EXAMENSARBETE INOM DATATEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2021*

A study of methods to synchronize different sensors between two smartphones

En studie av metoder för att synkronisera olika sensorer mellan två mobiltelefoner

JOHN ABDULNOOR

RAMY GAWRIYEH

KTH
SKOLAN FÖR KEMI, BIOTEKNOLOGI OCH HÄLSA

A study of methods to synchronize different sensors between two smartphones

En studie av metoder för att synkronisera olika sensorer mellan två mobiltelefoner

John Abdulnoor

Ramy Gawriyeh

Examensarbete inom Datateknik,
Grundnivå, 15 hp
Handledare på KTH: Jonas Willén
Examinator: Ibrahim Orhan
TRITA-CBH-GRU-2021:230

KTH
Skolan för kemi, bioteknologi och hälsa
141 52 Flemingsberg, Sverige

Abstract

Obtaining data simultaneously from different sensors located on different mobile devices can be useful for applications such as sports and medicine. In order for the data from the different sensors to be combined for analysis, the mobile devices need to be time synchronized first. This paper presents an application that can be used to calculate the difference between the internal clocks of two android devices using a combination of the Cristian and Marzullo algorithms. Different methods to connect the devices over Wi-Fi as well as the internet are tested to determine the optimal method for clock synchronization. The paper also validates the synchronization by testing different sensors on two identical android smartphones. The results show that clock synchronization between two mobile devices can be achieved with a round-trip time of 2 milliseconds or less using Wi-Fi Direct. Validation of the synchronization shows that a delay of 7 milliseconds or less can be achieved between two sensors of the same type on two identical android smartphones. It also shows that the least achievable delay between sensors of different types is 16 milliseconds. The conclusion is that once two android smartphones' clocks are synchronized, only data from sensors of the same type can be combined, with the exception of the camera sensor. Further testing with more robust equipment is needed in order to eliminate human error which could possibly yield more desirable results.

Keywords

Sensors, android, smartphones, synchronization, internal clocks, Cristian's algorithm, Marzullo's algorithm, round-trip time, Wi-Fi Direct.

Sammanfattning

Att erhålla data från olika sensorer som finns på olika mobila enheter kan vara användbart inom exempelvis sport och medicin. För att data från de olika sensorerna ska kunna kombineras för analys måste de mobila enheterna tidssynkroniseras först. Denna rapport presenterar en applikation som kan användas för att beräkna skillnaden mellan de interna klockorna på två Android enheter med en kombination av Cristian- och Marzullo -algoritmerna. Olika metoder för att ansluta enheterna via både Wi-Fi och internet testas för att bestämma den optimala metoden för tidssynkronisering. Rapporten validerar också synkroniseringen genom att testa olika sensorer på två identiska Android -smartphones. Resultaten visar att klocksynkronisering mellan två mobila enheter kan uppnås med en round-trip time på 2 millisekunder eller mindre med Wi-Fi Direct. Validering av synkroniseringen visar att en fördröjning på 7 millisekunder eller mindre kan uppnås mellan två sensorer av samma typ på två identiska Android -smartphones. Det visar också att den minst möjliga fördröjningen mellan sensorer av olika typer är 16 millisekunder. Slutsatsen är att när två Android smartphones är tidssynkroniserade kan endast data från sensorer av samma typ kombineras, med undantag för kamerasensorn. Ytterligare tester med mer robust utrustning behövs för att eliminera mänskliga fel vilket kan möjligen ge mer önskvärda resultat.

Nyckelord
Sensorer, android, smartphones, synkronisering, interna klockor, Cristians algoritm, Marzullos algoritm, round-trip time, Wi-Fi Direct.

Acknowledgements

This paper is part of our thesis study within the field of computer engineering at KTH - Royal Institute of Technology in Flemingsberg.

We would like to thank our examiner Ibrahim Orhan for giving us the chance to perform this study and we also want to thank our supervisor Jonas Willén for helping us numerous times by providing clarifications, feedback and suggestions throughout the study.

Table of Contents

1 Introduction

This introductory chapter will present the problem definition, goals and scope of the project.

1.1 Problem statement

Modern smartphones are packed with many different sensors of different types that can be used in many useful scenarios. A sensor is a piece of hardware that can listen to a specific type of events in the surrounding physical space such as sound waves, light or movements. It usually can represent those events in the form of analog or digital data that can be stored or processed. However, being able to utilize two different sensors on two devices simultaneously can in many scenarios be very useful, and in order for that to be feasible, the two sensors most of the time need to overcome a simple yet fundamental limitation that all modern hardware has, and that is the lack of synchronization between internal clocks. Synchronizing clocks between two systems is necessary in order to achieve data fusion for sensory input coming from the different sensors.

1.2 Goals of the project

The goal is to perform a study about how accurately different combinations of sensors located on two different devices can be time synchronized wirelessly in order to make data fusion possible.

1.3 Scope of the project and delimitations

The project concerns itself with the problem of synchronizing two sensors on two different mobile devices which exist in close proximity on the same local network.

The two devices are allowed to move within the coverage area of the aforementioned local network and are to exist in an (Newtonian) absolute space and time, and thus are taken to only possess the ability to move (Or be moved) with speeds under which the effects of special relativity can completely be neglected. That is speeds under 240*10^6 km/s. Which is deemed more than sufficient for virtually all of the applications that are expected to make use of such study.

2 Theory and previous work

This chapter presents important concepts that play a major role in sensor synchronization, a number of algorithms that are used for synchronization as well as the API methods used in order to obtain data from the different sensors involved in this study. The chapter also covers previous work in the area of time synchronization.

2.1 Background

Sensory inputs provided by hardware sensors like accelerometers, cameras and microphones are being more and more used for a wide spectrum of applications. From fitness tracking to robotics as well as some medical applications. In order to obtain more robust data than what typically can be achieved using one sensor, two or more sensors must be used simultaneously. There are many scenarios that can benefit from using sensor combinations, like using motion sensors that are placed in different places in an athlete's body or filming the same incident using two cameras from two different angles in order to match the videos and create a 3D model. However, making data that is captured by different sensors useful requires synchronization [1] since in most cases, any kind of sensory input is a function of time and may change within the smallest time intervals. If the sensory input does not have synchronized timestamps, the sensors' data would in most cases be useless. At the same time, it is well established that we cannot rely on the hardware that we have today to track time in a well-synchronized manner. This is due to the fact that the internal clocks of modern hardware devices aren't precise enough to track time without drifting from each other [2]. Hence, a software solution is needed.

Although smartphones are not exempted from the time drift problems that stand-alone sensors encounter, they are packed with several software and hardware sensors that can cover many possible applications that can make use of multiple streams of sensory input without requiring extra hardware. They are also more widely used than standalone sensors, but since their clocks are not synchronized with millisecond accuracy by default, synchronization algorithms need to be implemented in order to allow combining data from the different sensors on the devices.

2.2 Time in distributed systems

Time can both be defined and implemented in several ways. For this study, we chose to define time as a continuous cumulative value that steadily increases regardless of any events relative or non-relative to it, a concept that is broadly known as physical time [3]. In distributed systems, however, physical time cannot generally be used to determine the order of a pair of events due to the fact that the clocks of the different systems cannot be synchronized perfectly. Therefore, if two events would have occured at a process in a distributed system, then the order of these events is determined through the way the process observes these events. In addition, if a message is sent between two or more processes, then the event of sending the message is said to always have happened before the event of receiving the message. This is called the happened-before relation according to Lamport [1978] [4]. Determining the correct order of events is the primary goal of distributed systems [38].

2.3 Clocks in distributed systems

According to the book *"Distributed Systems"* by George Coulouris [4], most if not all devices have their own internal clocks that in most cases rely on some sort of oscillator to provide information about the speed of time passing that would be used to increment the internal clock of the system. However, unlike the very expensive atomic clocks in GPS systems which are accurate to the one billionth of a second [5], the accuracy of these oscillators is not sufficient for it to be used as a synchronized time source. Thus, whenever there's an implementation where timing is to be known to the exact millisecond (or even 10 milliseconds), things become problematic. The reason for that is these oscillators are not possible to be manufactured to achieve the accuracy needed to have millisecond perfect timing without them becoming too expensive to be embedded in consumer electronics. Even if that was possible, these oscillators are still prone to effects that are very difficult to control such as the temperature of the circuit and mechanical vibrations [39]. As such, synchronizing two different clocks in two different devices always requires a written software solution and will continue to do so in the foreseeable future. That is because we lack the necessary technology to create hardware clocks that are capable of providing millisecond perfect timing at a reasonable cost. To better understand and to elaborate on the complexity of synchronizing time without relying on software algorithms, two important concepts ought to be mentioned. These concepts are offset and drift.

Offset and drift

According to the book *"Distributed Systems"* by George Coulouris et al. [4], offset is the initial difference in timing between two systems. Since there is no way to perfectly synchronize the clocks of two systems it would be impossible to have a starting point where two or more devices would have zero offsets.Hence offset is practically inevitable.

Drift, unlike offset, is the difference in timing between two or more devices with two different internal clocks as it accumulates over time. As mentioned above, we do not have hardware devices that would oscillate at the exact same rate especially under different conditions, and therefore drift in turn is also inevitable.

A hardware-level solution for the problem of clocks in distributed systems can be used to eliminate both the offset and drift. However, this is not achievable for personal devices as in most cases it costs more than the price of the personal device. As such, this study will try to use software algorithms in order to achieve a software solution that is similar to synchronized clocks in distributed systems. Similar to software solutions that try to compensate for hardware shortcomings, it would be impossible to achieve millisecond perfect synchronous timing. Thus this study will mainly do its best to achieve the best results and describe the achieved results in a quantifiable manner.

2.4 Clock synchronization in local networks

There are a variety of different algorithms for synchronizing clocks between two or more systems in a local network. This section covers two algorithms which are commonly used for that purpose.

2.4.1 Cristian's algorithm

According to the book *"Distributed Systems"* by George Coulouris et al. [4], Cristian's algorithm is used to synchronize time in computers externally by connecting them to a time server which uses UTC time. The algorithm works in the following way:

1. A process *p* at the client machine sends a message m_r requesting the time from server *S* at time T_0. Server *S* listens for the request made by the client and replies to it with a message m_t which contains the current time *t* of server *S*.
2. Once process *p* receives the reply from the server at time T_1 it can then calculate round-trip time T_{round} by subtracting T_0 from T_1 and then adjust it's clock according to the time of server *S* by adding the time *t* to half of the calculated round-trip time as per the following formula:

$$T_{Cristian} = T_{round}/2 + t$$

This algorithm works accurately assuming both devices are running on the same network. If the devices are not running on the same network, then further transmission delays are likely to occur which would result in the message taking a different time to be sent and received. This in turn would render the algorithm unusable. It assumes that the elapsed time before and after server *S* placed it's time in m_t is equal. The accuracy of the algorithm is $\pm\ (RTT/2\ -\ min)$ where *min* is the earliest point at which *S* could have placed its time in m_t after process *p* dispatched the request message m_r [4]. However, in a busy network, the higher the accuracy needed, the smaller the chance of it being achieved. This is due to the fact that the most accurate results will be those in which both m_r and m_t are transmitted in a time close to the minimum possible time *min*.

The one drawback of this algorithm is that it is dependent on a single server computer, which means that if that computer fails, synchronization will be temporarily impossible. Cristian had therefore suggested that the time be provided by a group of synchronized time servers instead of just a single server. Each of these servers would have a receiver for UTC time signals. That way, the client would be able to multicast it's request to the group of servers and then use the first reply it receives [4].

2.4.2 Marzullo's algorithm

Marzullo's algorithm is an agreement algorithm used to select sources for estimating accurate time from a number of noisy sources. The algorithm takes in a set of source intervals and returns the smallest subinterval that is shared by all the source intervals [15].

In this study, we used a variant of the algorithm where given a set of source intervals, it finds the largest interval that is contained in the largest number of source intervals. To clarify, if we had some intervals that overlap in smaller sub-intervals, then this algorithm will return the interval that is shared between the largest number of the given intervals. If we had 30 intervals at the start where there is a smaller interval shared by three, and another smaller interval shared by 7, and a smaller interval shared by 20 intervals, then the last mentioned interval is the one that is to be returned by the algorithm. For example, applying this algorithm as in *Figure 2.1* will return the interval [11, 12]. *Figure 2.2* shows a code snippet that represents the variant of the algorithm that was used:

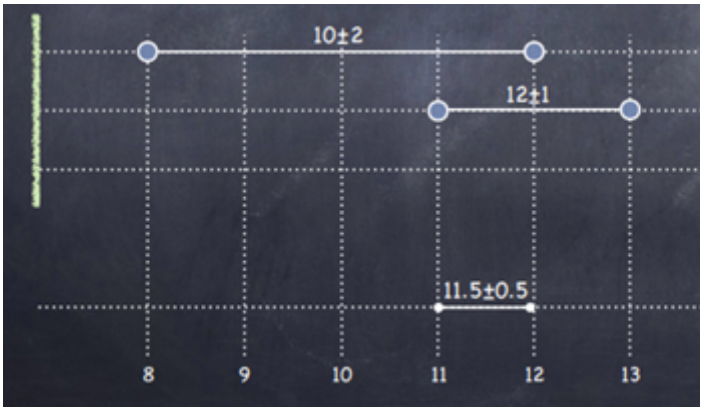

Figure 2.1 Marzullo's Algorithm

```
count=0, max=0
for all tuples<time[i],type[i]> {
        count = count + type[i]

        if(count>max) {
            max=count
            winStart=time[i]
            winEnd=time[i+1]
        }
}
return [winStart, winEnd]
```

Figure 2.2 Marzullo's Algorithm code snippet

The algorithm starts by storing all starting and ending points of the intervals in a set of tuples, where each tuple consists of a timestamp as well as the type of the timestamp. A starting timestamp is marked +1 and an ending timestamp with -1. The tuples are first sorted according to their timestamp while ignoring their type, then the algorithm loops through the sorted list and searches for the smallest interval that is shared by the largest number of the source intervals. Whenever the algorithm finds a smaller interval that is shared between a larger number of source intervals than the previously found one, it replaces it, so that at the end it returns the sub-interval that is shared by the largest number of source intervals.

There is another variant of this algorithm called the intersection algorithm, which returns the interval that is shared between all given intervals. This results in most cases in a larger returned interval, and in some cases much larger. Although this might reduce the accuracy it reduces jitter and allows for additional statistical algorithms to be applied.

2.5 Clock synchronization on the internet

Clock synchronization on the internet works differently than within the same local network due to the fact that messages between systems often have to travel long distances which introduces errors into the equation. In order to synchronize clocks between systems on the internet while minimizing error, the Network Time Protocol is used.

The Network Time Protocol

The Network Time Protocol is partly built upon the intersection algorithm mentioned previously. According to "*Distributed Systems"* [4] it "employs statistical techniques for the filtering of timing data and it discriminates between the quality of timing data from different servers". These servers are redundant, which means that they are reconfigurable and will continue to provide time synchronization even if one fails. Examples of such servers are time.windows.com for Microsoft Windows and time.euro.apple.com for Apple's MacOS.

The protocol consists of a hierarchy, called a synchronization subnet, which contains different servers that are synchronized with each other in 3 different ways depending on their level in the hierarchy. The levels are called strata and are represented by numbers starting with 1 at the lowest level. The lowest level is the most accurate and servers at stratum level 1 are directly connected to a UTC time source such as a GPS clock which provides the utmost accuracy. As the stratum number increases, the time accuracy decreases due to errors that occur at each level.

The NTP servers in the subnet synchronize with each other using 1 of the 3 methods as follows [4]:

1. The first method is called **multicast** which is used for server systems that are connected to each other via LAN where one or more servers multicasts the time to the other servers over LAN periodically where they can adjust their clocks according to the server and taking a small delay into account. This is the least accurate method but it is sufficient for most systems.
2. The second method is called **procedure-call** and it works similarly to Cristian's algorithm where one server accepts time requests from other systems and then replies with it's timestamp. The systems then adjust their clocks according to the server time minus the transmission delay. This method is more accurate than the multicast method and it can be used where more time accuracy is needed such as in file access systems as well as systems that don't have hardware support for multicast.
3. The final method is called **symmetric mode** which is used in the lowest stratum level servers as well as servers connected by LAN and it has the highest accuracy. It works by exchanging time information messages between a pair of stratum 1 level servers where the time data is kept and used over time to improve the synchronization accuracy between these servers.

It is also worth mentioning that systems that use NTP usually request timestamps from multiple servers in order to make sure that no anomalies occur in case the time from one server differs from other peers. This in turn helps the servers at the lower stratum levels to adjust peers accordingly where lower stratum servers can become higher stratum servers if they have transmission errors

Clock skew correction
There are 3 different actions that the client can take depending on the calculated clock skew [6]. The first is called **slewing** which means that the client will slightly slow down or speed up it's clock if the skew is less than 125ms. The clock will adjust by at most 500ppm (parts per million). The second action is called **stepping** which is taken when the skew is higher than 125ms but less than 1000ms indicating that the client should immediately step it's clock to match the server's clock. The final action is called **panic** mode which happens if the skew is larger than 1000ms, and here is where the client will not adjust the clock and human intervention will be required.

2.6 Previous work
There are a number of previous studies within the field of clock synchronization and smartphone sensors. These will be covered in this section.

2.6.1 Time synchronization and clock drift compensation in WPANs
The paper *"Performance Evaluation of Time Synchronization and Clock Drift Compensation in Wireless Personal Area Networks"* [7] aimed to find a way to achieve data fusion of different sensors detecting the movement of an athlete. In order to make the information that is received from the sensors usable, a means to synchronize these sensors is needed. To achieve that, the researchers needed to work around offset and drift. They solved the problem of offset by letting a master device handle all the calculations as opposed to synchronizing all sensors to a single clock.
The master stores a timestamp, t_1, and sends a synchronization request message, $m_{request}$ to a slave that hosts a sensor. After receiving $m_{request}$, the sensor node inserts its local time t_2 into the return message m_{reply} before transmitting it back to the master. The master generates timestamp t_3 when m_{reply} is received. *Figure 2.3* shows how a synchronization message is sent between a mobile phone and a sensor node. The master can, based on these timestamps, determine the offset between its own clock and the slave's clock, and accordingly synchronize the two clocks. When the master during consecutive data acquisition receives a sample from the slave, including its local timestamp, it adds the estimated offset to obtain the common synchronized time. The master performs all offset calculations and implements the synchronization, slaves are passive and simply respond to requests.

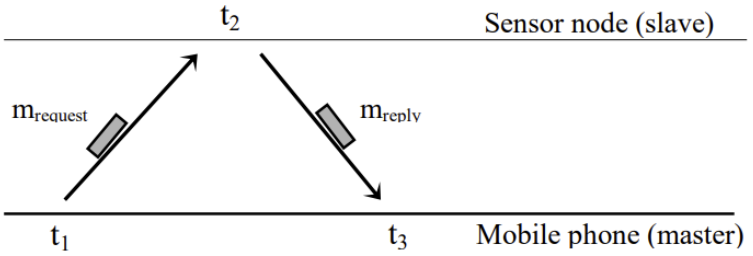

Figure 2.3 A synchronization message sent between a mobile phone and a sensor node, and the timestamps set by the master and slave.

In order to solve the problem of drift, the researchers decided to follow what they called a straightforward way to eliminate the drift parameter dt in the equation that represents a drift between two different clocks. The equation can be seen in *figure 2.4* on the next page.

$$t_{sensor1} = o + dt_{sensor2}$$

Figure 2.4 The relation between two nodes' local time

The technique they followed consisted of retrieving timestamps from two different sensors at different points in time and then calculating the value of the drift out of the equation set. In the case of multiple timestamps, the paper mentions that using linear regression is preferred. Once the value of drift is acquired, "drift compensation can be implemented either by multiplying the local node by the estimated drift parameter or by continuously adjusting the node's local time when necessary" [7] according to the paper.

The paper succeeded in allowing data fusion by achieving millisecond perfect synchronization. The results can be seen in *figure 2.5*.

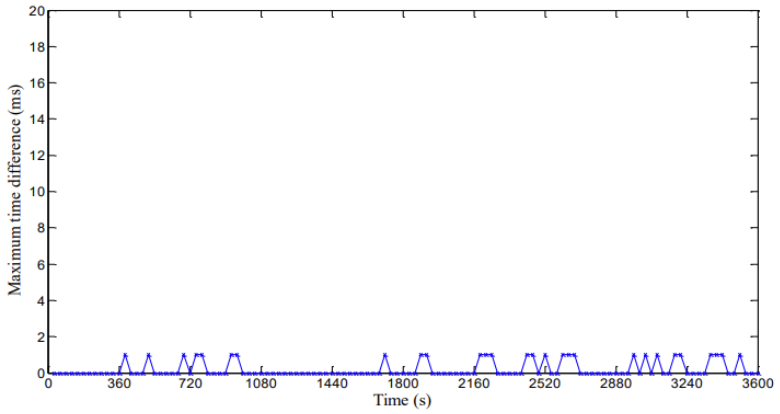

Figure 2.5 The maximum time difference in ms between the two most drifting sensors (slowest and fastest clock) with drift compensating function.

The communication technology used was ZigBee/IEEE 802.15.4 and the paper makes it clear that this result is specific to the aforementioned technology.

2.6.2 Time synchronization in Bluetooth piconets using mobile phones

The paper *"Local Time Synchronization in Bluetooth Piconets for Data Fusion Using Mobile Phones"* [8] studies the problems that arise while trying to achieve data fusion between mobile phones using Bluetooth. The paper does not touch the internal Bluetooth stack and interfaces with Bluetooth from the application layer. It uses the aforementioned Cristian's algorithm in order to estimate the offset between the clocks with the following formula:

$$o=(T_1-T_2)+T_{round}/2=T_1-T_2+(T_3-T_1)/2=(T_1-T_2+T_3-T_2)/2$$

T_1 and T_3 are timestamps set by the master, T_2 is a timestamp set by the slave, T_{round} is the estimated round-trip time (RTT). The estimated offset, *o*, between the clocks is the difference between T1 and T2 plus the delay between the readings of the clocks (half of the round-trip time). Cristian's algorithm also takes the clock drift rate into account, which can be compensated for by repeating the synchronization process with certain intervals. The paper states that the researchers have observed several performance issues with Bluetooth that make synchronization of data through an application in mobile phones complicated. The first problem that they discuss is sleep time where the device being a master, or a slave can delay sending of a message for the purpose of saving energy. The researchers tried using different Bluetooth dongles from two different vendors and they found that there were periods where no packets were sent which can be seen in *figure 2.6*.

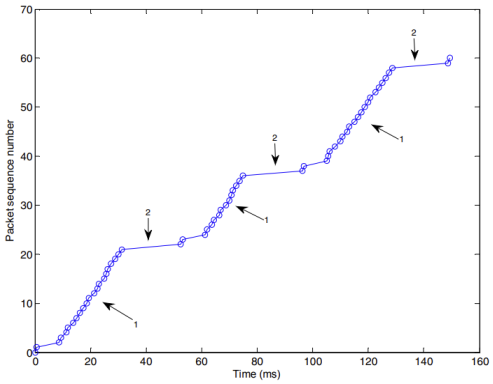

Figure 2.6 Packets sent between a master and a slave. The periods labeled 2 mean that no packets are sent.

Another problem that arises when using Bluetooth as a means to achieve time synchronization is the scheduling of multiple sensors. There the paper clarifies that more stochastic delays between the actual sampling times are present when the number of slaves that are active is higher.

The paper also discusses problems that arise while synchronizing mobile phones that are specific to mobile phones themselves and not the communication protocol used. The main problem that they mentioned is the stochastic delay between the sampling time on the sensor and the time when the sample is read by the application from the buffer on the mobile phone. They have found that the main contribution to that delay is waiting time in the outgoing buffers on the sensor. In order to combat this problem, the paper suggests a new approach where the sensors' clocks are synchronized to the master's clock locally using an application.

2.6.3 Exploiting smartphone peripherals for precise time synchronization

The paper *"Exploiting Smartphone Peripherals for Precise Time Synchronization"* [9] handles synchronization across many smartphones using different peripherals, namely Bluetooth Low Energy, Wi-Fi, and audio. It states that a one size synchronization solution "does not fit all" and tries to investigate the different ways to achieve synchronization by means of using different peripherals and tries to give an answer to which technique might suit certain applications more than others

The paper differentiates between two main synchronization techniques, receiver-to-receiver, R2R , and sender-to-receiver, S2R. Receiver-to-receiver is where all the devices listen to an external event that can be either opportunistically observed or intentionally generated in order to store the timestamp of the given event. After that, the devices share their own timestamp of the said event to all other devices that are meant to synchronize and then the offset of the internal clock is calculated based on that, knowing that that external event happens at the same physical time regardless of the different timestamps that the smartphones have. The paper notes that this technique might not be best used with sensors like proximity sensors cameras or ambient light sensors since the frequency of these sensors is usually between one hertz to 240 hertz which in practice means one reading each 4 to 10 milliseconds. Audio receiving sensors however have high sampling grades, and as such it was elected as a subsystem for the receiver-to-receiver implementation in the research.

The sender-to-receiver technique follows the master/slave approach where devices periodically ping a reference device in order to compute a relative clock offset.

The paper notes that when two smartphones are synchronized using a particular peripheral (Audio for their R2R implementation, and BLE for S2R), the clock difference fails to capture the true offset as it is affected by the peripheral timestamping delays.

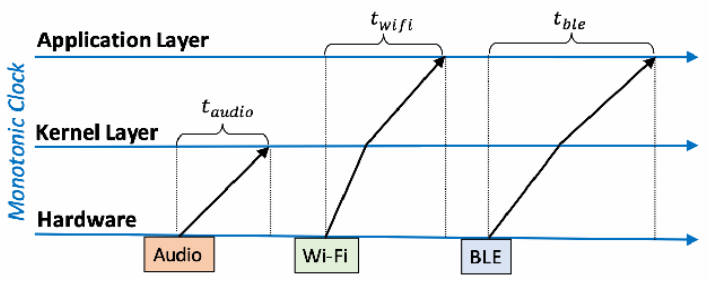

Figure 2.7 Timestamping events for audio, Wi-Fi and BLE peripherals in Android. Timestamp delays are not drawn to scale.

2.6.4 Precision smartphones sensors time synchronization

The paper *"Precision Smartphones Sensors Time Synchronization"* [10] deals with the problem of synchronizing several mobile phones with millisecond accuracy using an accelerometer. The researchers wanted to achieve data fusion for many sensors and used the synchronization achieved by an accelerometer for that purpose. Two devices were glued together, then external shaking was used as an external reference point for physical time. The glued devices were to capture the shakings at the very same time and the difference in timing of their internal clocks would then be calculated. The researchers found that they needed 5 to 7 seconds of shaking time in order to achieve optimal results. To find the time difference between the two devices, they used a numeric brute-force method with trapezoid rules.

The paper also discusses synchronizing more than two devices by synchronizing one of them to an external clock and then synchronizing the rest of the mobile phones to that device.
The paper mentions that despite using merely 20 acceleration records per second, they were able to achieve the desired accuracy in their implementation and the time between the devices was considered synchronized enough that it allowed for data fusion.

2.7 Sensors and timestamps

As there are different sensors in a smartphone, an API which consists of a collection of methods for each sensor type is required. One or more of those methods need to be used in a custom application which would invoke these methods and register a timestamp for the purpose of synchronization between devices. Custom methods to measure the quality of synchronization in milliseconds are also required since not all sensors provide useful timestamps in their respective APIs, as well as a means to assess how accurate timestamps on different sensors are.

2.7.1 The GPS sensor

The GPS sensor can calculate the coordinates of a smartphone providing it receives a signal from at least four different satellites. To get each fixed location, the device needs to solve an equation set with four variables, XYZ and time. The smartphone or any GPS receiver does not use its internal clock in order to determine the time in this equation set, since it isn't nearly as accurate as that of a GPS satellite system. It instead deals with time as an unknown variable in the equation set and calculates it algebraically [11][12]. As such, not only is the accuracy of the location dependent on the number of satellites that are available and the quality of their signal, but also the accuracy of the calculated time. When the error margin in location is big, then so is the error margin in time, as these variables' values are co-dependent in the equation set [11].

If we look at the GPS system specifications, we find that the satellites keep track of time with an accuracy of three nanoseconds [13], and since all electromagnetic waves travel at the speed of light, we can calculate that during 3ns, the signal travels one meter. As such, when we get a fix that is down to one-meter accuracy, we can safely say that the time accuracy of that fix is up to 3ns since the error margin of the equation set in that case is 1 meter. In reality, we might get a location fix that is only about 50 meters accurate. That corresponds to 150ns accuracy, and since we are dealing with milliseconds in this research, we can safely ignore the effect of GPS fix accuracy on timestamps that are calculated from GPS sensor readings.

There is also an error margin of 40ns that is specified by GPS specifications. It is stated that under 95% of the time, the accuracy is within 40ns. For smartphones however, this is only theoretical [12].

The following method is used in order to obtain a timestamp from the GPS:

Android [14]:

```
//get the device's location fix from the location listener
locationListener.onLocationChanged { location ->
//get the UTC time of this location fix in milliseconds since epoch (January
1, 1970)
gpsTime = location.time
}
```

Although the accuracy of the time calculated is down to a few hundreds of nanoseconds, there is no way of telling exactly when the timestamp provided by the GPS sensor was calculated [13]. As such, it would be meaningless to evaluate the accuracy of synchronization using GPS timestamps.

iOS [16]:

```
//get the device's location fix from the location listener
CLLocation* location = [[CLLocation alloc]
initWithLatitude:(CLLocationDegrees) 0.0 longitude:(CLLocationDegrees) 0.0];
//get the time at which this location was determined
NSDate* now = location.timestamp;
```

2.7.2 The IMU

Both Android and iOS allow for capturing the XYZ values of the three axes provided by the sensor through their native API. Both Android and Apple's motion sensor APIs include methods to access data from all types of hardware sensors. For this thesis study, the accelerometer will be used.

The following callback method is used to obtain sensor data when the sensor is used:

Android:

```
//get the x, y and z values as well as the timestamp from the sensor
sensorEventListener.onSensorChanged { event ->
//get the x value
x = event.values[0]
//get the y value
y = event.values[1]
//get the z value
z = event.values[2]
//get the timestamp
time = event.timestamp
}
```

The callback method onSensorChanged [17] is invoked whenever the sensor has new data, and the XYZ values will be stored in an array along with the timestamp of the device's up-time in nanoseconds. The provided timestamp value is useless for synchronization purposes since the devices' up-times will almost always be different. Therefore, obtaining a timestamp from the device is necessary here.

If the device is laying on a flat surface with the screen facing the sky, then the X value will change when the device is pushed towards the left or right. If it's pushed up or down, the Y value will change and if it's lifted up towards the sky then the Z value changes [18]

iOS [19]:

```
// Create a CMMotionManager instance
let manager = CMMotionManager()

// Read the most recent accelerometer value
x = manager.accelerometerData?.acceleration.x
y = manager.accelerometerData?.acceleration.y
z = manager.accelerometerData?.acceleration.z

//get the timestamp
time = manager.accelerometerData?.timestamp

// How frequently to read accelerometer updates, in seconds
manager.accelerometerUpdateInterval = 0.1

// Start accelerometer updates on a specific thread
manager.startAccelerometerUpdates(to: .main) { (data, error) in
    // Handle acceleration update
}
```

2.7.3 The camera sensor

The camera API contains a variety of tools that make use of the camera sensor. On android, the only way to obtain a timestamp is through the use of the CaptureResult class [36]. However, according to android documentation, the provided timestamp value is only useful for comparison on the same camera device. On iOS, a timestamp is only obtainable after an image is captured [37], which is not useful with a camera live view.

Therefore, additional code needs to be written for both android and iOS in order to analyze the current frame of the live view and obtain a timestamp from the device when the desired image properties are met. One way to do this is by analyzing the luminosity of the displayed live image and registering the system timestamp as soon as the sensor reports a high luminosity frame. This can be achieved using the following code:

Android [20][21]:

```
imageAnalyzer = ImageAnalysis.Builder()
   .build()
   .also {
       it.setAnalyzer(cameraExecutor, LuminosityAnalyzer { luma ->
           Log.d(TAG, "Average luminosity: $luma")
//get system timestamp when luminosity is high
           if(luma > 120) time = System.currentTimeMillis()
       })
```

The android CameraX API [20] provides an image analysis tool with the analyze callback method which returns the average luminosity of the current frame.

iOS [22][23]:

```
UIImage* image = [UIImage imageNamed:[NSString
stringWithFormat:@"%@%d.png",day,i]];
            unsigned char* pixels = [image rgbaPixels];
            double totalLuminance = 0.0;
            for(int p=0;p<image.size.width*image.size.height*4;p+=4)
            {
                  totalLuminance += pixels[p]*0.299 + pixels[p+1]*0.587 +
pixels[p+2]*0.114;
            }
            totalLuminance /= (image.size.width*image.size.height);
            totalLuminance /= 255.0;
            NSLog(@"%@ (%d) = %f",day,i,totalLuminance);
//get system timestamp when luminosity is high
if(totalLuminance > 0.7) time = CFAbsoluteTimeGetCurrent()
```

Apple does not provide an image analysis tool for luminosity, and therefore, custom code [22] like the one shown above is needed for that.

2.7.4 The microphone sensor

Both Android and Apple provide APIs to record audio using the microphone sensor. However, like in the case for the camera, there is no way to obtain a timestamp value while recording an audio sequence, therefore, a similar approach as the one mentioned for the camera sensor needs to be taken. A useful method that the APIs provide is getting the maximum amplitude of the sound wave during the recording. That way, a timestamp can be registered when a high amplitude is returned according to the following code:

Android [21][24]:

```
//start the recording
mediaRecorder.start()
//get the maximum amplitude while recording
amplitude = mediaRecorder.maxAmplitude
//register system timestamp when the amplitude value is high
if (amplitude > 7000) time = System.currentTimeMillis()
```

iOS [23][25]:

```
//start the recording
avAudioRecorder.record()
//Refresh the average and peak power values for all channels
avAudioRecorder.updateMeters()
//get the peak power of the sound while recording
power = avAudioRecorder.peakPower(channel)
//register system timestamp when the power value is close to 0dB
if(power > -20) time = CFAbsoluteTimeGetCurrent()
```

2.7.5 The touch sensor
The touch sensor is amongst the least complicated when it comes to obtaining a timestamp. When the device screen or a button on the screen is touched, the device can register it's timestamp using the following code:

Android [21][26]:

```
//the screen is touched
layoutView.setOnClickListener {
//get system timestamp
time = System.currentTimeMillis()
}
```

iOS [23][27]:

```
//the screen is touched
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)
{
//get system timestamp
time = CFAbsoluteTimeGetCurrent()
}
```

2.8 The Android ecosystem
For the purpose of this thesis study, the android ecosystem is utilized

2.8.1 Android OS
The Android operating system [28] has been owned and developed by Google LLC since 2005. It is an open-source operating system based on a modified Linux kernel. Development started in 2005 by a former company named Android, Inc. which was later bought by Google. The OS was primarily intended for use in touch screen devices such as smartphones and tablets. However, it was later implemented into other types of electronic devices such as televisions, cars and wristwatches. The OS is now very flexible and open-source enough that it could be implemented in any IoT device.

2.8.2 Android Studio
Android Studio [29] is the official development environment for the Android operating system, and it provides documentation for the required APIs to build applications as well as SDK tools within the software. This is the chosen programming platform for this thesis study in order to create the testing and synchronization applications for the different sensors. (do we need more info here?)

2.8.3 The kotlin programming language
Kotlin [30] is the predominant programming language used for developing android applications. It is a modern cross-platform, statically typed, general-purpose programming language with type inference. It is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library, but type inference allows its syntax to be more concise.

2.8.4 Extensible Markup Language
XML [31] is the markup language used to create the front-end part of the android program. Every activity in an android application is bundled with a XML file where all the elements such as texts, text fields and buttons are created and placed across the screen

2.9 Wireless connectivity methods

There are multiple ways to connect devices wirelessly, four of the most common connection methods were used in this study and are described below.

2.9.1 Wi-Fi Direct

WiFi-Direct [32] enables Wi-Fi devices to connect directly to each other without an intermediate access point. Using Google's Wi-Fi Direct APIs, devices that support Wi-Fi P2P can discover and connect to each other for the purpose of transferring data over a fast connection with a data rate of up to 250Mbps. This makes it much more convenient to use for bigger applications as well as an application which needs the fastest responses wirelessly.

2.9.2 Wi-Fi Local Only Hotspot

A Local Only hotspot [33] is a type of network that can only be created programmatically. It operates similarly to a traditional hotspot except that it does not have internet connection. It is mainly used for devices to communicate locally through android applications once they are connected to the network. When the user activates a local only hotspot, the network is created along with a reservation object which contains the SSID as well as the randomly generated password of the network. The hotspot can be requested by multiple applications at once, but every application can only make one request at a time. Applications can connect to a local only hotspot using a Network Specifier object, and once they are connected, they become the client while the application where the local only hotspot is created becomes the host.

2.9.3 Wireless LAN

Another way to connect two devices on the same network internally is using Wireless LAN [34] where the devices' IP-addresses are managed by the router on the local network. When a server is hosted on one of the devices, the second device can then connect directly to the server using its internal IP-address and act as a client. The internal IP-addresses can be obtained either by checking the devices' network configuration or through the router's administration page.

2.9.4 Wireless internet connectivity

When a client wants to connect directly to a specific device on another network, then the server device would need to set up port-forwarding [35] on it's network which would allow any external device trying to send data packets to it to be forwarded to that specific device's internal IP. This is done by specifying the host device's internal IP-address in the router's port-forwarding rule as well as the routing protocol and a port number or port range. The client would then need to enter the host's external IP-address as well as the specified port in order to connect to the host.

3 Method

In this chapter, the chosen methods, tools and frameworks used in order to connect two devices as well as calculate the delays between them are presented. In addition, the testing methodology for each sensor as well as different sensor combinations is described.

3.1 Research method

A literature study was performed, and based on that, we chose to test the synchronization of two phones over the four different wireless connectivity methods which are described in section 2.9. The synchronization combined Cristian's algorithm with Marzullo's algorithm in order to achieve the desired results as described in section 3.2. The hardware and software used for testing are described in section 3.3. Using the four wireless connectivity methods, the average round-trip time between 2 devices was calculated after a set of 10 tests was performed on each of the methods. Each set of these tests consisted of a different amount of messages sent between the devices. Section 3.4 covers the testing methodology for the connectivity methods. Once the best connectivity method was determined, the offset between the devices was calculated according to section 3.5 and then it was validated using different custom applications that were created to test the different sensors between 2 devices.

Applying the calculated offset to the client's clock programmatically was not possible since that requires root access which we did not have. As a result, we opted to simulate synchronization by adding the offset value to the client device's obtained timestamp in the different sensor apps at the time of testing. Section 3.6 describes the testing methodology for synchronizing the different sensors.

3.2 Choosing the right algorithms

In order to find a way to synchronize devices with high accuracy, NTP alone would first need to be insufficient for data fusion for data coming from sensors on two different devices. The aim is to achieve a delay of less than 10 milliseconds between the sensors on the different devices after clock synchronization. For that purpose, an app that measures the offset in time between two devices was developed. If the accuracy was in the tens of milliseconds or less with NTP as it is claimed [40][41], then one could say that NTP is already sufficient for data fusion for most applications. Thus, achieving synchronization using communication technologies like Wi-Fi Direct or Wi-Fi Hotspot would be unnecessary.

However, measuring offset between two different devices is a tricky task that cannot be achieved without relying on certain synchronization algorithms. This is because when a device sends the time of its internal clock to the other device, it would take the message an unidentifiable amount of time (measured in milliseconds) to reach the other device. One could rely on Cristian's algorithm alone and assume that the time it took the message to arrive is half of the round-trip time, however since there are uncontrollable factors like buffering and multi-threading that would delay the communication, the time it will take the message to arrive will vary each time. One could take the average of several attempts but this will include the delays caused by buffering and multi-threading. This is undesirable since it would reduce the accuracy from a few milliseconds to a few tens of milliseconds. As such, the Marzullo algorithm was implemented in conjunction with Cristian's algorithm in order to calculate the actual round trip time without extra delays that are not caused by mere communication.

There is still no guarantee that the interval is exactly as small as the round trip time, since there is always a chance that some delays are included. Even if there were a hundred intervals generated, we still cannot say with 100% certainty that the middle of that interval, i.e the average, is the exact offset between the two devices, however, one can say with absolute certainty that the true offset lies somewhere between the start and the end of the interval. As a result, we could say that the accuracy of the calculated offset is accurate down to the number of milliseconds represented by the width of the interval. For example, if we get the interval [180, 190] we would say that the offset is 185 milliseconds with an accuracy of 10 milliseconds.

3.3 Testbed

The hardware used for testing, debugging and synchronization are 3 mobile phones of which two are identical. The identical phone models are the Google Pixel 2 XL and the different device is the Huawei P10 Lite which is running an older version of android, 7.0. Both Pixel phones are running the latest version of android which is 11.0. Testing and synchronization is always performed between two devices simultaneously, but in order to check for anomalies, the third device is used for that purpose as well as to check if for example an older version of android also achieves similar results.

Connectivity and offset calculation application

In order to connect two android devices to each other as well as perform offset calculation between them, the application below was created.

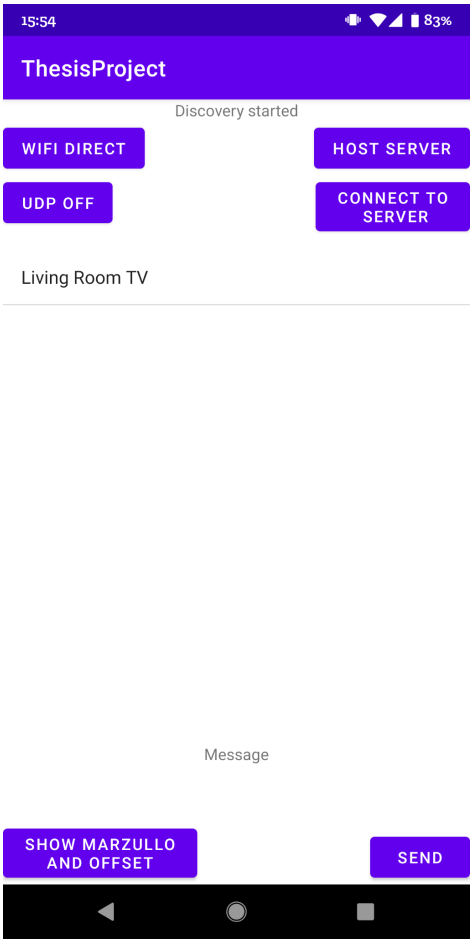

Figure 3.1 The Connectivity and Synchronization App

The application, which can be seen in the figure on the previous page, was made as simple as possible. It consists of a status message displaying connectivity status, a number of buttons, a ListView object which displays the list of available Wi-Fi Direct peers to connect to and finally another status message to display messages between server and client. The mentioned buttons perform the following actions when pressed:

- The WIFI Direct button enables discovery of other devices and the ability to connect to one of them using the list.
- The UDP Off button changes the network communication protocol to TCP. Only the UDP protocol was used throughout the testing.
- The Host Server button enables the device to host a server either using Wireless LAN or WiFi Local Only Hotspot.
- The Connect To Server Button allows a device to connect to a host device. Using this button, the device is able to connect to the other device via Wireless LAN or Local Only hotspot on the same network or Wirelessly from another network through port forwarding.
- The Send button can be pressed to send time requests once the client and the server are connected.
- The Show Marzullo and Offset button displays the results of applying marzullo's algorithm to an array of Round-Trip time intervals once the user has chosen the number of time requests it wants to send and receive from the server. The calculated offset between the server and the client clocks is also displayed in milliseconds under the marzullo interval.

3.4 Testing the connectivity methods

In order to determine the connectivity method with the least delay, the average round-trip time was calculated by sending and receiving 3 different amounts of messages between the client and server using each of the four methods. First, 10 messages were sent and received, then 20 messages and lastly 100 messages. This was done 10 times for each of the 3 message amounts. Marzullo's algorithm was applied to an array containing the time intervals of the chosen amount of messages. The result for the 100 messages was the determining factor for choosing the best connectivity method.

3.5 Obtaining the offset

When the client is connected to the server, the offset between the devices is obtained by pressing the Send button in the app on the client side and choosing the amount of requests to send and receive. There is a 200ms pause after sending each request in order to allow for longer responses to arrive. The offset is calculated in the following way:

1. When the client chooses the amount of requests, it's system time is stored in a global variable just before the send method of the DatagramSocket object is called.
2. Once the receive method is called, the server checks whether the received buffer is not empty in which case it registers it's system time in a variable right before sending it back to the client.
3. The client will then register it's timestamp when it has received the server's response and an interval will then be formed and returned by the client for it to be stored in the array of intervals.
4. Once the specified amount of responses is received, Marzullo's algorithm is applied onto the intervals' array and the device offset will also be obtained by taking the sum of the start and end of the marzullo interval and dividing the result by 2.

5. In order to calculate the average offset, 10 sets of tests were done over a period of 24 hours. Each set involved sending and receiving 10, 30 as well as 100 time requests (i.e 10, 30, 100 loops). The average and the standard deviation was then calculated from the obtained results.

3.6 Sensor synchronization methodology

In order to validate the calculated offset between 2 devices, the different sensors were tested against each other as well as other sensor combinations. The validation testing was done on the identical Pixel devices in order to eliminate errors produced by using different hardware. Each test on the identical and the different sensors was done 15 times. The methods used for testing are described below.

3.6.1 Synchronizing identical sensors

Prior to all testing, the offset was first measured between the devices before starting the tests. The offset would then be subtracted from the absolute value of the difference between the device timestamps in order to simulate synchronization.

GPS

A simple android application was created which continuously listens for changes in device location. When a button in the app is pressed, a timestamp from the GPS API is obtained using the method shown in the previous chapter. The testing of this sensor was only done to compare the time offset between the devices' clocks and the GPS clock and therefore cannot be tested against other sensors due inconsistent results as described in chapter 2.

IMU

An application was created to test this sensor. The Java Virtual Machine's System.CurrentTimeMillis method was used to obtain the device time whenever the accelerometer detected significant motion in either the X, Y or Z axes which would be reported by the onSensorChanged callback method as described in the previous chapter.

In order for the test to provide a correct result when two devices were tested against each other, they were temporarily glued together and then vertically dropped onto a surface which resulted in a drastic change in the Y value of the accelerometer data. That way, the timestamps registered could be compared between the devices for parity.

Camera

Using the image analysis tool mentioned in chapter 2, a camera application was created which registered the device's time whenever the luminosity analyzer returned a high value which implied that it was a bright light source. Using that method, the two device cameras were positioned towards a computer screen with a black background while running the camera application with the analyzer tool. A white image was then displayed on the computer screen which yielded a high luminosity value and a timestamp on both devices was registered.

Microphone

The concept of testing here was similar to when using the camera sensor. The 2 devices were placed next to an external speaker. Using the created application, a recording was started when a button was pressed on both devices. During the recording, a loud sound was played on the external speaker and the microphone sensors on the 2 devices detected a spike in the sound waves produced with the help of the maxAmplitude method mentioned in chapter 2. When that method returned a high value, the devices recorded their timestamps.

Touch screen
Using the method onClickListener, a simple application was written which called this method when the devices' screens were touched. The testing process involved touching both devices' screens at the same time, which was achieved using a U-shaped tool. Once the screens were touched, they registered their timestamps.

3.6.2 Testing sensor combinations
There were 6 sensor combinations to be tested in total, and while it was impossible to eliminate human factors from the tests, the following methods were used in order to achieve as close to synchronized results as possible:

1. For the camera against the microphone, one of the devices had the camera application running while being held up by a mobile stand. The second device had the audio recording application running while sitting next to an external speaker. A video file which contained a 1 second loud noise along with a bright white background was then played. The camera sensor then picked up the high luminosity from the image and registered it's timestamp at that moment while the audio recorder also registered it's timestamp when it detected the loud noise from the audio of the file.
2. For the camera against the IMU, the created accelerometer application was modified so that the screen would turn white as soon as a motion in the Y access was detected. The device timestamp was also registered at the same time. The second device had the camera application running while being held up by a stand as in the previous case. It then registered its own timestamp as soon as it detected the white screen of the other device when it was dropped vertically.
3. For the camera against the Touch sensor, the testing method was identical to the previous case except that the screen was touched on the device that had the touch application running and it's screen background turned white while registering it's timestamp.
4. For the microphone against the IMU, both devices were temporarily glued together and hung by a thread while both testing applications were running. The device running the IMU application was then pushed against the second device using the middle knuckle on the index finger which resulted in the accelerometer to register a strong motion in the Z axis while the microphone sensor on the second device picked up the loud noise from the knuckle.
5. For the microphone against the touch sensor, the exact same methodology as the previous point was used except with the touch application instead of the IMU.
6. The final sensor combination to be tested was the Touch sensor against the IMU which involved a similar testing method as the Microphone and IMU except that the finger was used for the touch app instead of the knuckle. The timestamps on both devices were registered when a more aggressive touch was applied to the device running the touch application which caused the devices to sway and the IMU sensor on the second device to detect a movement in the Z-axis.

4. Results

This chapter covers the results achieved when testing the four different wireless connectivity methods between 2 devices as well as sensor synchronization results.

4.1 Connectivity method results

When it comes to the wireless connectivity methods between the devices, the fastest method observed was Wi-Fi Direct as shown in the chart below. After performing 10 tests with the different amount of message requests, the average round-trip time was just 1.5ms between 2 identical devices after applying Marzullo's algorithm. Hence, Wi-Fi direct was chosen as the connectivity method to calculate the offset between devices moving forward.

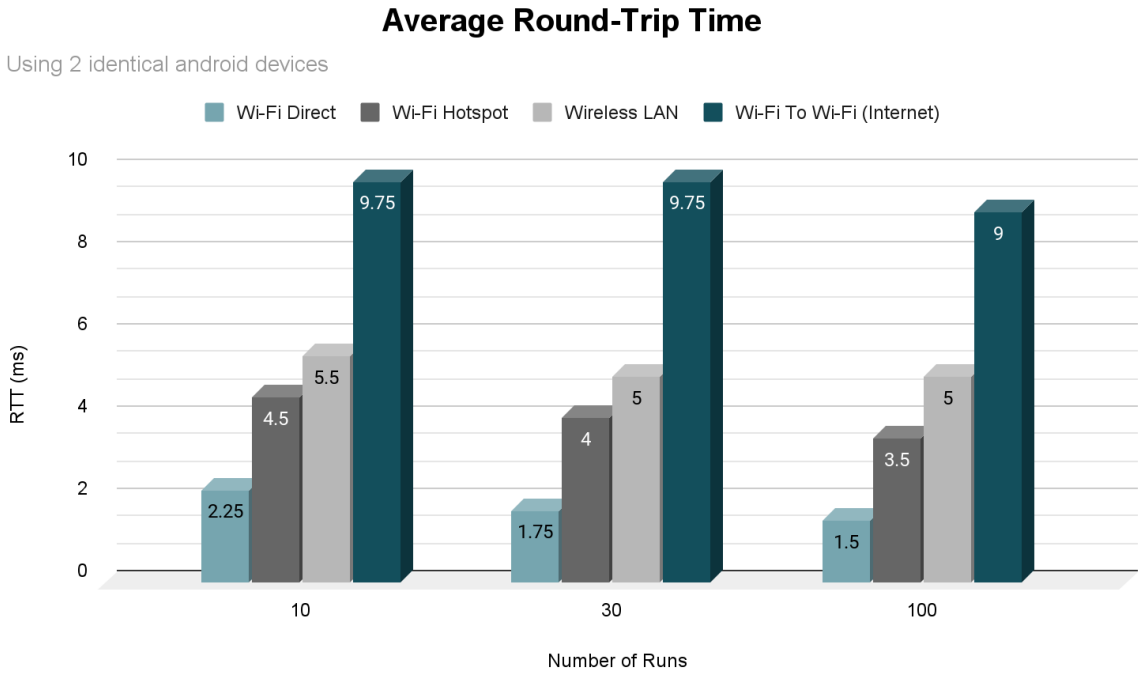

Figure 4.1 Round-Trip Time using 4 different wireless connectivity methods

In the case of Wi-Fi Direct and Wi-Fi hotspot the devices were immediately next to each other. In the case of wireless LAN and WAN, the two devices were 2 meters away from the router with no obstacles in between. It is worth mentioning that in every single testing scenario, Wi-Fi Direct consistently had a shorter round-trip time compared to all other communication technologies. This led to Wi-Fi Direct being the chosen method for offset calculation between the devices.

Table 4.1 on the following page shows the standard deviations in milliseconds of the round-trip time after running 10 tests on each connectivity method as described in the previous chapter:

Table 4.1 Standard deviation of the round-trip time in milliseconds after running 10 tests for each connectivity method

	10 Requests	30 Requests	100 Requests
Wi-Fi Direct	0.15ms	0.12ms	0.1ms
Wi-Fi Hotspot	0.31ms	0.23ms	0.21ms
Wireless LAN	0.56ms	0.41ms	0.38ms
Internet	1.5ms	1.17ms	1.1ms

4.2 Clock offset and drift

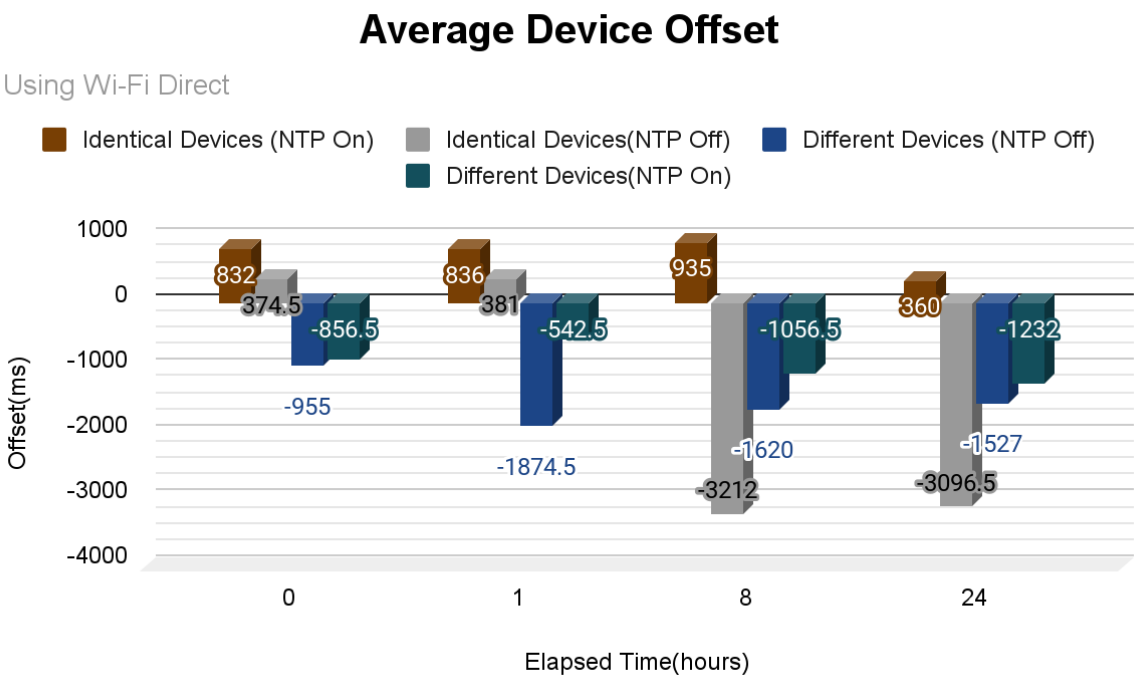

Figure 4.2 Average clock offset between devices with NTP on and off

With NTP synchronization on, which is normally the case with smartphones, the maximum offset observed between identical devices was around 1 second for identical devices (as per the graph in *figure 4.2* above), and 2.2 seconds for different devices. The devices were observed over a 3 days period. Putting the devices next to each other while looking at the digital clock, the offset between the two devices was as far as the human eye could tell exactly as long as the software calculated. The delay between the clocks could be seen as they were ticking (see *figure 4.3* on the next page).

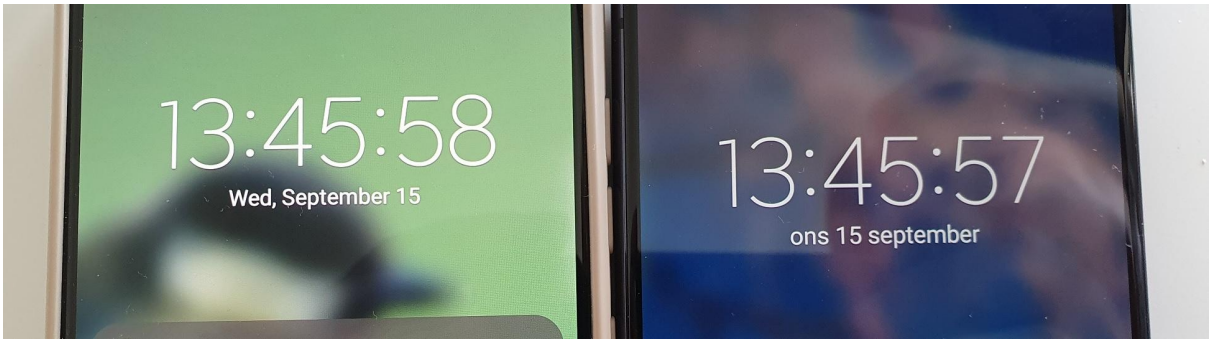

Figure 4.3 Clock offset of the digital clock between identical devices

After turning NTP off, the amount of offset observed between the 2 devices started at a low number but then kept increasing over time, exceeding 3 seconds after 24 hours. Using different devices saw a maximum difference of around 1.9 seconds between the devices.

The drift was not consistent with NTP off, even between similar devices. Not only did the offsets not drift linearly, they also did not drift consistently towards higher or lower values. Although most of the time the offsets did consistently drift either towards higher or lower values, but on certain occasions like in the chart above, they did not adhere to that role. After testing 3 different devices repeatedly with the same testing methodology described in chapter 3, an average of 700 milliseconds of offset (STD = 300) between two different devices was calculated. The lowest recorded was 70 milliseconds on the identical devices and the highest was 2700 milliseconds.

Table 4.2 below shows the standard deviations in milliseconds of the offset between identical and different devices:

Table 4.2 Standard deviation of the offset between identical and different devices with NTP on and off during one day

	At the start	**After 1 hour**	**After 8 hours**	**After 24 hours**
NTP On (Identical Devices)	166ms	168ms	189ms	150ms
NTP On (Different Devices)	237ms	201ms	285ms	300ms
NTP Off (Identical Devices)	305ms	316ms	420ms	409ms
NTP Off (Different Devices)	413ms	661ms	535ms	578ms

4.3 Sensor synchronization results

In order to get the most consistency and optimal results while testing the different sensors, the testing and simulated synchronization was done only on the two identical devices as mentioned in the previous chapter.

4.3.1 Identical sensors

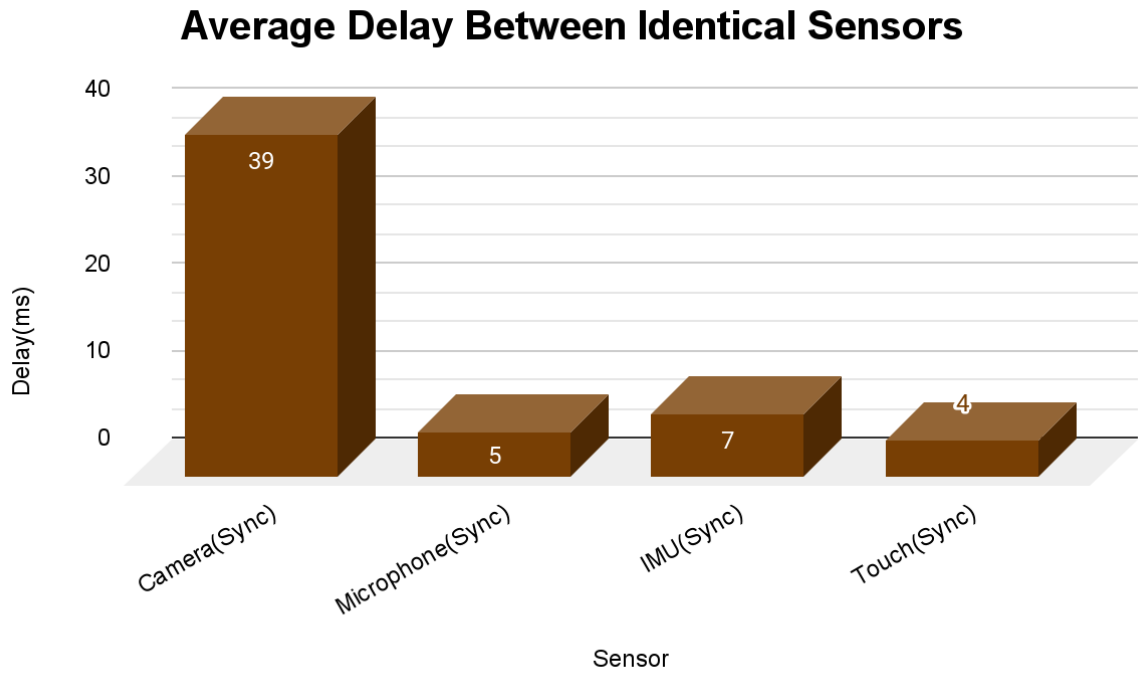

Figure 4.4 Average offset between the same sensor types of 2 identical smartphones

Firstly, sensors of the same type were tested between the devices. The sensors tested were GPS, camera, microphone, IMU (Accelerometer) as well as the touch sensor.

GPS

The results of GPS testing were inconsistent due to the fact that the time the GPS provides is anywhere within the last second, and therefore, the sensor was only tested to observe the difference between GPS time and device time, which showed that there indeed was a considerable delay between the GPS provided time and the device time even though NTP synchronization was turned on. However, the results showed a different offset between the GPS and the device, possibly due to the fact that the devices don't synchronize with NTP too often, resulting in an offset that can vary from 100ms all the way to 2000ms according to the test results. Therefore, the results for GPS testing were not included in the results' chart.

Camera

Testing the camera sensor on both devices at the same time, the result observed was 331ms in delay between the 2 devices without taking the offset into account. With the calculated offset subtracted from the result, it measured at 39ms.

Microphone
The delay observed while recording using the microphone sensors was 362ms without the offset taken into consideration. After subtracting the offset from the initial result, the delay becomes only 5ms.

IMU
The linear acceleration in the z-axis on both devices was measured for this test. A result of 416ms was observed which translates to 7ms after removing the calculated offset.

Touch
This was the most responsive sensor according to the test results. The resulting number was 4ms when the offset was removed from the initial delay.

Table 4.3A below shows the standard deviation in milliseconds of the offset between the identical sensors after simulated synchronization:

Table 4.3A Standard deviation of the offset between identical sensors after simulated synchronization

Sensor	Standard Deviation
Camera	2ms
Microphone	0.7ms
IMU	1ms
Touch	0.66ms

4.3.2 Sensor combinations

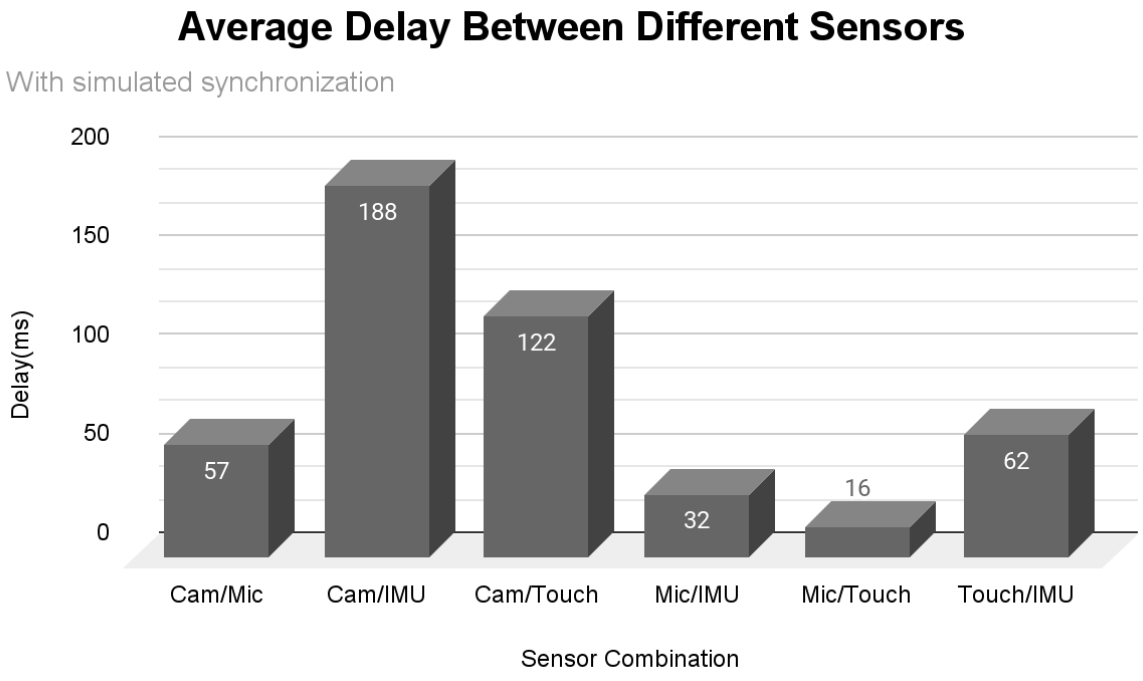

Figure 4.5 Average offset between different sensor combinations of 2 identical smartphones

When it comes to testing different sensor combinations, the results were more varied. All results shown above are with the offset taken into account. As *figure 4.5* shows, the highest average delay observed was between the camera and the IMU sensors at 188ms despite numerous attempts to reduce the delay by repeating the tests. This implies that the camera sensor is most likely the bottleneck since all results that involve the camera sensor have a somewhat higher delay. The lowest delay of 16ms was between the microphone and the touch sensors.

Table 4.3B on the next page shows the standard deviation of the offset between the sensor combinations after simulated synchronization between identical devices:

Table 4.3B Standard deviation of the offset between sensor combinations after simulated synchronization between identical devices

Sensor Combination	**Standard Deviation**
Camera/Microphone	10ms
Camera/IMU	26ms
Camera/Touch	15ms
Microphone/IMU	6ms
Microphone/Touch	3ms
Touch/IMU	10ms

5. Discussion

In this chapter, the test results of the offset and drift with and without using NTP are discussed as well as synchronization using wireless connectivity. An insight into sustainability for the environment is also provided.

5.1 Offset and drift

Surprisingly, the offset in internal clocks between two mobile devices synchronizing NTP protocol was often as high as 1 second (700 was the mean with a value of 300 as the standard deviation). This meant that achieving data fusion was practically impossible if the NTP protocol was the only measure to rely on. As expected, turning NTP synchronization off did not make things any better. Due to drift, the offset increased to a value near 5 seconds within 24 hours after turning NTP synchronization off.

The drift was not consistent without NTP, even between similar devices. Not only did the offsets not drift linearly, they also did not drift consistently towards higher or lower values. This isn't particularly surprising since the drift is known to be affected by many uncontrollable factors, such as temperature and mechanical vibrations aside from the properties of the oscillator itself. It did, however, counter the intuition that the effect those external factors have on drift would be too small to be able to shift the difference in offset from increasing to decreasing or vice versa.

5.2 Wi-FI synchronization

Prior to testing the different Wi-Fi connectivity methods, we anticipated that Wi-Fi Direct would have the least delays thanks to the very fast and responsive peer to peer connection that it provides. It was not evident to us whether Hotspot or Wireless LAN would take second place while connecting using the internet was deemed to be the slowest. The results matched our preconceived expectations in terms of overhead the technologies had. Wi-Fi Direct showed the least amount of overhead, achieving synchronization with accuracy of 1.5 milliseconds and a standard deviation of 0.1ms for 10 tests. Such accuracy is not only sufficient for data fusion in most applications, but it also is in some cases better than what is physically possible by some sensors, like the camera which captures a new frame each 33 milliseconds in the devices we used. It is unclear how achieving more accurate synchronization between the two devices, if possible, might positively affect the synchronization accuracy between the sensors.

5.3 Testbed

The similar devices we had access to were two Google Pixel 2 devices which are fairly old models. It is unclear if two more modern identical devices would synchronize more accurately, whether it is with using NTP, or with our application that implemented the Cristian's algorithm and the Manzullo's algorithm. However, we are confident that NTP alone would not have been sufficient for data fusion even with the most modern devices.

Although we used both similar and different devices for testing in order to account for differences in hardware between the two devices, it still remains a very small pool of devices. It is unclear how the results might have changed with other devices. It was also not possible for us to calculate any standard deviation for how the results might have varied between a larger number of different devices.

All three devices used for the majority of this study were Android devices, using different operating systems will most likely have affected the results as well.

5.4 Sensor synchronization

As the two systems were synchronized almost down to two milliseconds, we had the chance to test the sensors against each other where the difference in the internal clock of the two devices was almost no longer a factor. As it is shown in the results, virtually all the sensors could be synchronized so that the difference between them would be down to seven milliseconds, with the exception of the camera where it was 39. Although 39 milliseconds is relatively high compared to the rest of the values, it is practically a one frame difference in terms of the camera sensor. If the frame is delayed by e.g. 5 milliseconds, then the sensor is going to have to wait for the next frame, and as a result, the delay will be registered as 39 milliseconds. However, aside from the camera, one can draw the conclusion that data fusion between two sensors located on two different devices can in fact be achieved as long as it is two sensors of the same type, as the difference was 7 milliseconds at most.

Testing sensors of different types did not provide optimal results. The lowest that two different sensors could be synchronized down to was 16 milliseconds, which was the case of the microphone and touch screen. The offset can on average be as high as 188ms in the case of the camera and IMU. The maximum allowed offsets between two sensors in order to achieve data fusion depends on the application. However, unlike in the case of two similar sensors, data fusion can presumably be achieved for a small pool of applications that are not very sensitive to higher offsets if two different sensor types were used. In addition there is a considerable margin of human error in our testing methods, as we suspect that in the presence of advanced equipment and more specialized software the offset would be much smaller than in our experiments.

5.5 Economic, social, ethical and environmental aspects

As this work shows that internal clocks of mobile devices can be synchronized accurately enough to allow for data fusion in many cases, it is expected to contribute to decreasing production of new standalone sensors. Standalone sensors are usually considerably larger than the compact ones found in mobile devices. The ones that are found in mobile devices can be just as accurate for many applications, and they also already exist in almost every modern smartphone. Showing that these embedded sensors can be used instead of external sensors where achieving data fusion is needed could prompt many researchers around the world to experiment and do research without necessarily buying separate sensors, or at least not as many.

Being able to use embedded sensors in smartphones also means that research expenses for projects that use synchronized sensors can be decreased, allowing for the budget to be spent on other things. It should in turn also mean less strain on Earth's resources, especially considering that stand-alone sensors are usually provided with internal batteries that include some rare elements like lithium and cobalt.

Social aspects are non-applicable to the work done in this thesis, because the use of synchronized embedded sensors in mobile phones has no direct effect on any societal activity or problem. One possible aspect is the decrease in the demand for cobalt and lithium which in some cases are sourced through unethical labor practices in places that do not have strict laws that prevent such practices.

6. Conclusion

This thesis has presented a way of synchronizing two mobile devices using Cristian and Marzullo's algorithms and then validating the result using different sensors on these devices. The results showed that synchronizing the sensors with accuracy that is high enough to achieve data fusion is possible which was the main goal of this study. The testing showed that the two devices can, when using Wi-Fi direct, be synchronized with down to 1.5 milliseconds accuracy. In the case of two similar sensors on two different devices, the thesis showed that data fusion is possible for many applications, as the mean value was equal to or less than 7 milliseconds in most cases with the only exception being the camera sensor. Trying to synchronize two different sensors on two different devices proved to be more problematic as it had much lower synchronization accuracy, which is insufficient to achieve data fusion for many applications. However, as the testing methods used lacked highly specialized equipment, better results when combining different sensor types might be achievable if such equipment were to be available.

6.1 Future work

As the entire project was done on Android devices using the Kotlin programming language, it remains to be answered which results one might achieve if other combinations of software and hardware were to be used. Kotlin, unlike Java, does not rely on Virtual machines. With all the extra overheads virtual machines introduce, it would be interesting to know how overheads could change the results. Although the result in that case is expected to be worse, in other words, less accurate, it remains difficult to say with certainty unless tested.

As Android is not available on iPhones, there isn't a means to test the quality of the synchronization on Android devices using the iOS operating system. Therefore, the difference of synchronization quality between iOS and Android cannot be determined purely from a software point of view. Considering how different iOS and Android are, it would make for a very interesting subject to perform the same study on iOS devices. The intuition here is that better results would be achieved (especially when using different sensor combinations) since newer iPhones have faster processors and also the fact that iOS has the reputation of having fewer overheads than Android as an operating system.

In addition to Kotlin and Java, Android applications can be developed using React Native, which has its own set of APIs. It would also be very interesting to see the difference in synchronization accuracy not only between Java/Kotlin and React Native but also between iPhones and Android devices as React Native can be used to write apps for both systems.

References

[1] Galar, D. and Kumar, U., 2021. *Sensor Fusion*. [online] sciencedirect. Available at: <https://www.sciencedirect.com/topics/engineering/sensor-fusion> [Accessed 17 September 2021]..

[2] University of Delaware. 2021. *How do Computer Clocks work?*. [online] Available at: <https://www.eecis.udel.edu/~ntp/ntpfaq/NTP-s-sw-clocks.htm> [Accessed 18 September 2021].

[3] Sasso, D., 2021. *Physical Time and Time Clock*. [online] ResearchGate. Available at: <https://www.researchgate.net/publication/328879984_Physical_Time_and_Time_Clock> [Accessed 18 September 2021].

[4] Coulouris, G., Dollimore, J., Kindberg, T. and Blair, G., 2012. *Distributed systems*. 5th ed. Boston: Addison-Wesley.

[5] NASA. 2021. *What Is an Atomic Clock?*. [online] Available at: <https://www.nasa.gov/feature/jpl/what-is-an-atomic-clock> [Accessed 18 September 2021].

[6] Kleppmann, M., 2021. *Distributed Systems*. [online] Available at: <https://www.cl.cam.ac.uk/teaching/2021/ConcDisSys/dist-sys-notes.pdf> [Accessed 18 September 2021].

[7] Wåhslén, J., Orhan, I., Sturm, D. and Lindh, T., 2021. *Performance Evaluation of Time Synchronization and Clock Drift Compensation in Wireless Personal Area Networks*. [online] Available at: <https://www.researchgate.net/publication/262164027_Performance_Evaluation_of_Time_Synchronization_and_Clock_Drift_Compensation_in_Wireless_Personal_Area_Networks> [Accessed 18 September 2021].

[8] Wåhslén, J., Orhan, I. and Lindh, T., 2021. *Local Time Synchronization in Bluetooth Piconets for Data Fusion Using Mobile Phones*. [online] Ieeexplore.ieee.org. Available at: <https://ieeexplore.ieee.org/document/5955311> [Accessed 18 September 2021].

[9] Sandha, S. and Noor, J., 2021. *Exploiting Smartphone Peripherals for Precise Time Synchronization*. [online] Ieeexplore.ieee.org. Available at: <https://ieeexplore.ieee.org/abstract/document/8886639> [Accessed 18 September 2021].

[10] Hanusniak, V. and Jostiak, M., 2021. *Precision smartphones sensors time synchronization*. [online] Ieeexplore.ieee.org. Available at: <https://ieeexplore.ieee.org/document/7557159> [Accessed 18 September 2021].

[11] Courses.psu.edu. 2021. *Details of the GPS position calculation*. [online] Available at: <https://www.courses.psu.edu/aersp/aersp055_r81/satellites/gps_details.html> [Accessed 18 September 2021].

[12] Oceanservice.noaa.gov. 2021. *The Global Positioning System: Global Positioning Tutorial*. [online] Available at: <https://oceanservice.noaa.gov/education/tutorial_geodesy/geo09_gps.html> [Accessed 18 September 2021].

[13] Gps.gov. 2021. *GPS.gov: GPS Accuracy*. [online] Available at: <https://www.gps.gov/systems/gps/performance/accuracy/> [Accessed 18 September 2021].

[14] Android Developers. 2021. *Location | Android Developers*. [online] Available at: <https://developer.android.com/reference/android/location/Location> [Accessed 18 September 2021].

[15] Marzullo, K. and Owicki, S., 1984. *Maintaining the Time in a Distributed System*. [online] Uio.no. Available at: <https://www.uio.no/studier/emner/matnat/ifi/IN5570/v19/pensumliste/p44-marzullo.pdf> [Accessed 18 September 2021].

[16] Developer.apple.com. 2021. *Apple Developer Documentation*. [online] Available at: <https://developer.apple.com/documentation/corelocation/cllocationmanager/1423687-location> [Accessed 18 September 2021].

[17] Android Developers. 2021. *SensorEventListener | Android Developers*. [online] Available at: <https://developer.android.com/reference/android/hardware/SensorEventListener> [Accessed 18 September 2021].

[18] Android Developers. 2021. *Motion sensors | Android Developers*. [online] Available at: <https://developer.android.com/guide/topics/sensors/sensors_motion> [Accessed 18 September 2021].

[19] Advanced Swift. 2021. *Motion Data in Swift 5 [incl. Accelerometer, Gyroscope]*. [online] Available at: <https://www.advancedswift.com/get-motion-data-in-swift/> [Accessed 18 September 2021].

[20] Android Developers. 2021. *Getting Started with CameraX | Android Developers*. [online] Available at: <https://developer.android.com/codelabs/camerax-getting-started#5> [Accessed 18 September 2021].

[21] Android Developers. 2021. *System | Android Developers*. [online] Available at: <https://developer.android.com/reference/java/lang/System> [Accessed 18 September 2021].

[22] Transpire. 2021. *Obtaining Luminosity From An iOS Camera | Transpire*. [online] Available at: <https://www.transpire.com/insights/blog/obtaining-luminosity-ios-camera/> [Accessed 18 September 2021].

[23] Developer.apple.com. 2021. *Apple Developer Documentation*. [online] Available at: <https://developer.apple.com/documentation/corefoundation/1543542-cfabsolutetimegetcurrent> [Accessed 18 September 2021].

[24] Android Developers. 2021. *MediaRecorder overview | Android Developers*. [online] Available at: <https://developer.android.com/guide/topics/media/mediarecorder> [Accessed 18 September 2021].

[25] Developer.apple.com. 2021. *Apple Developer Documentation*. [online] Available at: <https://developer.apple.com/documentation/avfaudio/avaudiorecorder> [Accessed 18 September 2021].

[26] Android Developers. 2021. *View.OnClickListener | Android Developers*. [online] Available at: <https://developer.android.com/reference/android/view/View.OnClickListener> [Accessed 18 September 2021].

[27] Developer.apple.com. 2021. *Apple Developer Documentation*. [online] Available at: <https://developer.apple.com/documentation/uikit/uiresponder/1621142-touchesbegan> [Accessed 18 September 2021].

[28] ElProCus - Electronic Projects for Engineering Students. 2021. *Android Operating System : Introduction, Features & Its Applications*. [online] Available at: <https://www.elprocus.com/what-is-android-introduction-features-applications/> [Accessed 18 September 2021].

[29] Android Developers. 2021. *Meet Android Studio | Android Developers*. [online] Available at: <https://developer.android.com/studio/intro> [Accessed 18 September 2021].

[30] Heller, M., 2021. *What is Kotlin? The Java alternative explained*. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html> [Accessed 18 September 2021].

[31] Harvard Business Review. 2021. *Explaining XML*. [online] Available at: <https://hbr.org/2000/07/explaining-xml> [Accessed 18 September 2021].

[32] Wi-fi.org. 2021. *Wi-Fi Direct | Wi-Fi Alliance*. [online] Available at: <https://www.wi-fi.org/discover-wi-fi/wi-fi-direct> [Accessed 18 September 2021].

[33] Android Developers. 2021. *Local Only Hotspot | Android Developers*. [online] Available at: <https://developer.android.com/guide/topics/connectivity/localonlyhotspot> [Accessed 18 September 2021].

[34] Cisco. 2021. *What Is a Wireless LAN (WLAN)?*. [online] Available at: <https://www.cisco.com/c/en/us/products/wireless/wireless-lan.html> [Accessed 18 September 2021].

[35] Steve's Smart Home Networking Guide | Practical Home Networking and Home Automation. 2021. *Understanding Port Forwarding*. [online] Available at: <https://stevessmarthomeguide.com/understanding-port-forwarding/> [Accessed 18 September 2021].

[36] Android Developers. 2021. *CaptureResult | Android Developers*. [online] Available at: <https://developer.android.com/reference/android/hardware/camera2/CaptureResult#SENSOR_TIMESTAMP> [Accessed 19 September 2021].

[37] Developer.apple.com. 2021. *Apple Developer Documentation*. [online] Available at: <https://developer.apple.com/documentation/avfoundation/avcapturephoto/2873981-timestamp?language=objc> [Accessed 19 September 2021].

[38] Medium. 2021. *Time Synchronization in Distributed Systems*. [online] Available at: <https://medium.com/distributed-knowledge/time-synchronization-in-distributed-systems-a21808928bc8> [Accessed 26 September 2021].

[39] QVS Tech. 2021. *3 Factors That Affect Oscillator Frequency Stability | QVS Tech*. [online] Available at: <https://www.qvstech.com/3-factors-that-affect-oscillator-frequency-stability/> [Accessed 26 September 2021].

[40] Eecis.udel.edu. 2012. *Executive Summary: Computer Network Time Synchronization*. [online] Available at: <https://www.eecis.udel.edu/~mills/exec.html> [Accessed 14 October 2021].

[41] Ntp.org. 2011. *How does it work?*. [online] Available at: <http://www.ntp.org/ntpfaq/NTP-s-algo.htm#Q-ACCURATE-CLOCK> [Accessed 14 October 2021].

Appendix

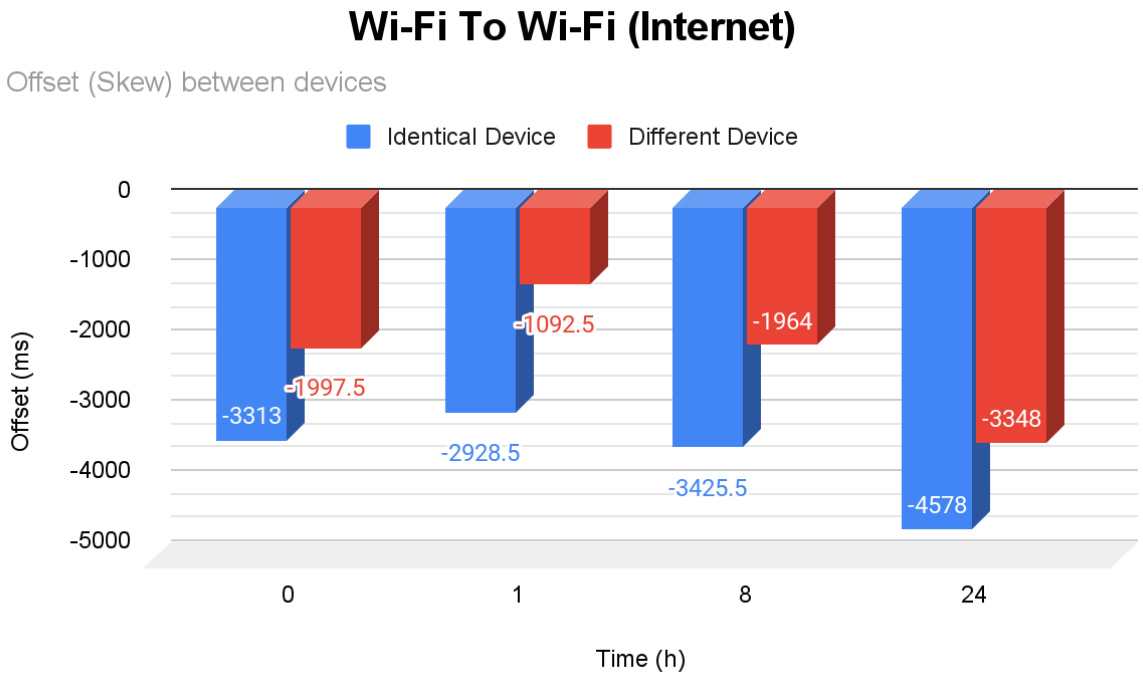

Figure A.1 Average offset between devices using internet

Standard Deviation:

Identical devices: at the start 272, after 1 hour 265, after 8 hours 297 and after 24 hours 343

Different devices: at the start 173, after 1 hour 158, after 8 hours 171 and after 24 hours 236

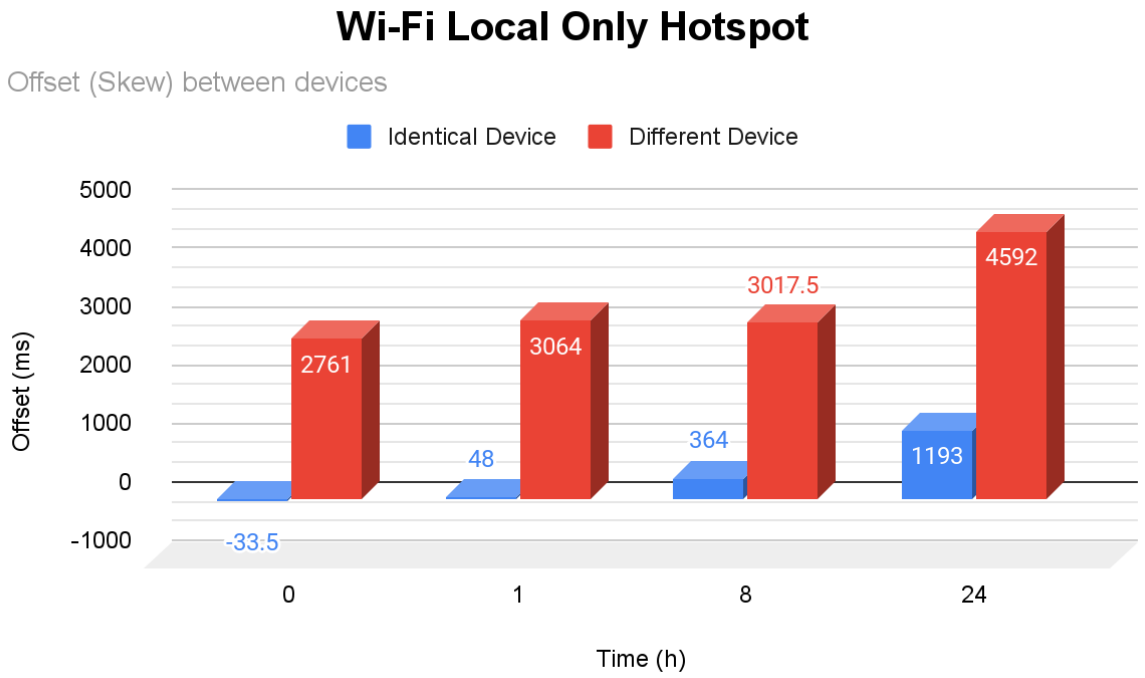

Figure A.2 Average offset between devices using hotspot

Standard Deviation:

Identical devices: at the start 251, after 1 hour 275, after 8 hours 317 and after 24 hours 335

Different devices: at the start 164, after 1 hour 172, after 8 hours 168 and after 24 hours 22

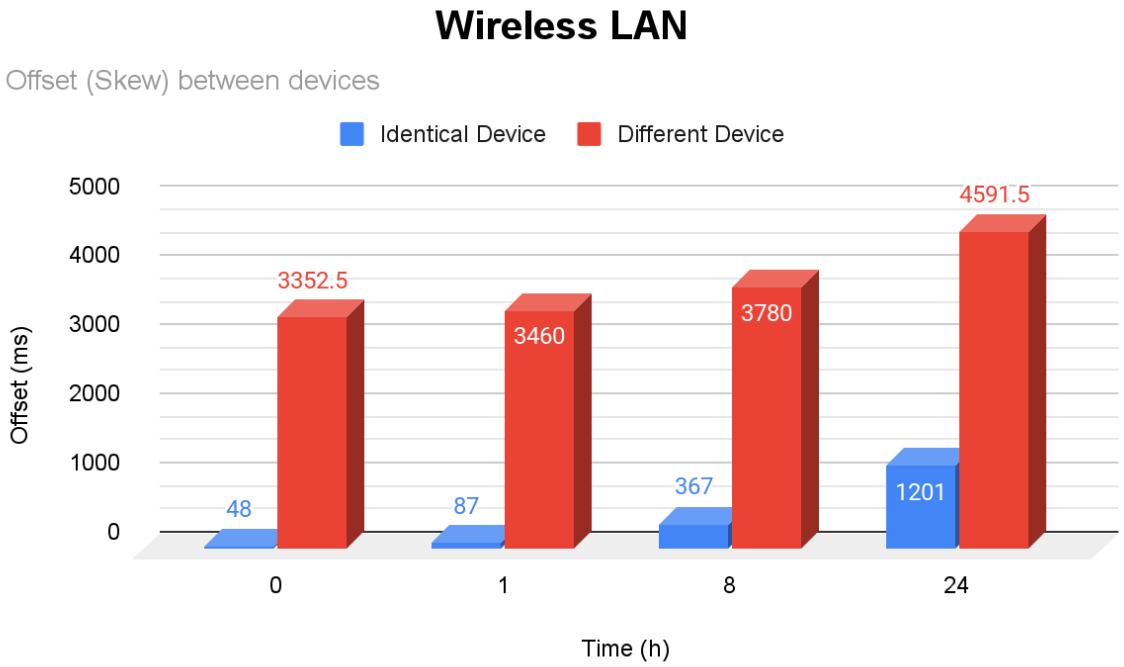

Figure A.3 Average offset between devices using wireless LAN

Standard Deviation:

Identical devices: at the start 273, after 1 hour 268, after 8 hours 310 and after 24 hours 341

Different devices: at the start 174, after 1 hour 183, after 8 hours 199 and after 24 hours 241

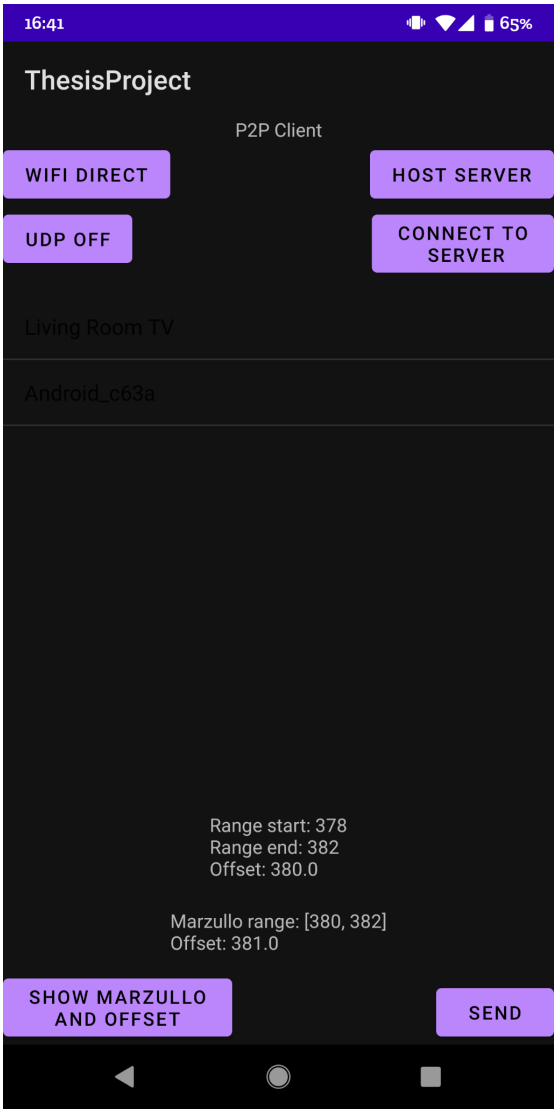

Figure A.4 An example of the results of calculating the offset and round-trip time using Cristian and Marzullo's algorithms in the

TRITA CBH-GRU-2021:230