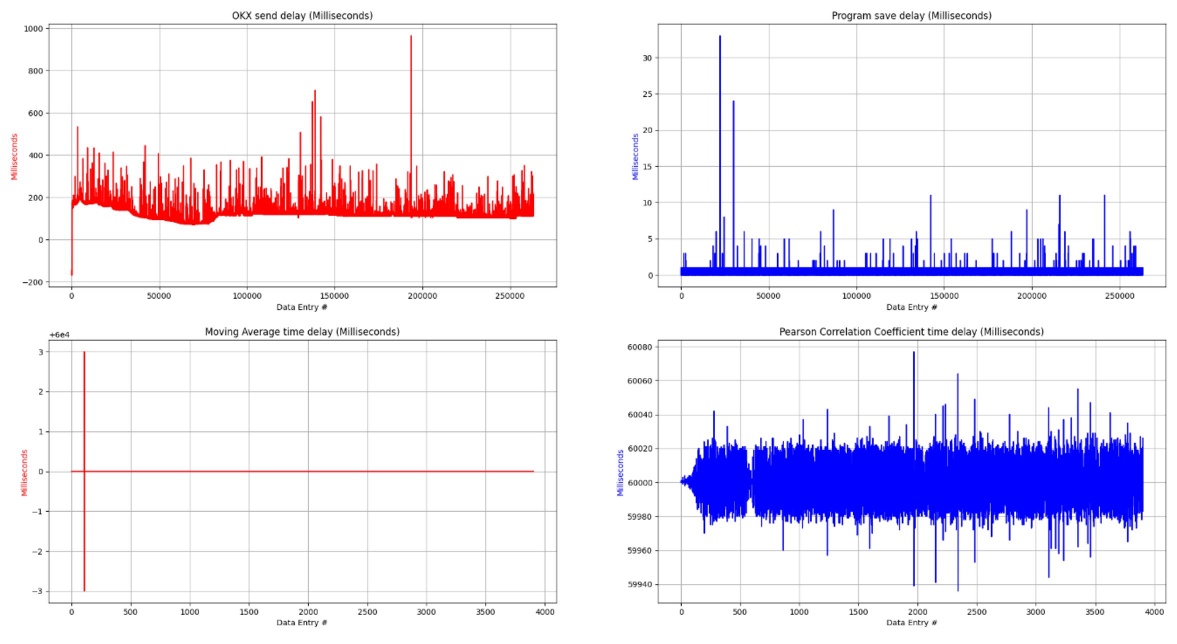
Time Delays for ADA-USDT



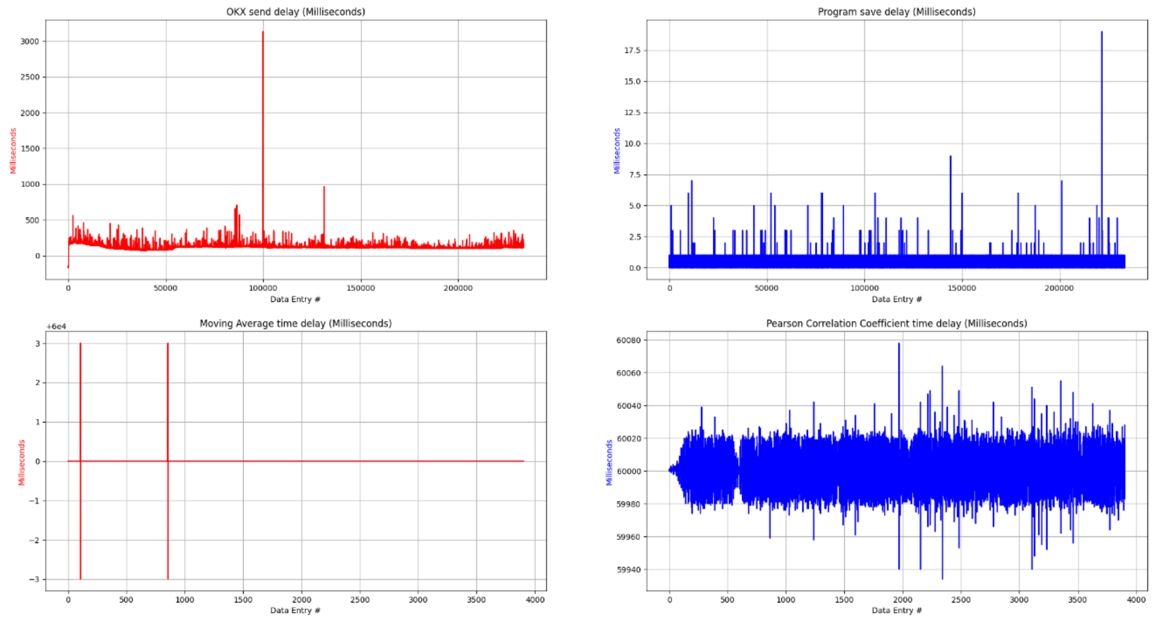
Time Delays for ETH-USDT



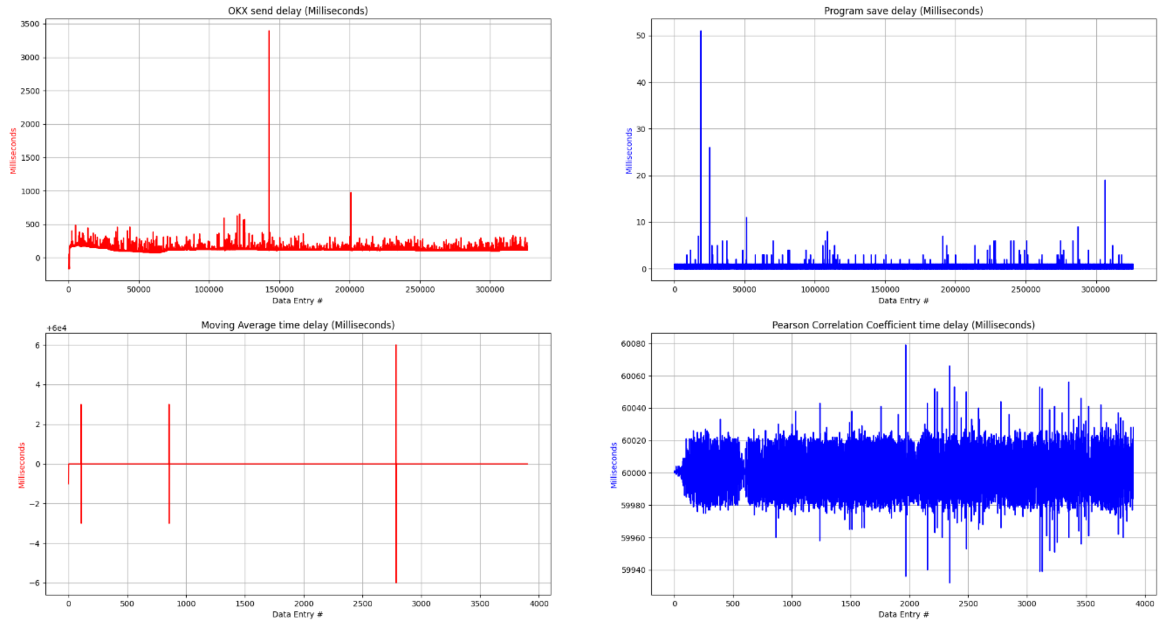
Time Delays for DOGE-USDT



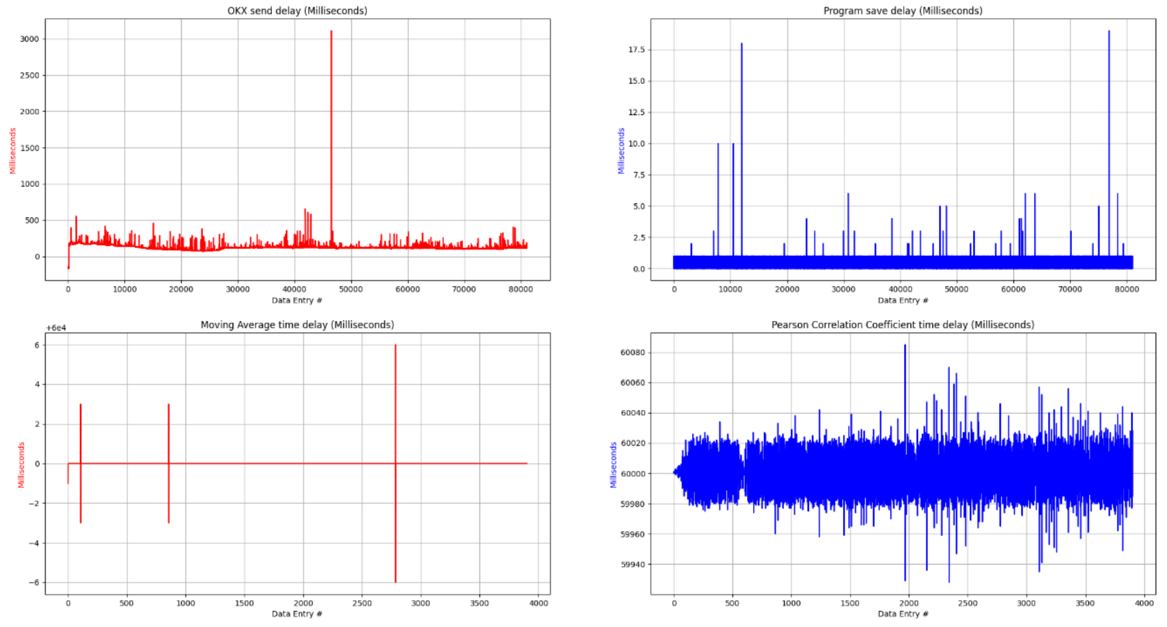
Time Delays for XRP-USDT

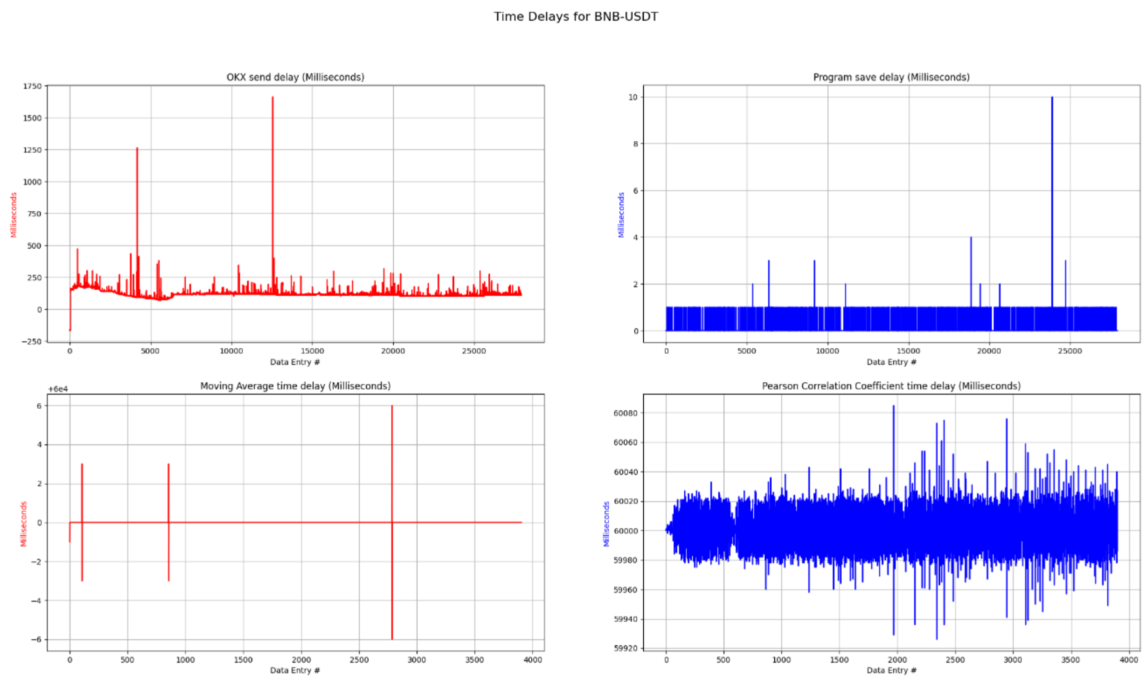


Time Delays for SOL-USDT



Time Delays for LTC-USDT





When it comes to the trade data itself, most of the time delay is on the part of OKX sending the data, ranging in the hundreds of milliseconds, whereas the program itself takes only a few tens of milliseconds to save it (mostly below 10 ms). This makes sense as sending data over the internet should take longer than waiting for mutexes to unlock and writing to a file.

When it comes to the per-minute processing and saving delays (calculated as “next” minus “previous”, which should be 1 minute apart), the moving averages are nearly always exactly 60000 ms apart (as expected), whereas the pearson correlation coefficients take a few tens of milliseconds to be saved. This is also expected, because the maximum coefficient is searched for among all previous moving average values, which involves file reading – a “heavy” investment in terms of time, compared to working directly with memory.

Nevertheless “drift” is avoided by incrementing the “everyInterval” thread’s **pthread_cond_timedwait()** timer by 60 seconds *immediately* upon each activation/wakeup, *before* any of the processing takes place, ensuring the “clock” will “ring” exactly 1 minute later, every time, without being impacted by the processing’s runtime. As such, no entries are “missed”: 3904 minutes results in 3904 entries. The only way “drift” could occur here is if the files got so large that parsing them took more than a whole minute – in which case this program will have indeed reached its limit. The *timespec* struct (incremented at the beginning of each wakeup) used for **pthread_cond_timedwait()** (which implements the timer) as well as the attached pthread condition are implemented using CLOCK_MONOTONIC, which is independent of network time changes, unlike CLOCK_REALTIME. In this way, it is more reliable, as the program’s internal timing cannot be disrupted by outside changes.

Signal handlers are NOT used as they are not thread safe according to **pthread.h** documentation. The main thread sleeps until SIGINT/SIGTERM appears using **sigwait()**. The other threads are barred from responding to any signal by means of signal masking.