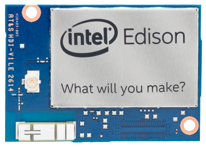
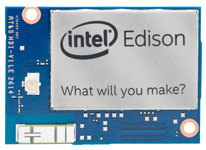
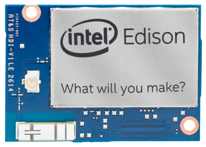


# Intel® Edison Tutorial: Timing Analysis and Synchronization



# Table of Contents

| Revision history | | |
|---|---|---|
| **Version** | **Date** | **Comment** |
| 1.0 | 10/28/2015 | Initial release |
| | | |
| | | |



## Introduction

Timing is a very critical component for all engineering projects, and even more so for computers and Internet of Things (IoT) applications. To quote an article from the National Institute of Standards and Technology (NIST) “Our fast-approaching future of driverless cars and "smart" electrical grids will depend on billions of linked devices making decisions and communicating with split-second precision to prevent highway collisions and power outages”.

As such, this tutorial will go over some basic timing analysis to check execution time for code. After the basic timing portion of this document, you will use two Intel Edison boards to see the difference in clocks.

In this tutorial you will

1. Learn how to experimentally determine how long your code took to execute
2. Manually edit the software and hardware clocks on your Intel Edison
3. Naively compare timestamps between different IoT nodes
4. Think about how to compare timestamps in a more robust manner

## Things Needed

1. 2x Intel Edison’s
2. A PC or a Mac
3. 2x Micro USB cables per Intel Edison
4. An internet connection

## Timestamps on Computers – Unix/Epoch/POSIX Time

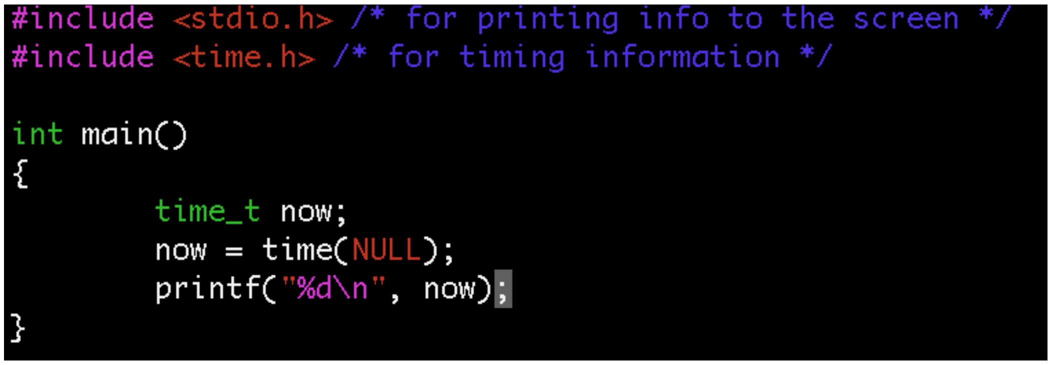
Computers, much like humans, need to have a concept of “time” so that they can perform tasks. However, as computers are much more suited to dealing with pure numbers than strings, they need to have a different way of recording time than the Gregorian Calendar.

The method most widely used on Unix like systems is **Epoch time**. This representation of time is defined as the number of seconds that have elapsed since 00:00:00, Thursday January 1st 1970 Coordinated Universal Time (UTC) **not counting leap seconds**. Since it does not account for leap seconds, this is not a **truly linear** or **true representation** of UTC. But it works very well for our purposes.

For example, the date January 1, 2016 00:00:00 UTC will have the **Epoch** timestamp of 1451606400.

Let’s examine what the UTC timestamp looks like, and how to access it through a C-program.

1. Log into your Intel Edison board **via serial interface**. You will be disabling the Wi-Fi for a later portion of this tutorial, which will break the SSH pipe between your Intel Edison and your personal machine.
2. Open a new C-file called “get_time.c”
3. Type the following code into the file



```
#include <stdio.h> /* for printing info to the screen */
#include <time.h> /* for timing information */

int main()
{
        time_t now;
        now = time(NULL);
        printf("%d\n", now);
}
```

**Figure 1: get_time.c**

4. Save and quit the file
5. Compile the source code into an executable binary file

   **$ gcc –o get_time get_time.c**

6. Run the executable

   **$ ./get_time**

7. Examine the output. Record it, and convert it to a human readable timestamp
8. Does the human readable timestamp agree with what time you think it should be?
9. What are some reasons that the timestamp may not be aligned with the real world time?

## Code Execution Time

Now that we can get the current time, let's try to see how long it takes to execute a code segment.

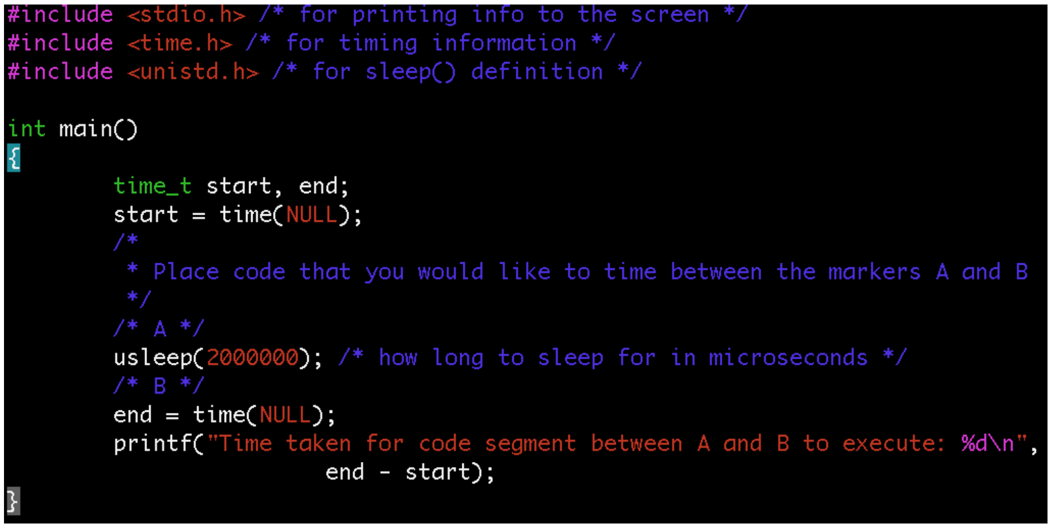
1. Open a file

   **$ vi simple_timing_analysis.c**

2. Type the following code into the file

   **NOTE:** You can place any code you want between the markers /* **A** */ and /* **B** */

   The reason we use **usleep()** is because we know that it should take approximately the same amount of time to return as the requested sleep time. Please see the following link for more information http://man7.org/linux/man-pages/man3/usleep.3.html



```
#include <stdio.h> /* for printing info to the screen */
#include <time.h> /* for timing information */
#include <unistd.h> /* for sleep() definition */

int main()
{
        time_t start, end;
        start = time(NULL);
        /*
         * Place code that you would like to time between the markers A and B
         */
        /* A */
        usleep(2000000); /* how long to sleep for in microseconds */
        /* B */
        end = time(NULL);
        printf("Time taken for code segment between A and B to execute: %d\n",
                        end - start);
}
```

Figure 2: simple_timing_analysis.c

3. Save and quit the file
4. Compile the source code into an executable binary file

   **$ gcc –o simple_timing_analysis simple_timing_analysis.c**

5. Run the executable

   **$ ./simple_timing_analysis**

6. Edit the source code file such that the sleep function is now for 2,100,000 microseconds
7. Compile and run the code
8. What do you notice about how long it said to run between A and B from step 6 compared to step 4?

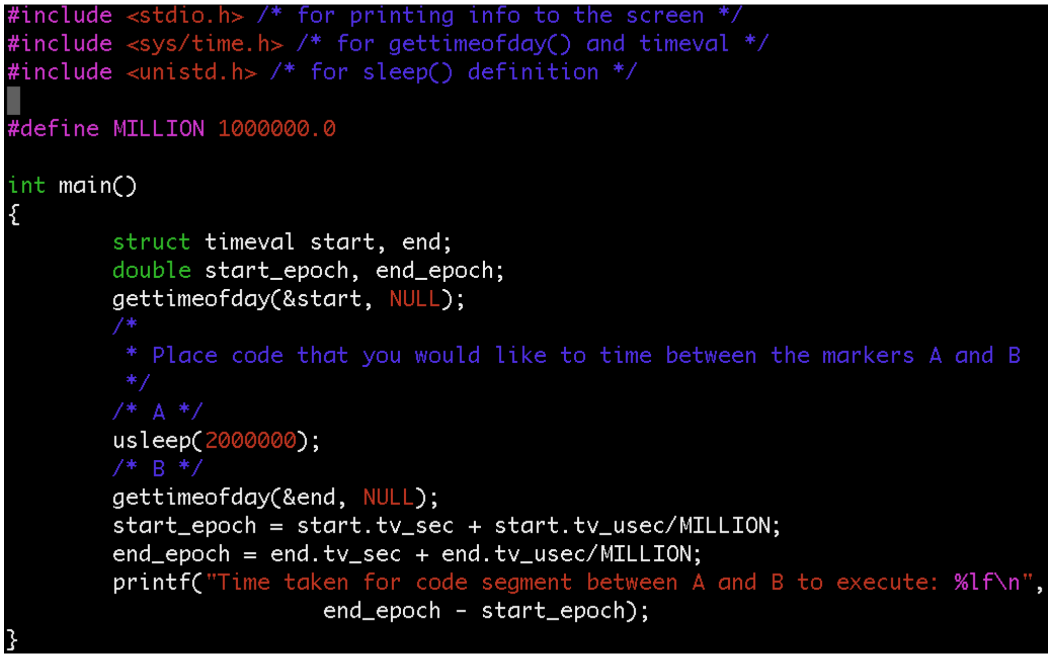
## Adding Precision

The above example gave you a good way to find out how long your code took to execute at the granularity level of a full second. However, this is not enough information in most applications. Typically, you will need to see how many microseconds it takes to run a segment of code. As such, we will need to use more precise data structures to check this time.

1. Open a new file

   **$ vi precise_timing_analysis.c**

2. Type the following code into the file



```
#include <stdio.h> /* for printing info to the screen */
#include <sys/time.h> /* for gettimeofday() and timeval */
#include <unistd.h> /* for sleep() definition */

#define MILLION 1000000.0

int main()
{
        struct timeval start, end;
        double start_epoch, end_epoch;
        gettimeofday(&start, NULL);
        /*
         * Place code that you would like to time between the markers A and B
         */
        /* A */
        usleep(2000000);
        /* B */
        gettimeofday(&end, NULL);
        start_epoch = start.tv_sec + start.tv_usec/MILLION;
        end_epoch = end.tv_sec + end.tv_usec/MILLION;
        printf("Time taken for code segment between A and B to execute: %lf\n",
                        end_epoch - start_epoch);
}
```

Figure 3: precise_timing_analysis.c

3. Save and quit the file
4. Compile the source code into an executable binary file

   **$ gcc –o precise_timing_analysis precise_timing_analysis.c**

5. Run the executable

   **$ ./precise_timing_analysis**

6. Edit the source code file such that the sleep function is now for 2,100,000 microseconds
7. Compile and run the code
8. What do you notice about how long it said to run between A and B from step 6 compared to step 4?

## What is Synchronization?

To understand the importance of synchronization, let's first examine an application. Imagine you are an engineer equipping a local high school with bells. The high school has four separated buildings. Each of these buildings needs its own bell to indicate to students and teachers that the current session of class is over. This high school is also very modern, and as such, the administration would like to send a command to the bells with a time of day to ring. The school has specified that the bell must ring at the specified time of day ± **1 minute**.

Once you are done equipping the high school with bells, you test them by issuing a command to ring the bells at **12pm** in the **GMT time zone**. Your test equipment records the bells ringing at the below times:

| Bell | Epoch |
|---|---|
| A | 1423569654 |
| B | 1423569032 |
| C | 1423570217 |
| D | 1423569551 |

### Tasks

1. Convert the above table from Epoch to GMT times
2. Record which bells are not ringing at the correct time

## Cause of Drift

Hopefully, the above example showed you how differences in clocks can cause a system to fail. Now, let's examine why the differences in clocks exist in the first place. To do this, we must learn how clocks work.

First, let's define an electronic oscillator as an electronic circuit or device that generates a periodically oscillating signal. This signal is typically either a sine or a square wave. This signal is then fed into a counter which counts each time the signal either crosses a pre-determined value (for example, each time the wave hits '0' on the y-axis). The counter then updates the clock based on how many counts it receives.

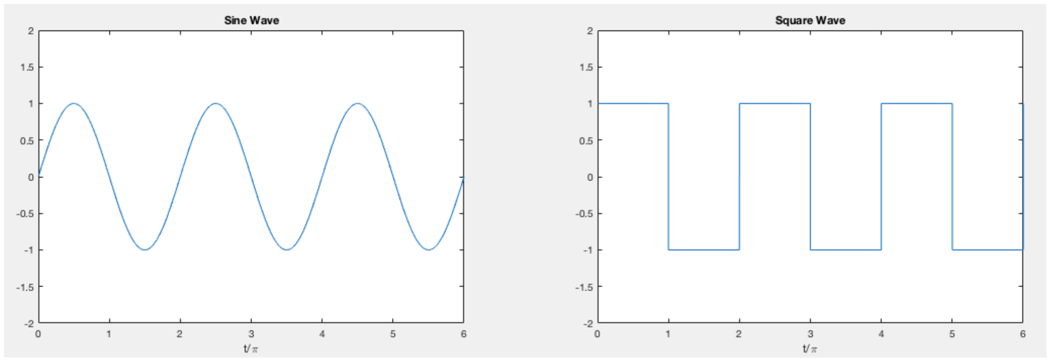


Figure 4: sine and square waves

An electronic oscillator comprises of two main elements, an amplifier and a crystal. To paraphrase a very understandable analogy from a user on Stack Exchange:

*Think of a crystal as being a tiny bell made from a piezoelectric material. Piezoelectric stems from the word* ***piezo*** *which is Greek for* ***push*** *and* ***electric*** *which is Greek for* ***amber*** *which was an ancient source of* ***electric charge****. When you hit a bell, it makes a pure sound. Similarly, a crystal makes electricity when you hit it and changes shape when you shock it with electricity.*

*The crystal is connected to an amplifier to continuously produce that pure "bell-like tone". The amplifiers job is similar to someone pushing you on a swing. When you get to just a little past the peak of one swing they'll give you a push to make sure you come back for another one.*

*The piezoelectric nature of the crystal causes it to change shape when the amplifier output "pushes" it with an electric signal. The amplifier then stops providing this "push". In response, the crystal "springs" back and generates its own electric signal. This signal is fed to the input of the amplifier at just the right time for the amplifier to generate another push, thus regenerating the cycle, forever.*

It is important to note that the reason crystals are used in generating timing pulses is because they can generate pulses at very specific and precise frequencies. For example, if a crystal is said to oscillate at 32 kHz, its nominal frequency is very likely to be 32,000.000000 Hz. However, there are external conditions that cause this frequency to change and drift.

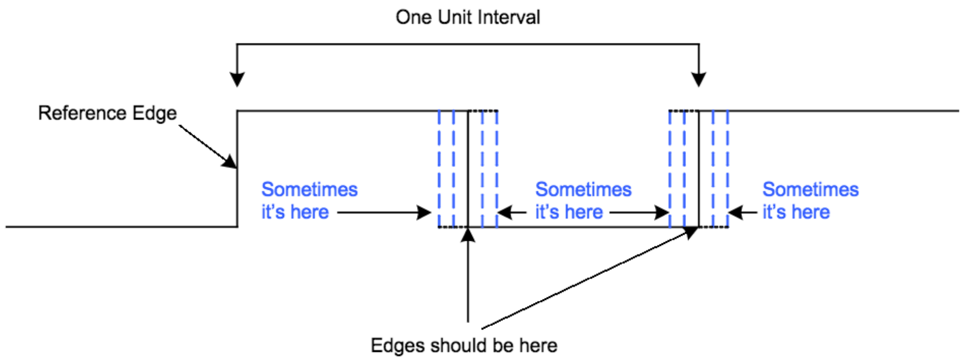


Figure 5: cause of time drift

One of these factors is temperature. To illustrate, consider a clock rated at 32,000.00Hz at room temperature. If the temperature that the crystal operates in is above or below this value, the clock frequency will now be slightly different. This relationship can be seen below:

$$f = f_0 \times [1 - K \times (T - T_0)^2]$$

Where

$f = real\ frequency$
$f_0 = nominal\ frequency$
$K = constant, for\ 32kHz\ crystals\ its\ about\ 0.4\ parts\ per\ million/°C^2$
$T = real\ temperature$
$T_0 = nominal\ temperature$

Consider a clock rated at 32MHz rated at standard room temperature (27C) operating at an actual temperature of 17C

$$f = 32kHz \times \left[1 - \frac{0.04}{1{,}000{,}000} \times (17 - 27)^2\right] = 31{,}999.872Hz$$

It also follows that for this difference in frequency at this temperature, we lose about 2 minutes per year.

$$time\ difference = \left(1\ year - \frac{31{,}999.872}{32{,}000} \times 1 year\right)(365days \times 24hours \times 60minutes)$$
$$= 2.1\ minutes$$

For further reading, please examine the below links

- http://electronics.stackexchange.com/questions/117624/how-does-a-crystal-work
- https://en.wikipedia.org/wiki/Crystal_oscillator
- https://www.edgefx.in/crystal-oscillator-circuit-working-applications/

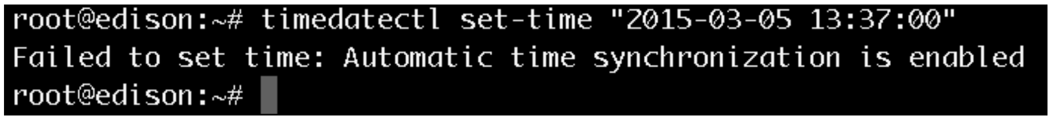
## Editing the Clock Manually

To mitigate the problem of drift, we can align our clocks to some central clock that we know has accurate time. Similar to a watch, you can edit the clock on your Intel Edison to make it align with a more correct clock. Let's try to set the date of the Intel Edison board to March 5th, 2015 1:37pm

1. Make sure your Intel Edison is connected to the internet
   Type the following command

   **$ timedatectl set-time "2015-03-05 13:37:00"**

   Notice how you get the below error when trying to set the date using this method

   ```
   root@edison:~# timedatectl set-time "2015-03-05 13:37:00"
   Failed to set time: Automatic time synchronization is enabled
   root@edison:~#
   ```

   Figure 6: error from attempting to change the clock manually

   This is because the Intel Edison already has a service that synchronizes your device across the internet

   **NOTE:** The date format is by default: "**YYYY-MM-DD hh:mm:ss**". The reasoning behind this is that when you sort dates in strings, it is important to have the year first.

   To illustrate, consider formatting the date string as the standard American definition "**MM-DD-YYYY**". Now, let's examine the following array of strings:

   ["03-05-2015", "03-04-2014", "02-06-2016"]

   If we were to sort this lexicographically (by simply looking at each character and not considering what the data actually represents), the order of the strings would be:

   ["02-06-2016", "03-04-2014", "03-05-2015"]

   The above output is clearly incorrect when looking at the data the strings represent

   However, if you have the dates formatted as the following strings "**YYYY-MM-DD**"

   ["2015-03-05", "2014-03-04", "2016-02-06"]

   When they are sorted lexicographically, the output would be

   ["2014-03-04", "2015-03-05", "2016-02-06"]

   Which is the correct way to order the timestamps

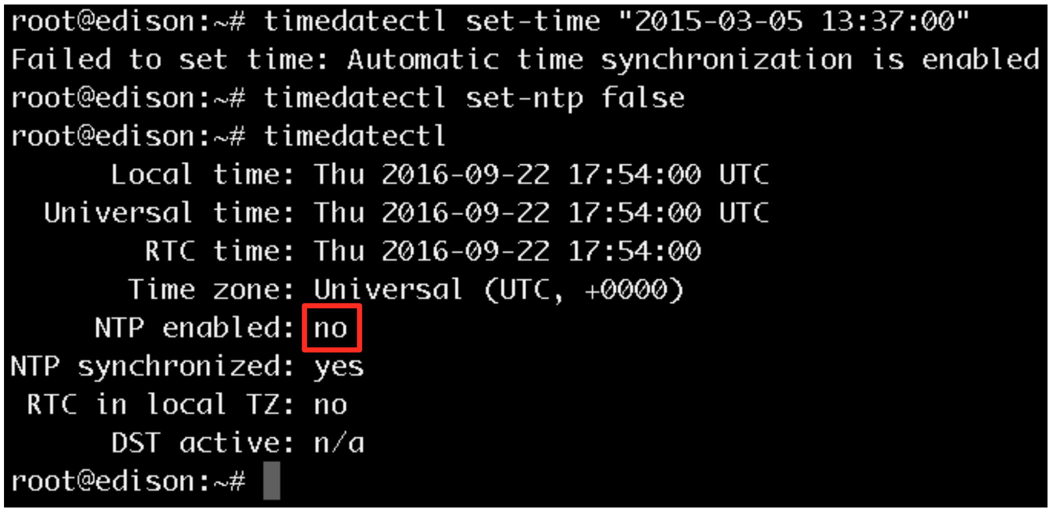
2. Now, let’s force change the clock. First we must to disable the automatic synchronization

   **$ timedatectl set-ntp false**

3. Verify that the automatic synchronization has been disabled

   **$ timedatectl**

   The output should have the **NTP enabled** value as **no**



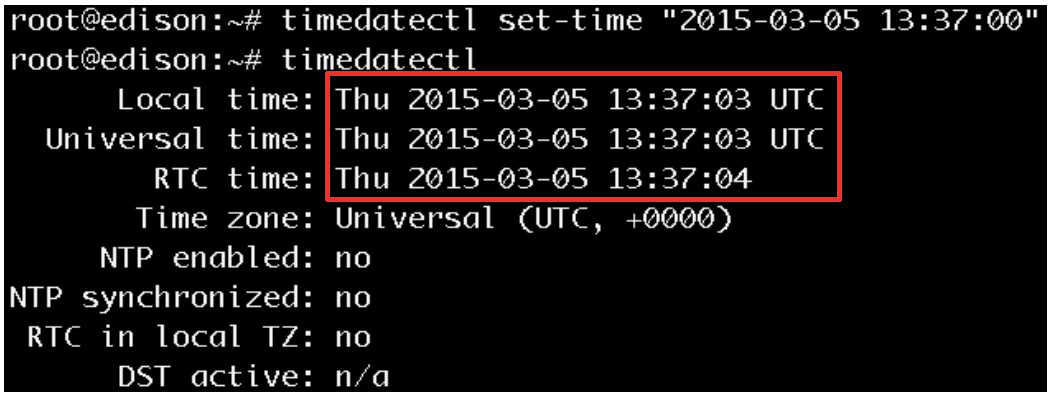
```
root@edison:~# timedatectl set-time "2015-03-05 13:37:00"
Failed to set time: Automatic time synchronization is enabled
root@edison:~# timedatectl
      Local time: Thu 2016-09-22 17:54:00 UTC
  Universal time: Thu 2016-09-22 17:54:00 UTC
        RTC time: Thu 2016-09-22 17:54:00
       Time zone: Universal (UTC, +0000)
     NTP enabled: no
NTP synchronized: yes
 RTC in local TZ: no
      DST active: n/a
root@edison:~#
```

Figure 7: successfully disabling synchronization

4. Now that the NTP is disabled, let’s set the time

   **$ timedatectl set-time “2015-03-05 13:37:00”**
   **$ timedatectl**

   The output should match the output below



```
root@edison:~# timedatectl set-time "2015-03-05 13:37:00"
root@edison:~# timedatectl
      Local time: Thu 2015-03-05 13:37:03 UTC
  Universal time: Thu 2015-03-05 13:37:03 UTC
        RTC time: Thu 2015-03-05 13:37:04
       Time zone: Universal (UTC, +0000)
     NTP enabled: no
NTP synchronized: no
 RTC in local TZ: no
      DST active: n/a
```

Figure 8: clock after manual adjustment

5. Let's now re-enable time synchronization by issuing the following command

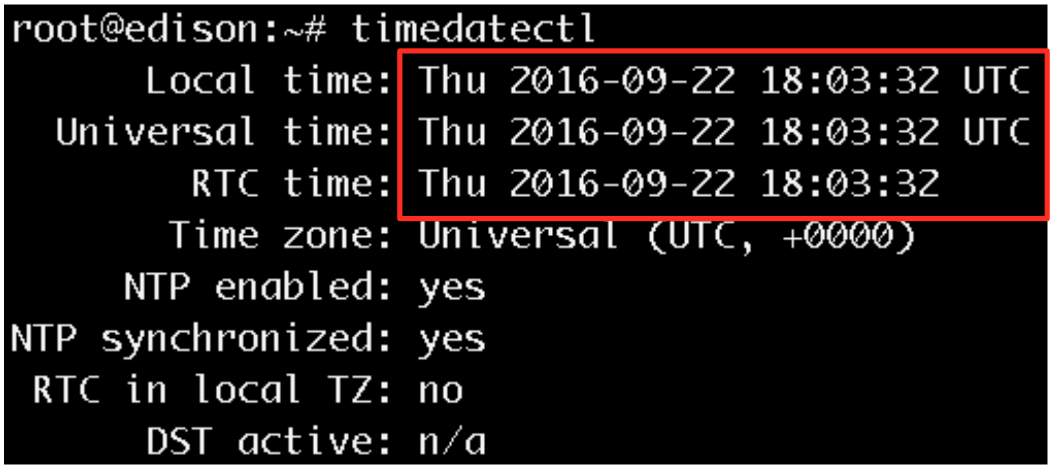
   **$ timedatectl set-ntp true**

   **Wait about one minute** and examine the output from **timedatectl**

   It is critical to wait as if you check the output right away, the operating system may not have queried the NTP server, so the clock may appear wrong.

   Even if the Local and Universal time are correct, the RTC time may not be. RTC stands for **R**eal **T**ime **C**lock, it is a module (usually a piece of hardware) that keeps track of time with its own internal power source. This is so that when the overall system doesn't lose track of time when it has no power supplied to it. The RTC is usually updated more slowly than the software clock

   **$ timedatectl**



```
root@edison:~# timedatectl
      Local time: Thu 2016-09-22 18:03:32 UTC
  Universal time: Thu 2016-09-22 18:03:32 UTC
        RTC time: Thu 2016-09-22 18:03:32
       Time zone: Universal (UTC, +0000)
     NTP enabled: yes
NTP synchronized: yes
 RTC in local TZ: no
      DST active: n/a
```

Figure 9: clock after enabling synchronization

   **Note:** Check the **Local**, **Universal** and **RTC** time. They should all be the same, but they will not agree with the above figure, since the screenshot was taken at the time of writing this tutorial.

## Difference in Timestamps

Now that you understand that the timestamps ***can*** be different, let's try and show that they ***are*** different.

1. Connect your Intel Edison's to the internet
2. Designate one Intel Edison as the server
3. Discover the IP address of your server Intel Edison

   **$ configure_edison --showWiFiIP**

4. Push the file labelled "server.c" in the folder labelled "FILES/initial/" to your server Intel Edison via SFTP
5. Compile the file

   **$ gcc –o server server.c**

6. Designate the other Intel Edison as the client
7. Push the file labelled "client.c" in the folder labelled "FILES/initial/" to your client Intel Edison via SFTP
8. Compile the file

   **$ gcc –o client client.c**

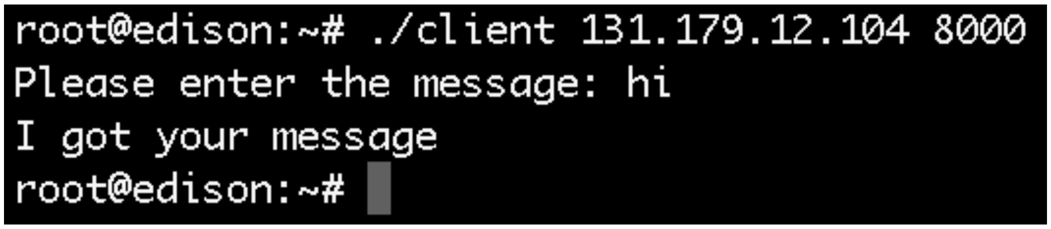
9. Start the server on your server Intel Edison

   **$ ./server <PORT_NO>**

10. Start the client on your client Intel Edison

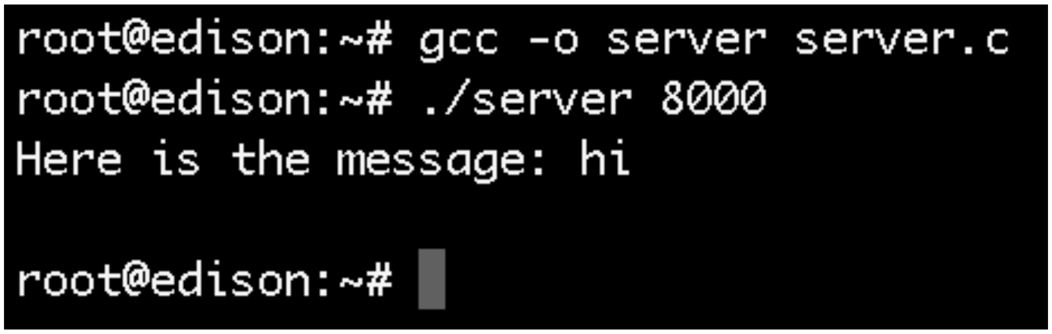
    **$ ./client <IP_ADDR_SERVER> <PORT_NO>**

11. Verify that the client and server are functioning correctly



```
root@edison:~# ./client 131.179.12.104 8000
Please enter the message: hi
I got your message
root@edison:~#
```

Figure 10: client functioning correctly



```
root@edison:~# gcc -o server server.c
root@edison:~# ./server 8000
Here is the message: hi

root@edison:~#
```

Figure 11: server functioning correctly

Now that we have verified that the server and client code works, let's examine the difference in timestamps

To do this, the client Intel Edison will generate the current time, and send this as a string to the server Intel Edison. Once the server Intel Edison receives a message from the client, it will generate its own timestamp.

The server Intel Edison will then compute the difference in timestamps

$$timestamp\ difference = server\ time\ stamp - client\ time\ stamp$$

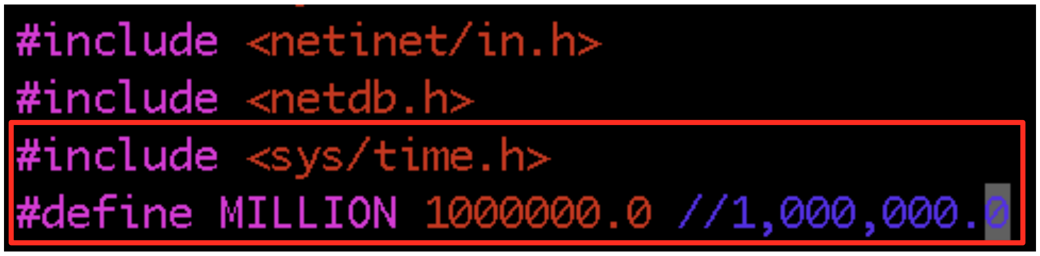
Follow the below steps to generate a current timestamp with microsecond precision on the client and send it to the server as a string

12. Open the client.c file on your client Intel Edison

    **$ vi client.c**

13. Include the sys/time.h library and define MILLION

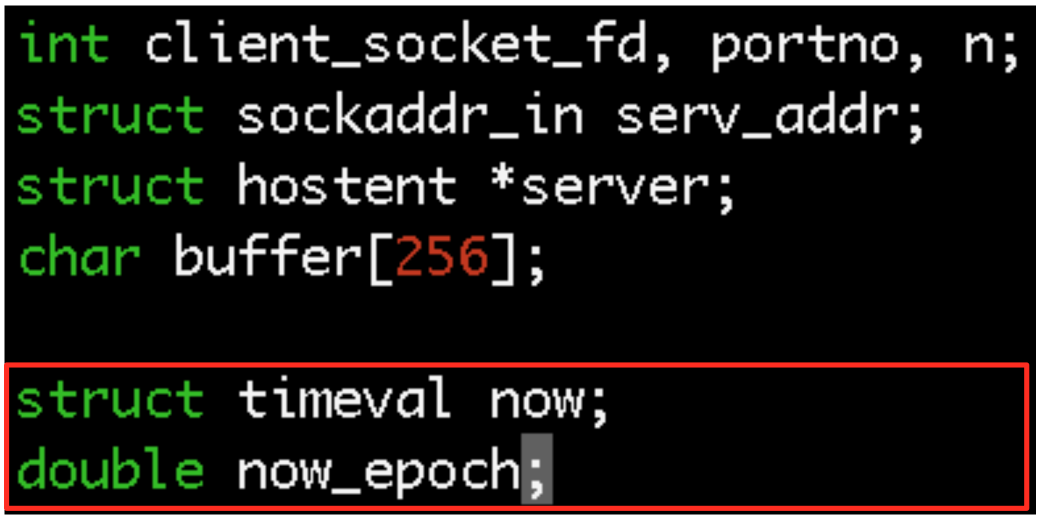
    **#include <sys/time.h>**
    **#define MILLION 1000000.0 //1,000,000.0**



```
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>
#define MILLION 1000000.0 //1,000,000.0
```

Figure 12: including the "sys/time.h" library and defining MILLION

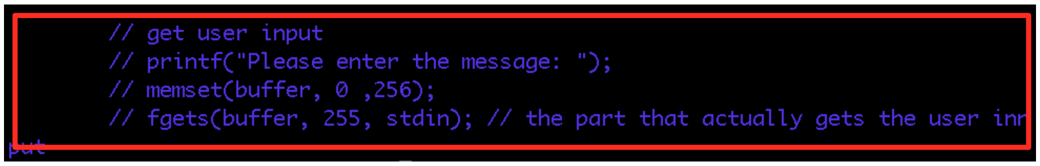
14. Declare the variables **now** and **now_epoch** as follows



```
int client_socket_fd, portno, n;
struct sockaddr_in serv_addr;
struct hostent *server;
char buffer[256];

struct timeval now;
double now_epoch;
```

Figure 13: declaring now and now_epoch

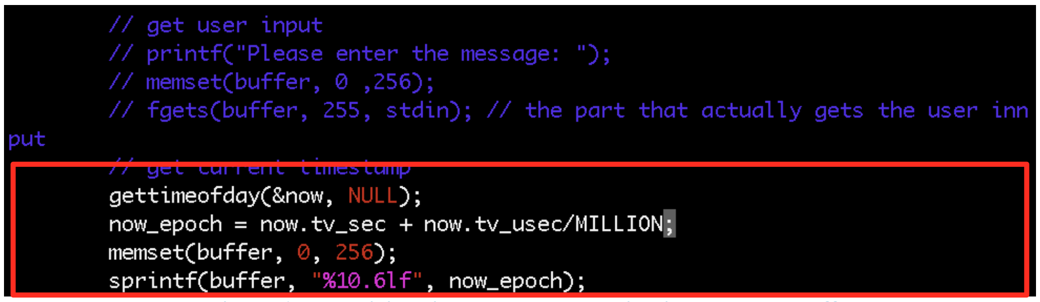
15. Comment out the section that says "get user input"



```
        // get user input
        // printf("Please enter the message: ");
        // memset(buffer, 0 ,256);
        // fgets(buffer, 255, stdin); // the part that actually gets the user inp
put
```

Figure 14: commented out code

16. Add the following lines directly below



```
        // get user input
        // printf("Please enter the message: ");
        // memset(buffer, 0 ,256);
        // fgets(buffer, 255, stdin); // the part that actually gets the user inp
put
        // get current timestamp
        gettimeofday(&now, NULL);
        now_epoch = now.tv_sec + now.tv_usec/MILLION;
        memset(buffer, 0, 256);
        sprintf(buffer, "%10.6lf", now_epoch);
```

Figure 15: acquiring timestamp and placing in character buffer

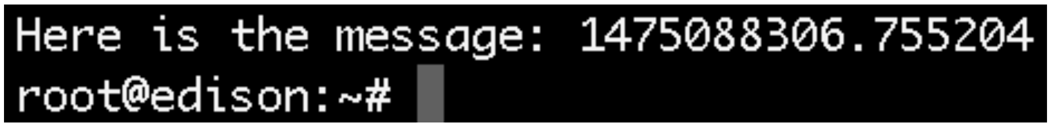
17. Save and quit the file. Then compile the code

**$ gcc –o client client.c**

18. Run the code

**$ ./client <SERVER_IP_ADDR> <PORT_NO>**

19. Repeat steps 9 and 10 above, the output at the server should look similar to below (but with a different timestamp)



```
Here is the message: 1475088306.755204
root@edison:~#
```

Figure 16: correct client-server behavior

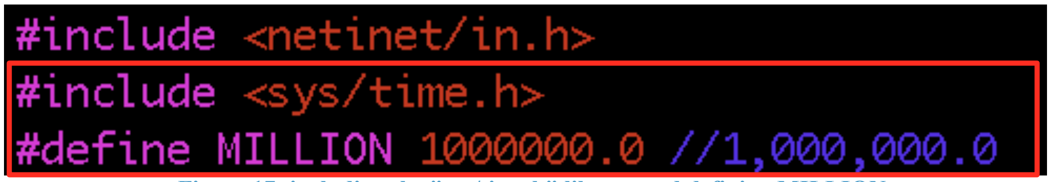
Now that we have the client working, let's modify the server in a similar fashion to generate the timestamp directly after reading the message the client sent

20. Open the server.c file

    **$ vi server.c**

21. Include the sys/time.h library and define MILLION

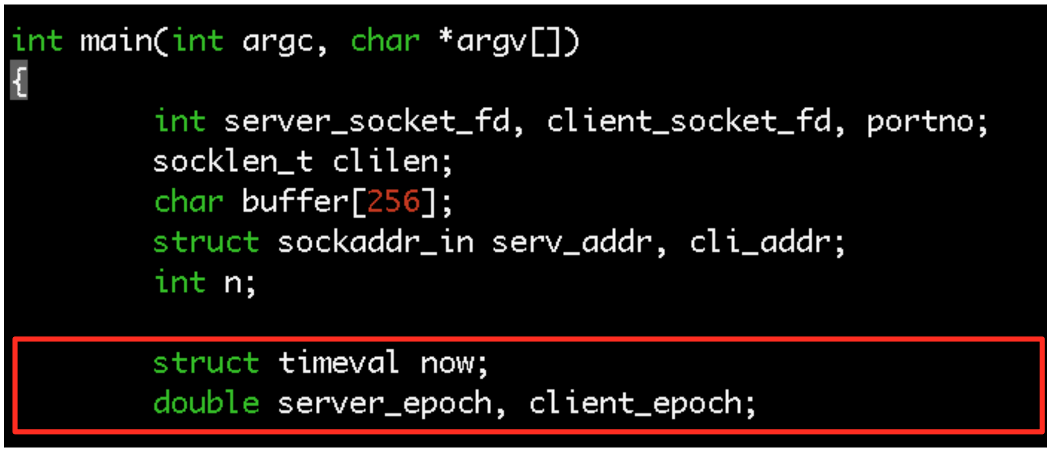
    **#include <sys/time.h>**
    **#define MILLION 1000000.0 //1,000,000.0**



```
#include <netinet/in.h>
#include <sys/time.h>
#define MILLION 1000000.0 //1,000,000.0
```

Figure 17: including the "sys/time.h" library and defining MILLION

22. Declare the variables **now**, **server_epoch**, and **client_epoch** as follows

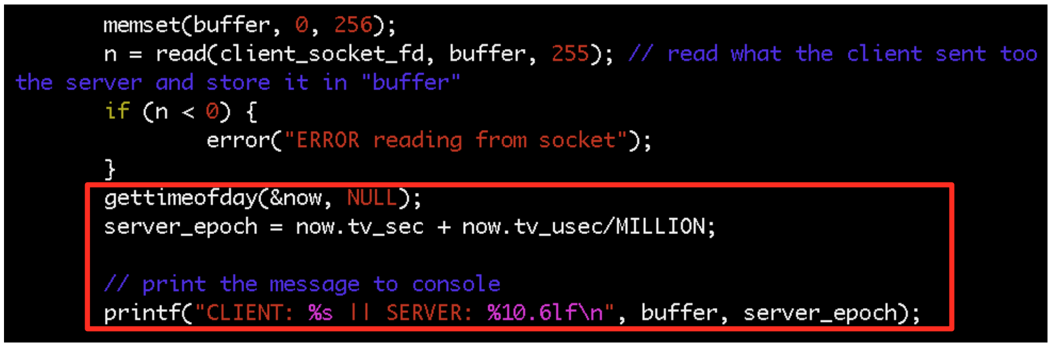


```
int main(int argc, char *argv[])
{
        int server_socket_fd, client_socket_fd, portno;
        socklen_t clilen;
        char buffer[256];
        struct sockaddr_in serv_addr, cli_addr;
        int n;

        struct timeval now;
        double server_epoch, client_epoch;
```

Figure 18: declaring now, server_epoch and client_epoch

23. Get the time of day directly after reading a message from the client

    Edit the message the server displays when it has received a message to show the timestamp received from the client, and the timestamp recorded by the server



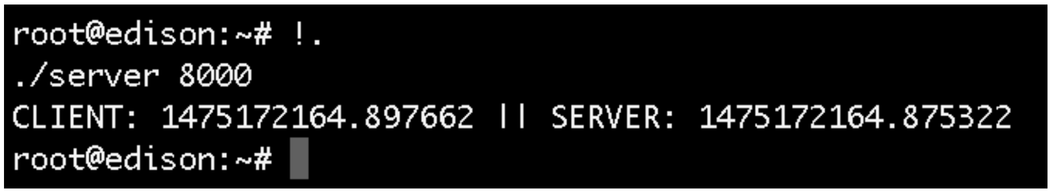
```
        memset(buffer, 0, 256);
        n = read(client_socket_fd, buffer, 255); // read what the client sent too
the server and store it in "buffer"
        if (n < 0) {
                error("ERROR reading from socket");
        }
        gettimeofday(&now, NULL);
        server_epoch = now.tv_sec + now.tv_usec/MILLION;

        // print the message to console
        printf("CLIENT: %s || SERVER: %10.6lf\n", buffer, server_epoch);
```

Figure 19: getting epoch and editing printf statement

24. Save and quit the file. Then compile the code

    **$ gcc –o server server.c**

25. Repeat steps 9 and 10 above, the output at the server should look similar to below (but with a different timestamp)

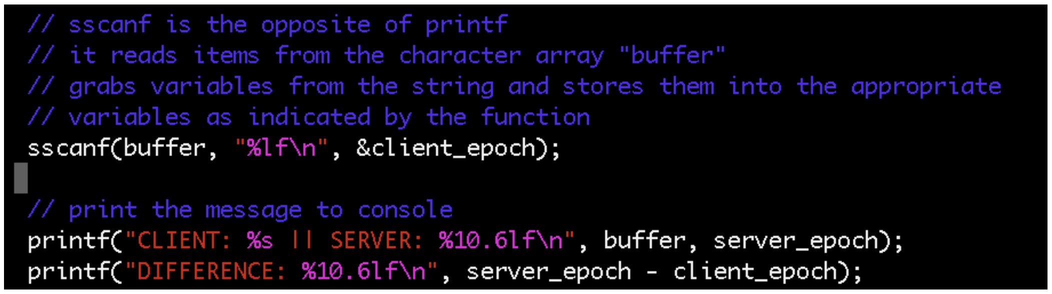


```
root@edison:~# !.
./server 8000
CLIENT: 1475172164.897662 || SERVER: 1475172164.875322
root@edison:~#
```

Figure 20: server output

26. Notice how it's difficult to immediately see the difference in timestamps. Let's try to get the program to generate the difference automatically. For this, we must first convert the message we received as a **string** to a **double**.

    Edit the **printf** section of the code such that it matches the below screenshot



```
 // sscanf is the opposite of printf
 // it reads items from the character array "buffer"
 // grabs variables from the string and stores them into the appropriate
 // variables as indicated by the function
 sscanf(buffer, "%lf\n", &client_epoch);

 // print the message to console
 printf("CLIENT: %s || SERVER: %10.6lf\n", buffer, server_epoch);
 printf("DIFFERENCE: %10.6lf\n", server_epoch - client_epoch);
```

Figure 21: printing the difference in time stamps

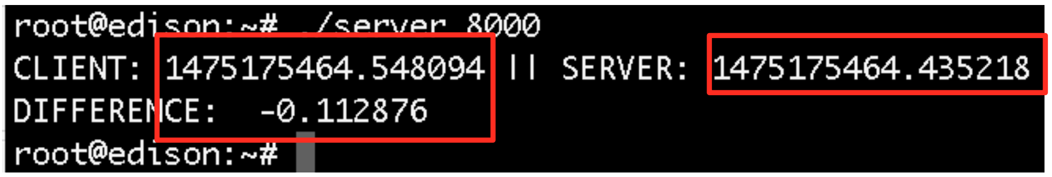
Since we know the input will only ever be a floating point, we use the function **sscanf** to extract the timestamp information. However, we should only use this function if we are guaranteed to have input in the correct format. Please examine the below links if you would like more information about how **sscanf** works

https://linux.die.net/man/3/sscanf
https://www.tutorialspoint.com/c_standard_library/c_function_sscanf.htm
http://www.cplusplus.com/reference/cstdio/sscanf/

27. Save and quit the file. Then compile the code

    **$ gcc –o server server.c**

28. Repeat steps 9 and 10 above, the output at the server should look similar to below (but with differences in the highlighted boxes)



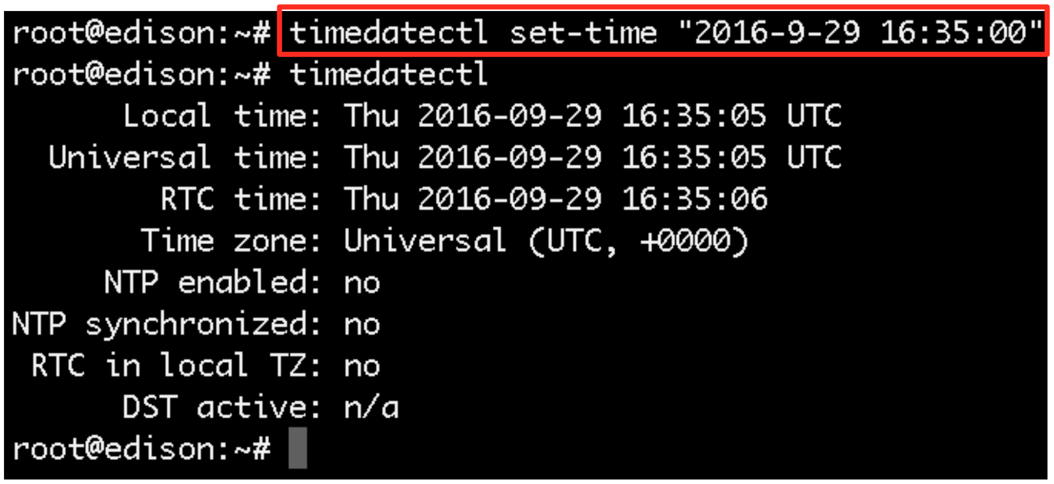
```
root@edison:~# ./server 8000
CLIENT: 1475175464.548094 || SERVER: 1475175464.435218
DIFFERENCE:  -0.112876
root@edison:~#
```

Figure 22: server output

29. Let's simulate an event where the client's clock drifts

    Type the below commands to set the **client's** clock back about 10 minutes

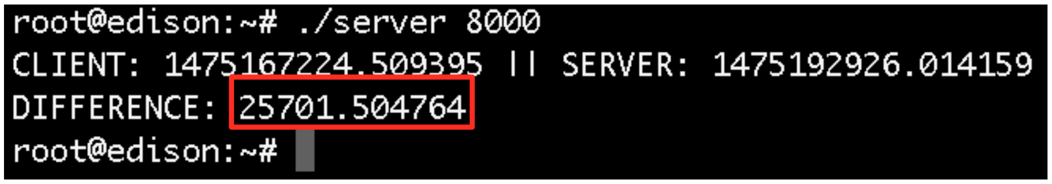
    **$ timedatectl set-ntp false**
    **$ timedatectl set-time <TEN_MINUTES_AGO>**



```
root@edison:~# timedatectl set-time "2016-9-29 16:35:00"
root@edison:~# timedatectl
      Local time: Thu 2016-09-29 16:35:05 UTC
  Universal time: Thu 2016-09-29 16:35:05 UTC
        RTC time: Thu 2016-09-29 16:35:06
       Time zone: Universal (UTC, +0000)
     NTP enabled: no
NTP synchronized: no
 RTC in local TZ: no
      DST active: n/a
root@edison:~#
```

Figure 23: manually setting the time

30. Repeat steps 9-10 above, notice the timestamp difference is now much larger



```
root@edison:~# ./server 8000
CLIENT: 1475167224.509395 || SERVER: 1475192926.014159
DIFFERENCE: 25701.504764
root@edison:~#
```

Figure 24: server output

31. Re-enable NTP on your client Intel Edison

    **$ timedatectl set-ntp true**

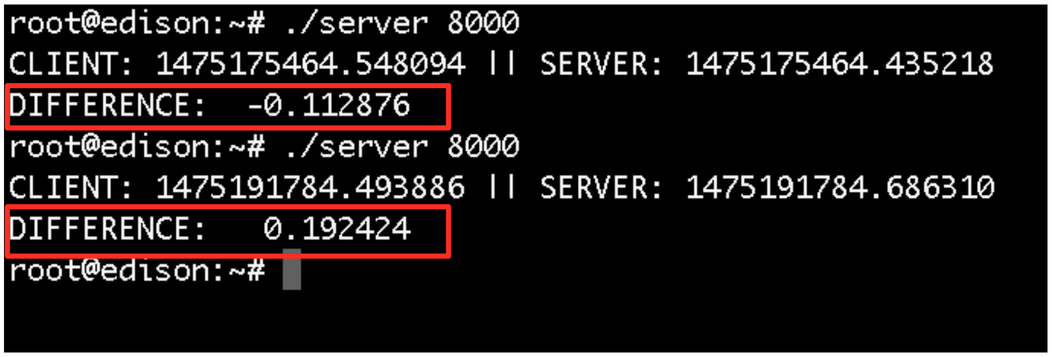
## Tasks

1. The cause of this timestamp difference has factors other than differences in internal clocks. List two other possible reasons for the timestamps being different

   **HINT:** (1) has your Wi-Fi ever been slow? (2) Do all the lines of code execute at the same time?

2. Describe a strategy to try and find out how each the factors you have listed above the difference in timestamps

3. Repeat steps 9-10 above, examine the value of **DIFFERENCE**. Notice that it changes. Describe a method that would be a enable you to find the average clock difference



```
root@edison:~# ./server 8000
CLIENT: 1475175464.548094 || SERVER: 1475175464.435218
DIFFERENCE:  -0.112876
root@edison:~# ./server 8000
CLIENT: 1475191784.493886 || SERVER: 1475191784.686310
DIFFERENCE:   0.192424
root@edison:~#
```

Figure 25: change in value of "difference" after successive code runs

4. **EXTENSION**

   Implement your idea from tasks 2 and 3