18-647 Connected Embedded Systems Lab 2
Clocks and Synchronization

Jun Ye
jun.ye@sv.cmu.edu
March 2, 2018

***Abstract*—For distributed systems, synchronizing time is an important task. This ensures correct communication across connected devices. This lab aims to help us understand time synchronization techniques better. It has three parts building up from each one, first building a clock, then synchronizing the clock using NTP and lastly measuring the synchronization performance. In the end, the local clock is able to synchronize with the server clock within 200 μs. The clock drift rate is 4.7 μs/s. The synchronization accuracy is affected by network condition. More specifically, the longer the delay is, the less precise the calculated offset becomes. Thus we should choose the offset when delay is small to synchronize the clock.**

I. Introduction

Time is an important aspect in distributed systems, because it is needed to synchronize communications across connected devices. Different timing precisions are required for different applications. In this lab, I first built a 1 Hz local clock and synchronized it wirelessly with the golden clock of the server using Network Time Protocol (NTP). Then I compared my synchronized clock accuracy with another synchronized clock. Phase alignment and clock drift rate are measured. Further improvement using delay and offset is mentioned in the end.

II. Related Work

The local device used NTP to synchronize with the server's clock. A NTP packet includes four timestamps, t1, t2, t3, and t4. T1 is the time when the local clocks sends the packet to the server. T2 is the time when server receives the packet. T3 is when server sends the packet to the local clock. T4 is when the local clock receives the packet from the server. From these four timestamps an offset of the time between the local clock and the server can be calculated, assuming the the round trip to and from the server is symmetrical:

$$offset = ((t2 - t1) + (t3 - t4))/2$$

The round trip delay is calculated as:

$$delay = ((t4 - t1) - (t3 - t2))/2$$

The timestamp includes a 32-bit field representing seconds and a 32-bit field representing fractions of a second. They represent seconds and fractions of a second since Jan 1st, 1900 [1].

III. Approach

This lab is separated into three parts. Part one is building a clock. Part two is synchronizing the local clock with the server using NTP. Part three is measuring synchronization.

In part one, I configured a 1 Hz clock using the Timer Counter Module 0(TC0) and channel 1. I used timer clock 4 which ran at 656.25 kHz. When the counter value Rc reached 656250, it reached one second and an interrupt was generated to increase the seconds counter. Another option is to use clock chaining to get the seconds value. The output of the TC0-CH1 can be the input of TC0-Ch0. Thus the TC0-Ch0 became a 1 Hz counter, and it saved CPU cycles without incrementing the seconds counter. In order to get microsecond accuracy, I had to change the clock because the clock is not fast enough running at 656.25 kHz. I used clock 3 which runs at 2.625 MHz. I got the μs reading from the reading the counter register *t->usec = (double)TC_ReadCV(TC0,1)/2.625*. I also need to divide the counter value by 2.625 to get microseconds.

In part two, I used NTP protocol to synchronize the local clock with the server's clock. A NTP helper class was given to us, so that we could extract the timestamps from the packets. On the local client side, it received 3 timestamps, t0, t1, and t2 from the packet. It recorded the time when the packet arrived as t3. The formula to calculate the offset and delay between the local machine and the server is described in section II. The offset and delay had two parts, the seconds part and the microseconds part. I added the offset second to the seconds counter by using a offset variable. Because seconds variable is updated in the interrupt handler, there can be race condition when we update seconds directly. For the μs part, I reset the Rc value by the offset amount in the interrupt handler and set it back immediately afterwards, so the Rc value was changed for only one cycle. It is the only safe place to change the Rc value to avoid setting new Rc value below the current Rc counter value which can cause the clock to run until the maximum value of the counter register is reached. Figuring out the μs offset amount was not as straightforward as the seconds part. I couldn't use the given formula, and I had to take the negative value into consideration and whether set the cycle longer or shorter based on the offset amount. Lastly, I need to multiply 2.625 to get the new Rc value *newRc = (uint32_t)((1000000 - offset->usec)*2.625)*.

In part three, I measured the synchronization results. I used NTP to synchronize every 10 seconds. I compared my clock with the reference clock on the oscilloscope to see the phase offset. After getting my clocked synchronized, I turned off the synchronization and measured the drift rate on the oscilloscope. I then measured the average offset and delay over a period of time.

During this lab, I had one bug that bothered me for the whole lab. Starting from part two, when I read t0, t1, t2 and

t3 and just printed them out without adding offset, after about 5 minutes, the reading of t0, t1 and t2 became the same. T3 was little different. An example output was shown in Figure 6 in section VII. I was very confused how t1 and t2 could be changed since they were read from the packet. I checked my offset calculation, gettime() and addoffset() functions. However, the problem still existed. Finally, I changed my time interval of doing NTP request from 5 seconds to 10 seconds, and the problem disappeared. Specific reason how the short interval caused the problem still needed further investigation.

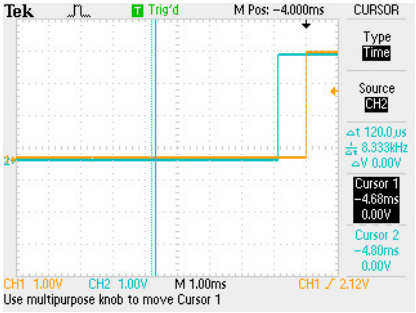

Fig. 1: Phase Alignment

IV. Experimental Results

For part three, the phase alignment varied with time. It changed between 500 μs to 3 ms. The average result I got when aligning with the reference clock is about 1 ms as shown in Figure 1. maybe the reference clock was not very accurate, even though my offset reading was at around 200 μs, from *SerialUSB.print()*. I measured the drift rate as 4.7 μs movement per second. Two example images are shown in Figure 2 and Figure 3. The total movement is 500 μs over 106 s, so the drift rate is 4.7 μs/s.

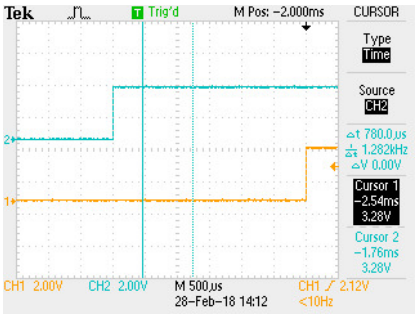

Fig. 2: Drift Measurement 1

For measuring the average offset and delay overtime, part of the data recordings are shown in Figure 5 in sectionVII due to the large size. A graph is plotted as shown in Figure 4. The data is recorded over 830 s. The absolute values of the offsets are used in the graph for better comparison with the delay.

V. Analysis

As we can see from Figure 4, the measured time period is 830 s. Offset and delay values have similar shape. Delay is mostly flat at around 2000 μs, but there are spikes coming

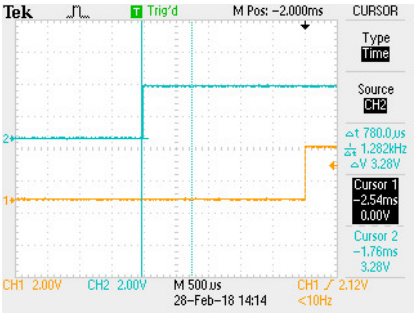

Fig. 3: Drift Measurement 2

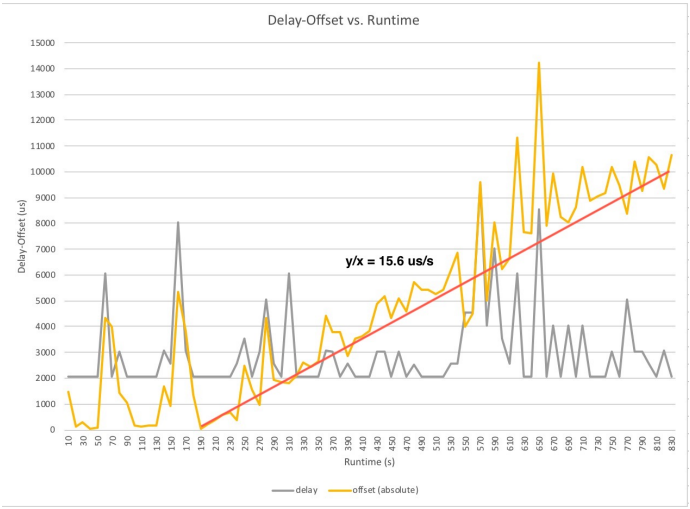

Fig. 4: Offset-delay Graph

up. Offset is mostly flat before 190, it's because of the NTP synchronization. After 190 s, I stopped NTP synchronization. We can see that the delay and offset are correlated from the spikes, when delay increases, offset also increases and vice versa. The spikes are caused by network conditions or TCP/IP buffering mechanism. We can also see that the offset keeps increasing after 190s at a rate of 15.6 μs/s. This relates to the drift problem. This article [2] shows a scatter plot between the offset and the delay, and we can see that they are proportionally related. This tells us that we should synchronize the clock when the delay is small, and thus the offset value would be more accurate.

VI. Conclusion

From this lab, I learned about building clocks running at different frequencies. I learned how to use NTP to synchronize time between the local clock and the server clock wirelessly. I got about 200 μs phase offset after synchronization. Network conditions can affect the synchronization accuracy. The longer the delay is, the less accurate the calculated offset becomes. So choosing when to synchronize time is important. Currently, I am just synchronizing time at a regular interval. For future work, a more precise algorithm can be implemented, such as the one described in [2]. This articles talks about using a buffer to store 8 delay values and choose the lowest value to use to calculate the offset.

References

[1] "Technical information - NTP Data Packet," https://www.meinbergglobal.com/english/info/ntp-packet.html, accessed: 2018-02-26.

[2] "Clock Filter Algorithm," https://www.eecis.udel.edu/~mills/ntp/html/filter.html, accessed: 2018-02-27.

VII. Appendix

time (s)	offset (us)	delay (us)
10	-1479	2045
20	147	2050
30	298	2047
40	36	2050
50	68	2051
60	4346	6049
70	-4016	2048
80	1417	3049
90	-1065	2047
100	162	2050
110	147	2050
120	168	2048
130	153	2050
140	1689	3054
150	-925	2548
160	5329	8046
170	-3811	3074
180	-1357	2048
190	33	2048
200	209	2046
210	362	2047
220	579	2048
230	684	2048
240	378	2549
250	2481	3550
260	1567	2046
270	986	3049
280	4336	5049
290	1944	2550
300	1850	2050
310	-1829	6045
320	2111	2048
330	2609	2046
340	2436	2050
350	2652	2046
360	4401	3057

Fig. 5: Offset-delay Table

To separate images.

offset: 0.00 s 2259.00 us
delay: 0.00 s 2056.00 us
t0: 1519957658s 996526us
t1: 1519957659s 842us
t2: 1519957659s 872us
t3: 1519957659s 669us
offset: 0.00 s -95.00 us
delay: 0.00 s 4056.00 us
t0: 1519957663s 996526us
t1: 1519957664s 487us
t2: 1519957664s 517us
t3: 1519957664s 4669us
offset: 0.00 s 1945.00 us
delay: 0.00 s 2055.00 us
t0: 1519957668s 996526us
t1: 1519957669s 527us
t2: 1519957669s 559us
t3: 1519957669s 669us
offset: 0.00 s -2071.00 us
delay: 0.00 s 2071.00 us
t0: 1519957673s 996526us
t1: 1519957673s 996526us
t2: 1519957673s 996526us
t3: 1519957674s 669us
offset: 0.00 s -2071.00 us
delay: 0.00 s 2071.00 us
t0: 1519957683s 996526us
t1: 1519957683s 996526us
t2: 1519957683s 996526us
t3: 1519957684s 669us
offset: 0.00 s -2571.00 us
delay: 0.00 s 2571.00 us
t0: 1519957693s 996526us
t1: 1519957693s 996526us
t2: 1519957693s 996526us
t3: 1519957694s 1669us
offset: 0.00 s -2571.00 us
delay: 0.00 s 2571.00 us

Fig. 6: Short Interval Bug