Sync Tests

Bram Vanbilsen, Adam El M'Rabet, Liam Volckerick

February 2020

1 Introduction

In a previous report a system was described to synchronize between multiple devices for performing specific tasks (think for example of a synchronized countdown displayed on multiple screens belonging to separate but connected devices). Stating that clocks are 'synchronized' can be interpreted in different ways. It is therefore needed to describe it in a more formal way supported by scientific tests.

A central aspect that could be used for synchronizing tasks is the local clock of the client's device. Experiments show that clocks might vary depending on multiple factors [?]. The central goal of this report is to investigate these factors and formulate a formality of clock-offsets through empirical analysis by running tests on different devices.

Near the end of this report we will also touch on an alternative task synchronization method that is independent of the local clock. We then end this report by a comparison of both methods and a conclusion.

2 Transferring data packets over networks

2.0.1 TCP

TCP is a transmission protocol used by HTTP(s), FTP and other common protocols [?]. It prioritizes reliability. It uses various methods to make sure a packet does not get corrupted or lost, e.g. re-sending lost packets and making the receiver acknowledge that it received the packet. As a result, TCP is very reliable and is great for sending data on (less) stable connections and for sending data that needs to be 100% correct.[?]

2.0.2 UDP

Besides TCP, there is the UDP protocol which prioritizes speed. In general, UDP packets are smaller in size compared to TCP packets due to less required headers for error checking and ordering. Usually, it does not make any attempt to make sure packets arrive or arrive without corruption. As a result, UDP is great for fast communication that can handle some missing or incorrect data [?]. Because of these properties, it is used by the NTP protocol [?].

2.1 NTP

The network time protocol (NTP) uses a reference time to sync clocks of computers over the internet. At its core, NTP works as follows: a client requests the current time from a server, passing its own time with the request. The server adds its time to the data packet and passes the packet back to the client. Upon reception of the packet at the client side, the client can retrieve two important sets of information. The reference time of the server and the travel time of the packet (as measured locally). Multiple iterations of the strategy mentioned above can provide a clearer (reducing network jitter) view of the delay between the local clock and the reference clock server side [?] [?].

3 Investigation of accuracy of local device clocks

In this section, we investigate how accurate local device clocks are. We test a multitude of different operating systems which all use NTP (or some variation on it). Because we checked `time.is` on some devices before starting this study, we already know that local device clocks are not always as well synchronized as we would hope (or perhaps `time.is` has some bugs). In fact they are almost always different by some varying amount of milliseconds. The goal of this section is to investigate how much these clock offsets can vary for multiple devices and specifications.

3.1 Strategy

In order to investigate the accuracy of local clock-times scientifically and compare between multiple devices and operation systems, we need to make an empirical analysis by running an experiment a number of times and document the results.

The main goal of this report is to find a correlation between the device type or specification and the difference in clock offsets. We decided to proceed as follows:

1. Record the local device's current time ($= t0$).
2. Get $NTPTime$ from the 'ntp.UGent.be' time server pool.
3. Record the local device's current time ($= t1$).
4. Calculate average time it takes to reach 'ntp.UGent.be': $travelTime = \frac{t1-t0}{2}$.
5. Calculate clock offset: $(NTPTime + travelTime) - t1$

For being able to formulate a precise conclusion on experiments, it is important to make a number of (in this report called) iterations. We therefore decided to repeat this method 100 times, record the result of every iteration in *CSV-files*, show the results graphically on charts and base our conclusions on the median and average results[1].

Unless mentioned otherwise, the NTP server used by the operating system is default. A similar method to benchmark device clock accuracy is also described in the Microsoft documentation [?].

Important is that all the tests were executed in a Node environment, not in the browser. This allows us to use the UDP protocol [?] which in turn allows us to use the NTP protocol [?].

Because running a node environment is not straight forward on mobile devices such as iOS and Android, we wrote a wrapper around our tests to run natively on these systems using React Native [?].

[1]In most cases the median was very close to the average. But we also calculated the average because for some tests the average gave a better result than the median due to the small amount of devices and big differences in offsets between those devices. That's why we decided to take into account both the average and median.

3.2 Hypothesis

We expect Apple devices to have well synchronized clocks. The reason being that MacOS and iOS only run on a handful of hardware. As a result, Apple can focus on optimization instead of compatibility. It is also for this reason that we expect Windows, Ubuntu (in general Linux distributions) and Android to be less synchronized. It'd make sense that some compromise had to be made to find an optimal solution for such a great amount of hardware.

3.3 Scientific tests

3.3.1 MacOS

We only had one MacOS device available, but the results that we got are similar to other groups that showed us `time.is` on their MacOS devices. On average, the device was 15.72 milliseconds behind the NTP server and the median delay was 15 milliseconds. As shown in the graph below, the maximum offset with the NTP server was 27 milliseconds.

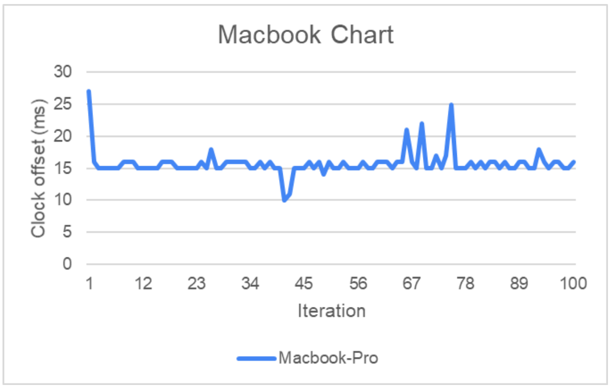

Figure 1: Clock offsets on MacOS systems

This spike of 27 milliseconds was at the very start of the execution of the tests. We noticed similar behavior with the other operating systems. This is likely due to the fact that Javascript is single threaded. At the beginning of the execution, our Node.js environment is starting up which can cause a larger call stack as supposed to when the program is already up running. [?]
Since recently, Apple does not allow MacOS users to change NTP settings [?]. It thus seems that Apple feels very confident about their own optimization of their systems. Because of the small amount of hardware supported, they were supposedly able to optimize their NTP daemon to get such good results (see 3.3.4).

3.3.2 Ubuntu

We tested 4 different types of Ubuntu systems and were fairly surprised by how synchronized all clocks were. As shown in graph 2, the maximum time difference between the NTP server and our devices was only 22 milliseconds.

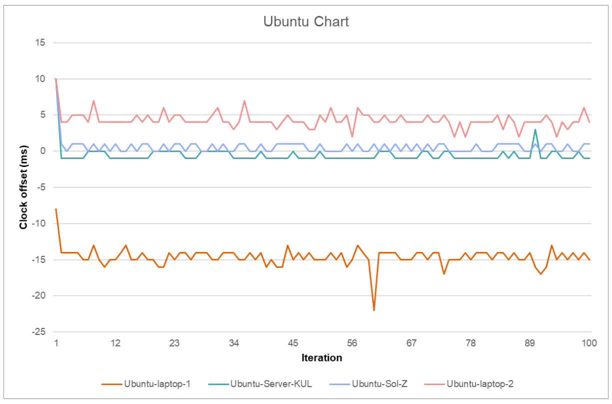

Figure 2: Clock offsets on Ubuntu systems

The devices were on average 2,5775 milliseconds ahead of the NTP server and the median clock difference was 0 milliseconds! Also, all these systems use the systemd-timesyncd daemon [?] [?] to sync their clocks. This implements SNTP rather than NTP (the more accurate, but heavier, protocol).
To get such good results, Ubuntu (or at least the versions of Ubuntu we could test) determines a poll interval to sync with the NTP server based on the device's clock drift [?]. Diving a little deeper into utilization, we found that the 'systemd' process (which includes 'timesyncd'), on average, used 0% of the CPU and 0% (out of 8GB) of the RAM. This was on the ertvelde server provided by KUL. We chose this system because it has shared resources across multiple users, thus many processes may be running at the same time. For example, many node servers were running by other students. Thus this system is constantly under load and yet has no problem syncing its clock with an accuracy of just 0.54ms on average!
Out of these results, we conclude that our hypothesis was completely wrong regarding Ubuntu systems. Due to a vary optimized and lightweight SNTP implementation [?], clocks on Ubuntu systems are very accurate.

3.3.3 Windows

We were able to test 5 different Windows machines and got results in line with our hypothesis. As seen in the chart below, there was a lot more variation between the Windows systems compared to the Ubuntu systems.

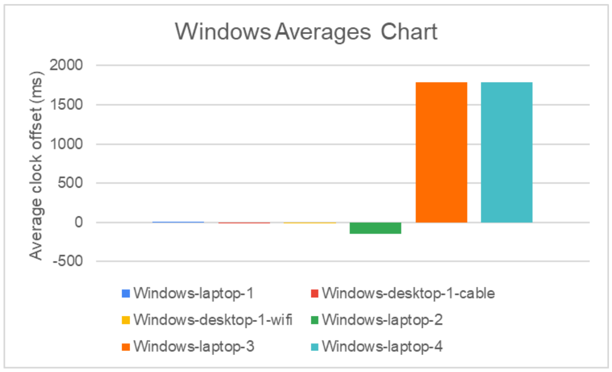

Figure 3: Clock offsets on Ubuntu systems

On average, the clock difference for Windows systems compared to the NTP server was $570,08$ milliseconds while the median difference was only $7,5$ milliseconds. In this case (due to the small number of devices) the average illustrates the reality better than the median.
As expected, the differences between the Windows systems are greater than with other operating systems. Windows explains that these differences [**?**] may be caused by the following three factors:

- Network conditions
- The accuracy of the computer's hardware clock
- The amount of CPU and network resources available to the Windows Time service

Other than 'Windows-desktop-1' (see chart 3), the devices were connected to the same wifi network. Thus the network conditions were very similar for the devices. The accuracy of the computer's hardware clock is difficult to check and is prone to clock drift errors [**?**]. But given an NTP implementation that syncs clocks regularly, this should also not be a big issue. Lastly, the amount of CPU and network resources available might be what causes the large deviation in results. The 'highest-spec' device was the desktop and as expected, this one had the smallest clock offset. But, 'Windows-laptop-2' has far worse specs than 'Windows-laptop-3' and 'Windows-laptop-4' (see chart 3), yet it has a smaller clock offset. 'Windows-laptop-3' and 'Windows-laptop-4' are from the era of 'ultrabooks' (compared to an old bulkier laptop as 'Windows-laptop-2'), in which battery life is a big selling point (we did not find any reports confirming clock synchronization was worse on ultrabooks though).

Notice that we ran the test twice on 'Windows-desktop-1' (see chart 3) once using a cable connection and once using wifi. These results are nearly identical, the cable test had an average offset of $-6,1ms$ and the wifi test $-5,97ms$. Because we accounted for the UDP packet's travel time, this result was expected. But it is important to keep in mind that the calculated travel time is just an estimate of the real travel time. For example, it may take just $5ms$ to reach the NTP server, but the NTP server could be very busy. This may result in a difference between when the NTP server records its time and when the server actually sends back the packet containing the time. This 'lag' on the server might cause errors. This example makes it clear that the device's operating systems should use a reliable NTP server. To check how reliable the default NTP server (`time.windows.com`) for Windows 10 is, we tested 'Windows-laptop-3' (see chart 3) again with a different NTP server (`time.nist.gov`). The results show a clear improvement: an average offset of $753,31ms$ (from $1783,55ms$).
These results clearly show that the accuracy of the clock on Windows devices vary greatly from device to device and from what NTP server is used to synchronize clocks.

3.3.4 iOS

We tested 4 iOS devices (3 iPhones and 1 iPad) and got results similar to what we expected. The clock offset was on average $8,47ms$ and the median was just $7,5ms$.

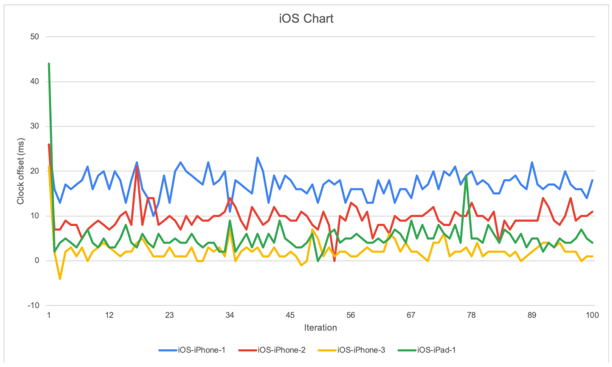

Figure 4: Clock offsets on iOS systems

During our research, we found that iOS and MacOS share a common NTP codebase [?]. Thus, similar results to MacOS, as shown in graph 1 and 4, were to be expected.

3.3.5 Android

As with the other operating systems, Android does use NTP by default [?]. The results compared to the other operating systems were average. An average clock offset of $690,04ms$ and a median clock offset of $472ms$.

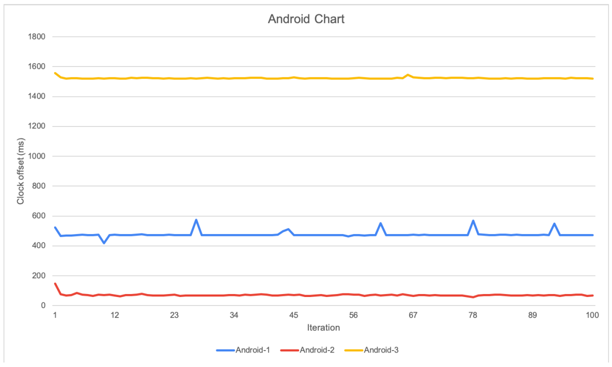

Figure 5: Clock offsets on Android systems

The average clock offset is similar to Windows, which could be explained by the amount of different hardware both operating systems support. On the other hand, Android is based on the Linux kernel [?] thus, one could have also expected results similar to the Ubuntu results. But, because Android is mainly focused on mobile devices, battery constraints could have played a big roll in the difference between the two.

4 Conclusion & Importance for Cat Caster

Although NTP is a standard protocol [?], the performance is far from standard. This could be explained by operating systems using compatible variations on this protocol such as the Windows W32t service [?], also different hardware and NTP servers could be to blame. For our implementation in the Cat Caster program, we do not need the clocks to be in sync with a public NTP server. Instead, we use our server as a sort of SNTP server with which the client can connect. This implementation uses the TCP protocol [?] rather than the UDP protocol. This means that due to the error checking properties of

TCP, the synchronization might be a bit slower due to reasons explained in the example of 3.3.3. But, because Cat Caster is only dependent on synchronization at some points of execution, this is not a big concern. When a client first connects, we synchronize the client every second which allows us to get a fairly accurate rough estimate of the clock offset compared to the server.
Finally, we wrote that we used `time.is` before starting our tests. Our test results have always been fairly similar to what `time.is` stated.

5 Alternative synchronization method

So far in this report we have only described a synchronization method that is based on synchronizing the local clock-times of the connected devices. Synchronizing tasks would then be performed **by sending a start time to the clients**. It was therefore important to investigate the accuracy of the local clock-times.

Our implementation of the synchronization task is independent of the clock time of the devices. Instead of calculating the offsets of the clocks one might calculate the offsets induced by the difference in the current times and the *socket-io* emit latency. Then instead of sending a start-time **the server would send a delay before performing the task** (like "start the countdown in 10 seconds" instead of "start the countdown at XX:XX:XX").

The goal of this section is to investigate by how much the calculated offsets using this method would differ through scientific testing.

5.1 Scientific testing

Without getting too much in detail about how socket.io emits work [2] and how a time offset is calculated through such a communication between a server and its clients, we mention that we implemented an extension to our code that calculates the offsets every second from the start of the creation of the slave for a duration of 5 minutes. Due to previous experience we expected the resulting curve to be a down-sloping curve starting with a very high offset due to the big load on the single-threaded JavaScript call stack (as explained earlier in this report) at the start of the process [3]. According to our assumptions this curve should then go down until reaching a stable state close to the "real" offset.

But after the first run of the test we got something very unexpected (see figure 6). Instead of getting a down-sloping curve we got a fluctuating curve.

[2]See https://socket.io/ for more details

[3]When a slave connects to the server a lot of initialization-processes are running. That is why we expected the stack to be very loaded at the beginning what would cause emits to take more time and therefore result into higher offsets.

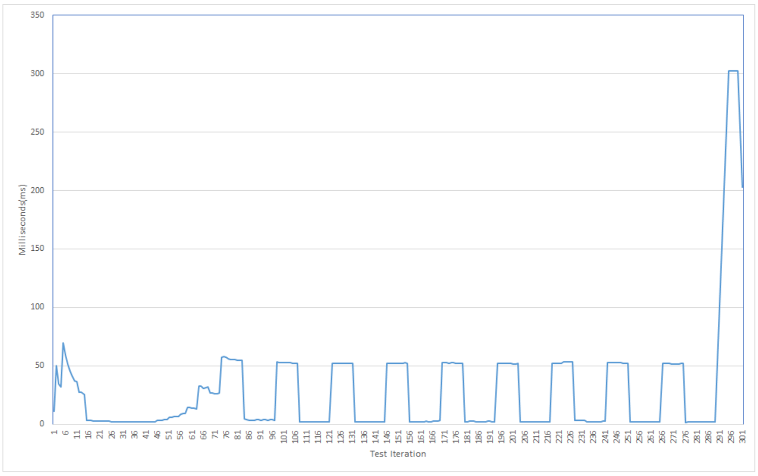

Figure 6: First test of the alternative method using a busy computer

After some thinking we assumed that the cause for this fluctuation was the fact that the laptop on which the test was run was busy doing some other work. After killing the significant processes that could slow down the communication process we got what we expected (see figure 7). After the 14th iteration the offset stabilized around 3ms.

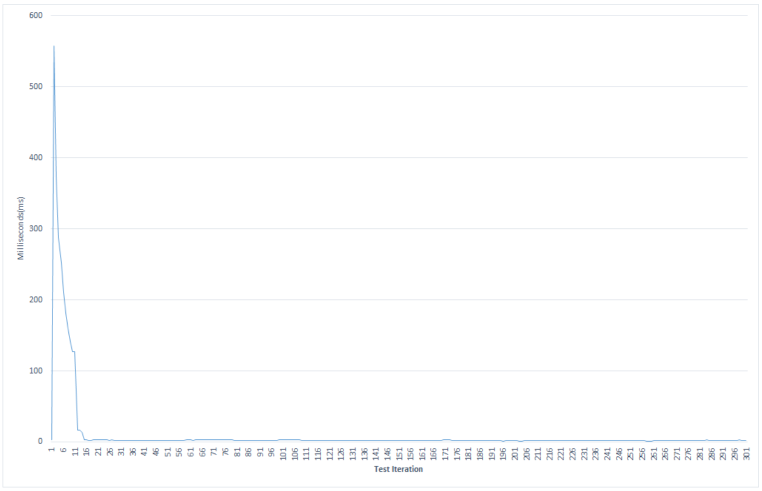

Figure 7: Second test of the alternative method with computer in rest

5.2 Conclusion

In order for this alternative method to work well the client should frequently ”re-synchronize” (by for example calculating the time-offsets with the server every second). Then when a synchronization task should be performed, only the last calculated offsets should be taken into consideration. This is better than the earlier method of synchronizing the clocks with an external ntp server due to the fluctuating differences seen and described in this report. Furthermore this would require manually calculation and saving of the clock offsets of every device before connecting that device to the server (as it is not possible to use UDP and as a result NTP in the browser). We therefore conclude that frequent re-synchronization through communication between client and server, by using the described sntp protocol, is a better method which guarantees synchronization at every moment instead of synchronizing the clocks of every connected device with an ntp server.