COVID-19 Contact Tracing

Gitopoulos Giorgos, 9344

Real Time Embedded Systems @ ECE AUTh, August 2021

1. Introduction

COVID-19 Contact Tracing is the process of identifying all people that a COVID-19 patient has come in contact with in the last two weeks. This project deals with the development of an application that simulates contact tracing, collecting virtual **Bluetooth MAC Addresses** of the contacts of the user and notifying his close contacts in case of a positive COVID-19 test. The application was tested on a **Raspberry Pi 4 Model B - 2GB RAM**, running *Raspberry Pi OS Lite* and working as an **embedded computer**. The source code of the project can be found **here**.

2. Implementation

2.1. Work Distribution in Threads

The code of the application was written in *C* and *PThreads* were used in order to achieve **real-time behavior** through **multi-threading**. So, 4 threads are created and the total work is split into them.

2.1.1. *Timer Thread*

The first thread, is the **timer thread**, that is responsible to update the current-time variable, based on the moment that the program started running. After testing, it was decided to make the thread sleep for 100 μs in each iteration in order to free some resources, as it was noticed that it does not affect the real-time behavior of the application. The *gettimeofday()* function was used for time counting.

2.1.2. *COVID Test Thread*

The second thread, takes over the **virtual COVID testing** of the user, every *TEST_TIME* (predefined number) seconds. Firstly, the thread function creates 3 binary files for storing data (as it will be described in 2.2) and right after, an infinite loop starts, where in each iteration the thread sleeps for *TEST_TIME* seconds and then performs a virtual test. The COVID test is simulated through *test_covid()* function, which returns *true* with probability *POS_TEST_PROB* %. If the test gives positive result, the *upload_contacts()* function is called. This function, writes the close contacts (will be defined in 2.1.4) of the user in a binary file and they are supposed to be uploaded in a server in order to notify the possibly infected people. Also, a break condition is placed for the purposes of the project in order to terminate the loop after *END_TIME* seconds.

2.1.3. *Recent Contacts Thread*

Recent contacts thread is responsible to detect the contacts of the user and store them in a data structure for an amount of time. First of all, the function *bt_near_me()* generates and returns a *contact struct*, which includes the *contact timestamp* and a random *contact MAC address* (from a predefined number of addresses). So, every *SEARCH_TIME* seconds, this function is called and the new contact is added in a static-size *FIFO queue*. Also, before adding a new contact, the oldest element of the queue is checked and if it has remained in the queue for more than *RECENT_DEL_TIME* seconds, it is popped out, as it is no longer considered as a recent contact.

As previously, an infinite loop is used with *usleep()* function and the same break condition.
The maximum possible number of recent contacts in the queue in a specific moment was computed as $RECENT\_DEL\_TIME / SEARCH\_TIME + 1$, so that is the size of the static queue. It was decided for the queue to be modified by only one thread, in order to avoid **mutexes**, so both the adding and the popping are performed by the same thread.

*2.1.4. **Close Contacts Thread***

The last thread is the **close contacts thread**, which detects and updates the close contacts of the user. The process takes place using the *recent contacts queue*, as every time a new recent contact is added, the current thread searches in the queue for the same *MAC address* between *MAX_CLOSE_CONTACT_TIME* seconds and *MIN_CLOSE_CONTACT_TIME* seconds before the timestamp of the last added recent contact. If the same *MAC address* is found in this time period, the contact is considered as a close contact and it is added in the *close contacts queue* with the most recent timestamp of the contact.
The *close contacts queue* is a static *FIFO queue*, as the *recent contacts queue*. The thread performs an infinite loop, starting with a 0.5 seconds sleep in order to free unecessary resources, as there is no need for the operation to be executed continuously. After sleeping time, there is a break condition (after *END_TIME* seconds the program terminates) and a check in the oldest element of the queue, as it has to be popped out *CLOSE_DEL_TIME* seconds after its timestamp. The loop iteration goes on with the close contact search that was described. Moreover, for the same reason as previously, adding and popping are performed by a single thread.

2.2. File Writing & Reading

As mentioned earlier, *upload_contacts()* function is called in case of a positive COVID test and writes the close contacts of the patient in binary files. Three files are created: the first one (*close_contacts.bin*) contains the *contact structs* (8 bytes for each *unsigned long long - MAC address* and 8 bytes for each *double - timestamp*) of all the uploaded contacts successively. The second file (*upload_times.bin*) consists of the timestamps (*double* - 8 bytes per element) of all the uploads, while the last one (*contacts_nums.bin*) contains the number (*int* - 4 bytes per element) of contacts that was uploaded in every upload moment.
In addition, *bt_near_me()* function writes the timestamp of every Bluetooth search in the *bt_search_times.bin* file in order to analyze the real-time behavior of the application (3.2).
Some reading functions were implemented to read the mentioned binary files and can be executed running the *reader/reader.c* file, which prints the content of the files of *out* directory.

3. Testing & Statistical Analysis

The program, as already mentioned, was executed on a *Raspberry Pi 4 Model B - 2GB RAM*, running *Raspberry Pi OS Lite*. The *C* file was **cross-compiled** in a *x86 machine* for *ARM* processor and the executable file was transferred and run in *Raspberry Pi*.

3.1. Parameters Values & Results

The target was to test the application for 30 days, but it was more feasible to speed up the operation by 100 times. So, the defined parameters of the test were the following (the parameters that were not explained earlier are explained here):

- *SEARCH_TIME* = 10 seconds
- *TEST_TIME* = 4 hours / 100 = 144 seconds
- *RECENT_DEL_TIME* = 20 minutes / 100 = 12 seconds
- *NUM_OF_ADDRESSES* = 30 (number of possible *MAC addresses*)
- *MIN_CLOSE_CONTACT_TIME* = 4 minutes / 100 = 2.4 seconds
- *MAX_CLOSE_CONTACT_TIME* = 20 minutes / 100 = 12 seconds
- *POS_TEST_PROB* = 10 %
- *END_TIME* = 30 days / 100 = 25920 seconds
- *CLOSE_DEL_TIME* = 14 days / 100 = 12096 seconds

Note that the values of *NUM_OF_ADDRESSES* and *POS_TEST_PROB* were selected on purpose low and high respectively, in order increase the density of the results in a restricted time period. The results of the test can be found in *out/out.txt* directory of the **repository** of the project.

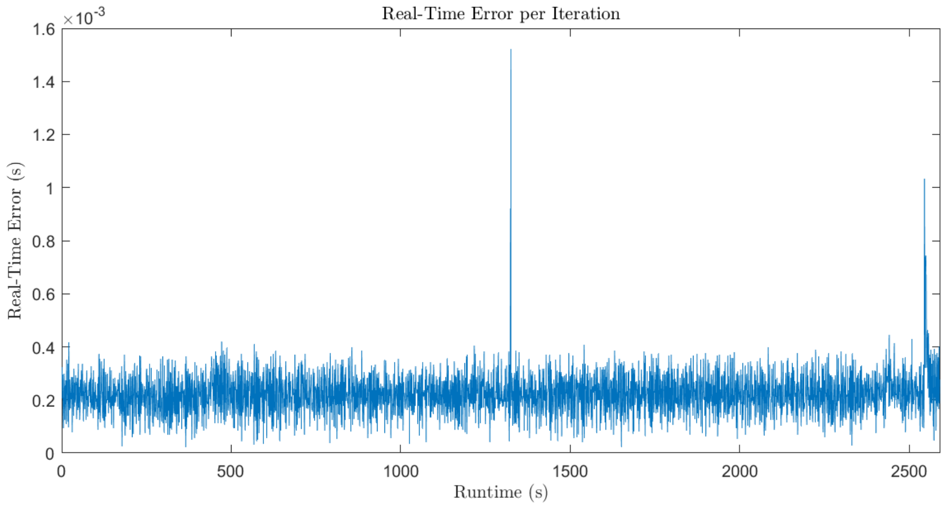
3.2. Time Error

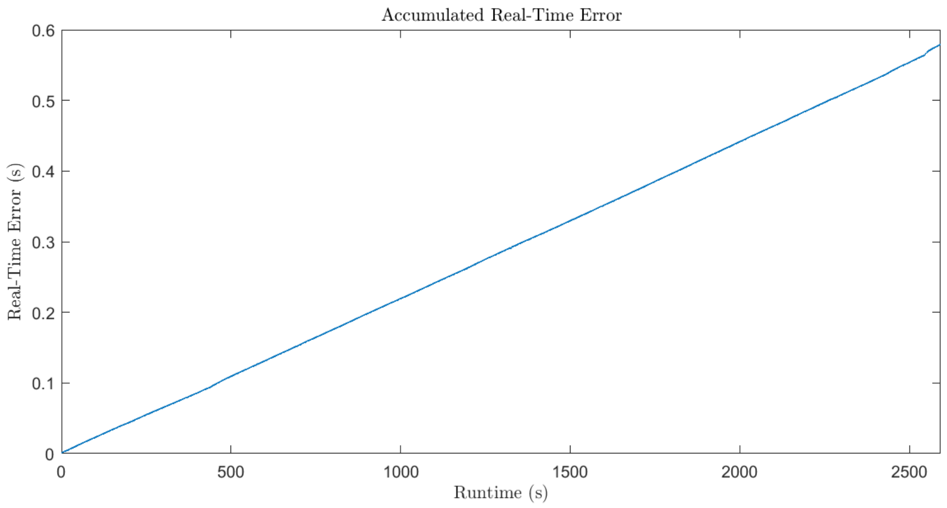
One of the most important benefits of the embedded systems is their **real-time behavior**. An approach of evaluation of the real-time behavior of the application takes place in this subsection. As it was mentioned, *bt_near_me()* function stores its execution moments (timestamps) in a file. The ideal situation is that every two neighboring timestamps differ by (*SEARCH_TIME* =) 10 seconds. However, in each iteration an error is added and the difference is not exactly 10 seconds. This fact is reasonable, as after the *recent contacts thread* sleeping time in each iteration, the operations that are executed inside the loop add a time overhead, that results in this deviation of the desirable behavior. So, the value

$$e_i = |(t_i - t_{i-1}) - 10|$$

will be called **real-time error** of iteration i. Also, after some iterations, this error is accumulated and leads to a bigger deviation from the desirable time value. This error will be called **accumulated real-time error**.

In the next figures, both the *real-time error per iteration* and the *accumulated real-time error* are presented:





The **mean** of *iteration error* is equal to 223.8 μs and its **variance** is 7.13 10^{-9} s^2. It is clear in the second diagram that at the error is accumulated in an almost constant way and reaches the value 0.5791 seconds after the 7 hour process.

3.3. Discussion & Evaluation

Overall, the presented application is considered as a **soft real-time** application and the order of magnitude of the *real-time error* is not considerable. A deviation of less than 1 second after 7 hours of execution will not affect the effectiveness of the program as its real-time requirements are not hard restricted. On the other hand, in a **hard real-time** system, such as an autopilot system, a *real-time error* of these values could be catastrophic. As a result, the real-time behavior of the presented application can be considered satisfactory for its requirements.