School of Science and Technology
University of Siegen

Synchronization of Sensor Nodes in a Wireless Network Ubiquitous Systems

Student Research Project

Mechatronics

K Harish
1536256

February 17th, 2022

Examiners
Prof. Dr. Kristof Van Laerhoven — School of Science and Technology
M. Sc. Florian Wolling — Ubiquitous Computing Group

Adviser
M. Sc. Florian Wolling

This Student Research Project is handed in according to the requirements of the University of Siegen for the study program Master of Science (M. Sc.) Mechatronics.

Process period
October 1st, 2021 to February 17th, 2022

Examiners
Prof. Dr. Kristof Van Laerhoven
M. Sc. Florian Wolling

Adviser
M. Sc. Florian Wolling

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Abschlussarbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie die Stellen, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen sind, mit genauer Quellenangabe kenntlich gemacht habe.

Darüber hinaus erkläre ich, dass diese Arbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Declaration of Originality

I hereby declare that this master's thesis is my own original work and only the indicated sources and resources were used. All references, ideas and direct quotes taken from other sources and used in my thesis have been fully and properly cited.

Furthermore, it does not contain any material previously published or submitted, neither in whole nor in part, for credit of any other degree at any institution.

. . . .	. . . .
Ort, Datum / place, date	Unterschrift / signature

Abstract

This paper explains the time synchronization problem in wireless sensor networks and details the basic algorithms proposed in this area.

Wireless Sensor Networks (WSN) consist of a number of low-cost, miniature and lightweight sensor nodes like ESP32 devices that can communicate to each other over a wireless radio standard. For recording vital parameters most prominent characteristic of a wireless sensor network is the time synchronization, due to the physical separation between all the sensor nodes and each node has its own clock.

The system setup consists of three or more wireless sensor nodes with an implemented time synchronisation protocol, which aims at equalizing the local times for all nodes in the network. After synchronization,evaluation is performed where the three nodes gets simultaneously an interrupt by a fourth node to send their timestamp, which can be analysed later by comparing the timestamp of all nodes to master timestamp.

Conclusion in this case, without any time synchronization, the time offset mounts to more than 5 ms after 10 minutes when implemented with multiple wireless network devices. However, if all devices have the same clock frequency, which means no clock drift between devices, the period for which all nodes are within +/- 5 ms can be increased. When we choose the proper frequency of synchronization and all the devices in the network have identical clock frequencies or minimum clock drift between them, the implemented and evaluated Time Synchronization Protocol works in an optimal solution and the offset is always less than 5 ms with a possible error of less than 5%. Furthermore, we can conclude that aging of the ESP32 device can reduce clock speed.

Keywords: Wireless Sensor Networks (WSN), Time Synchronization Protocol (TSP) and ESP32 devices

Contents

1 Introduction

Aim of this student research project (Studienarbeit) is the synchronization of ESP32 sensor nodes in wireless network. Two or more nodes would receive synchronization signals to adjust their local time to global time with a targeted accuracy of less than 5 ms. Limiting Factors include the available limited performance of ESP32 nodes, and a good energy efficiency is needed. The wireless synchronization can take place via Wi-Fi or Bluetooth. The nodes we use are common ESP32 microcontrollers while using an additional central hub can be either Wi-Fi router or even a Raspberry Pi computer (local time or with internet access).

Specialized and highly efficient algorithms running on recent micro-controllers enable devices and promote for not only just motion but also for biomedical (electrocardiogram (ECG), photoplethysmogram (PPG) Heart rate) signals. Since the conventional ECG and PPG monitors are based on signals received form sensors mounted on humans through hard-wires, which are known to be accurate and synchronized. But the hard-wires from the nodes to monitors make it cumbersome and at times can damage the cables. We can implement wireless network of ESP32 devices with sensors connected to Wi-Fi/Bluetooth. We already have well developed libraries for ESP32 devices. The limiting factors are available performance and latency between devices and Wi-Fi network. If we can achieve latency with a targeted accuracy less than 5 ms along with a desirably better energy efficiency, then we can transmit sensor data over wireless network to desired client and synchronize data accordingly.

The most important aspect of the project is researching the various available microcontrollers and why the ESP32 microcontroller is best for our project. One must conduct research on the available wireless network communication modes, programming IDEs with ESP32 microcontroller extensions, and clock synchronization protocol algorithms.Finally, making decisions based on research and appropriate explanations that led to design decisions.

2 Theoretical Background

Any research for the project first starts with researching of the components required for our task. The most important component part is ESP32 device manufactured by Express-if and related components like WiFi router. After complete understanding of the work flow, the components required for our design are a computer with arduino IDE or ESP-IDF (Espressif IoT Development Framework) along with vs-code IDE, at-least three esp32 devices and batteries for the devices. All this put together with suitable libraries and programming each device helps us in achieving synchronization.

2.1 ESP32 Device

The ESP32 is a single 2.4 GHz WiFi-and-Bluetooth combo chip designed with the TSMC ultra-low-power 40 nm technology and created and developed by Espressif Systems, a Shanghai-based Chinese company. The ESP32 series is powered by a Tensilica Xtensa LX6 dual-core or single-core microprocessor, a Tensilica Xtensa LX7 dual-core or a single-core RISC-V microprocessor, and includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules. It is intended to deliver the best power and RF performance while demonstrating robustness, versatility, and dependability in a wide range of applications and power scenarios. It is the ESP8266 micro-controller's successor.

Since the original ESP32 was released, a number of variants have been introduced and announced. They are part of the ESP32 microcontroller family. These chips have varying CPUs and capabilities, but they all use the same SDK and are mostly code-compatible. The following are the family's main variants:

- **ESP32-S2:** This version includes a single-core Xtensa LX7 CPU with a clock speed of up to 240 MHz and no hardware Floating Point Unit (FPU). It has 320kilobytes of SRAM, 128kilobytes of ROM, and 16kilobytes of RTC memory. It does not support Bluetooth and has 43 programmable GPIOs as well as a USB OTG.
- **ESP32-C3:** This model has a single-core 32-bit RISC-V CPU with a clock speed of up to 160 MHz. It has 400 kilobytes of SRAM and 384 kilobytes of ROM. It supports Bluetooth 5 (BLE) and consumes Low Energy, and it has 22 programmable GPIOs that are pin compatible with the ESP8266.
- **ESP32-S3:** This version includes a dual-core Xtensa LX7 CPU with a clock speed of up to 240 MHz and additional instructions to speed up machine learning applications. It has 384 kilobytes of RAM and another 384 kilobytes

of SRAM. It has 44 programmable GPIOs and a USB OTG, and it supports Bluetooth 5 (BLE) and consumes Low Energy.

- **ESP32-C6:** This model includes a single-core 32-bit RISC-V CPU with a clock speed of up to 160 MHz. It has 400 kilobytes of SRAM and 384 kilobytes of ROM. It is IEEE 802.11ax (Wi-Fi 6) compatible on 2.4 GHz, with 20 MHz bandwidth in 11ax mode and 20 or 40 MHz bandwidth in 11b/g/n mode, Bluetooth 5 (BLE) support and Low Energy consumption, and 22 programmable GPIOs.

Furthermore, the original ESP32 was revised. The ESP32 series of chips includes the ESP32-D0WD-V3, ESP32-U4WDH, ESP32-S0WD, ESP32-D0WDQ6-V3 (NRND), ESP32-D0WD (NRND), and ESP32-D0WDQ6 (NRND), with the ESP32-D0WDQ6 (NRND) based on the ECO V3 wafer.

In this project, the ESP32 board is preferred due to its small size and ability to fit onto a breadboard, the ESP32 has built-in Wi-Fi capabilities, the memory size of the ESP32 is much larger when compared to devices like arduino zero, clock speed on ESP based boards is up to 160 MHz, which is much faster compared to boards running with 16 MHz, which is mostly used on all arduino boards, i.e., 10 times faster, and ESP-based boards are both inexpensive and power efficient when compared to other boards with comparable functionalities.

2.2 Programming

There various programming languages, frameworks, platforms, and environments used for ESP32 programming. They are as follows below :

- **Visual Studio Code** with the officially supported **Espressif Integrated Development Framework (ESP-IDF)** Extension.
- **Arduino IDE** with the **ESP32 Arduino Core**.
- **MicroPython** A lean implementation of Python 3 for microcontrollers.
- **Espressif Mesh Development Framework**.
- **Espruino** – JavaScript SDK and firmware closely emulating **Node.js**.
- **Lua Network/IoT** toolkit for **ESP32-Wrover**.
- **Mongoose OS** – an operating system for connected products on microcontrollers; programmable with JavaScript or C. A recommended platform by Espressif Systems, AWS IoT and Google Cloud IoT.
- **mruby** for the ESP32.
- **NodeMCU** – Lua-based firmware.
- **Zerynth** – Python for IoT and microcontrollers, including the ESP32
- **.Net nanoFramework** – Is a free and open-source platform that enables the writing of managed code applications for constrained embedded devices.

Out of the above programming tools for ESP32 programming, most commonly used are Visual Studio Code with ESP-IDF and Arduino IDE with ESP32 Arduino core. Lua, Python, and other programming languages are being ported and under developed. They will take a little longer to mature than the previous two options. However, Micro-Python is extremely stable, powerful, and simple to use. It is less difficult to use than many other embedded software platforms. But the previous two options has more readily available libraries and examples.

2.2.1 Visual Studio Code with ESP-IDF

Espressif offers basic hardware and software resources to assist application developers in realizing their ideas with the ESP32 series hardware. Espressif's software development framework is intended for the creation of Internet-of-Things (IoT) applications with Wi-Fi, Bluetooth, power management, and a variety of other system features.

Requirements :

- **Hardware :** An ESP32 board, USB cable - USB A / micro USB B and Computer running Windows, Linux or macOS.
- **Software :** Here we have a choice to either download and install the following software manually with **Toolchain** to compile code for ESP32, **Build tools** - CMake and Ninja to build a full Application for ESP32 and **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32 and scripts to operate the Toolchain. We can also use the following official plugins for integrated development environments (IDE) described in separate documents.
  - **Eclipse Plugin :** We can follow instructions here `https://github.com/espressif/idf-eclipse-plugin`
  - **VS Code Extension :** We can follow instructions here `https://github.com/espressif/vscode-esp-idf-extension`

2.2.2 Arduino IDE with ESP32 Arduino Core

The Arduino Integrated Development Environment (IDE) is a cross-platform application (for Windows, macOS, and Linux) written in C and C++ functions. It is used to write and upload programs to Arduino compatible boards, as well as other vendor development boards with the help of third-party cores. The Arduino IDE comes with a software library from the wiring project that includes many common input and output procedures. User-written code only requires two basic functions, which are compiled and linked with a program, to start the sketch and the main program loop.

The Arduino IDE has an add-on that allows you to program the ESP32 using the Arduino IDE and its programming language. To install the ESP32 board in your Arduino IDE, follow these next instructions refer to `https://docs.espressif.com/projects/arduino-esp32/en/latest/installing.html`

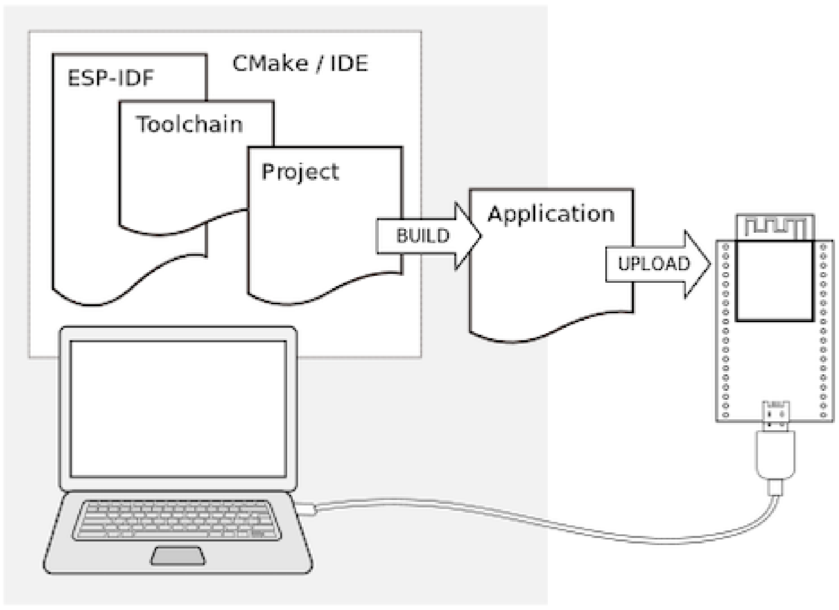


Figure 2.1: The Overview of Development of Applications for ESP32 an Illustration from `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/`.

2.3 Bluetooth

Bluetooth is a wireless technology standard for exchanging data over short distances that offers benefits such as robustness, low power consumption, and low cost.The chip incorporates a Bluetooth link controller and Bluetooth baseband, which perform baseband protocols as well as other low-level link routines like modulation/demodulation, packet processing, bit stream processing, frequency hopping, and so on. The Bluetooth system is divided into two types: classic Bluetooth and Bluetooth Low Energy (BLE). ESP32 supports dual-mode Bluetooth, which means it can communicate with both Classic Bluetooth and BLE devices.

The Bluetooth protocol stack is divided into two sections: a "controller stack" and a "host stack." The controller stack, which includes the PHY, Baseband, Link Controller, Link Manager, Device Manager, HCI (host controller interface), and other modules, is responsible for hardware interface and link management. The host stack contains Logical Link Control and Adaptation Protocol (L2CAP), Security Manager Protocol (SMP), Service Discovery Protocol (SDP), Attribute Protocol (ATT), Generic ATTribute Profile (GATT), Generic Access Profile (GAP), and various profiles and serves as an interface to the application layer, allowing it to access the Bluetooth system. The Bluetooth Host can run on the same device as the Controller or on a separate device. Both methods are supported by ESP32. Figure 2.2 depicts some common application structures.

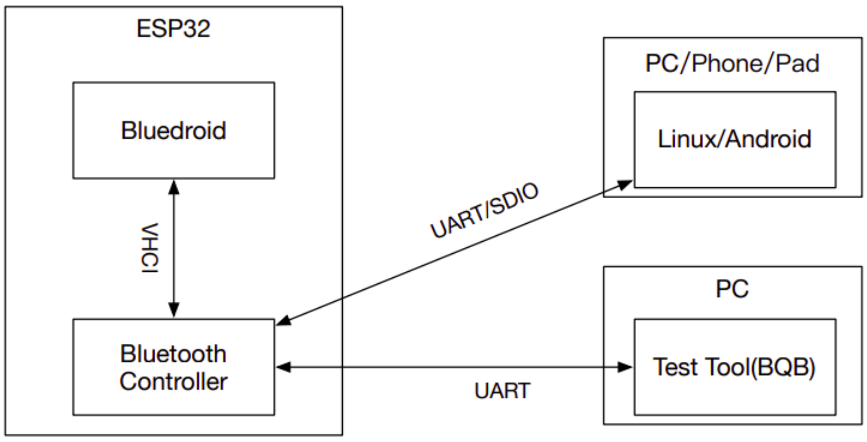


Figure 2.2: The Architecture of Bluetooth Host and Controller an Illustration from `https://www.espressif.com/sites/default/files/documentation/esp32_bluetooth_architecture_en.pdf`.

2.4 Wi-Fi

Wi-Fi is a wireless technology that allows computers, tablets, smartphones, and other devices to connect to the internet. Wi-Fi is a radio signal transmitted from a wireless router to a nearby device, which converts the signal into data that you can see and use. The device sends a radio signal back to the router, which is wired or cabled to the internet commonly.

The ESP32 supports TCP/IP as well as the full 802.11 b/g/n Wi-Fi MAC protocol. It supports the Basic Service Set (BSS) Station mode (STA), in which the ESP32 connects to an access point, the Soft Access Point mode (SoftAP), in which stations connect to the ESP32, and the Station/AP-coexistence mode, in which the ESP32 is both an access point and a station connected to another access point. These operations fall under the purview of the Distributed Control Function (DCF). Power management is handled with minimal host interaction to reduce the active-duty period.

2.4.1 Wi-Fi Radio and Baseband

The following features are supported by the ESP32 Wi-Fi Radio and Baseband: 802.11 b/g/n, 802.11 n MCS0-7 in both 20 and 40 MHz bandwidth, 802.11n MCS32 (RX), 802.11n 0.4 μs guard-interval (which is the space between symbols / characters being transmitted), up to 150 Mbps data rate, Receiving STBC 21 Transmitting power of up to 20.5 dBm, adjustable transmitting power, and antenna diversity with an external RF switch, the ESP32 supports antenna diversity. The RF switch is controlled by one or more GPIOs, which selects the best antenna to minimize the effects of channel fading.

The Modulation Coding Scheme (MCS) index is a well-known industry metric that is based on several parameters of a Wi-Fi connection between a client device and a wireless access point, such as data rate, channel width, and the number of antennas or spatial streams in the device. A guard interval is a time interval used to separate transmissions so that they do not interfere with one another. To improve throughput, the guard interval in IEEE 802.11n has been reduced from 800 ns to 400 ns. STBC (space-time block coding) is an optional feature of 802.11ac. It enables a transmitter to send multiple copies of a data stream using a variety of antennas. It also enables a receiver to choose the best copy of data from among multiple copies in order to improve reliability.

2.4.2 Wi-Fi MAC

The ESP32 Wi-Fi MAC automatically performs low-level protocol functions. They are as follows: 4 × virtual Wi-Fi interfaces, Simultaneous Infrastructure BSS Station mode/SoftAP mode/Promiscuous mode, Request To Send (RTS) protection, Clear To send (CTS) protection, Immediate Block (acknowledgement) ACK, Defragmentation,TX/RX A-MPDU (Aggregate MAC Protocol Data Unit), RX A-MSDU (Aggregate MAC Service Data Unit), TXOP (Transmit opportunity), WMM (Wi-Fi Multimedia), CCMP (Counter Mode CBC-MAC Protocol), TKIP (Temporal Key Integrity Protocol is a security protocol), WAPI (WLAN Authentication and Privacy Infrastructure), Wired Equivalent Privacy (WEP), and Automatic beacon monitoring hardware Timing Synchronization Function (TSF).

The 802.11 wireless networking protocol uses RTS/CTS (Request To Send / Clear To Send) as an optional mechanism to reduce frame collisions caused by the hidden node problem. Originally, the protocol also addressed the exposed node issue, but modern RTS/CTS includes ACKs and does not address the exposed node issue. Transmit opportunity (TXOP) is a MAC feature in 802.11. TXOP increases throughput for high priority data by providing contention-free channel access for a period of time.

A cipher block chaining message authentication code (CBC-MAC) is a method for creating a message authentication code using a block cipher. The message is encrypted in CBC mode with a block cipher algorithm, resulting in a chain of blocks where each block is dependent on the proper encryption of the previous block. This interdependence ensures that any change to any of the plaintext bits will cause the final encrypted block to change in ways that cannot be predicted or countered without knowing the block cipher key.

2.5 Clock Drift

Clock drift refers to a group of related phenomena in which a clock does not run at the exact same rate as a reference clock. That is, after a certain period of time, the clock drifts apart, or gradually desynchronizes from the other clock. All clocks drift, resulting in eventual divergence unless re-synchronized. Component aging, temperature changes that alter the piezoelectric effect in a crystal oscillator, or problems with a voltage regulator that controls the bias voltage to the oscillator are all possible causes. As a result, the same clock can have different drift rates at different times. Clock drift is traditionally measured in hertz per second (Hz/s). Clock stability is defined as the absence (or extremely low level) of frequency drift.

More advanced clocks and older mechanical clocks frequently have a speed trimmer that allows the user to adjust the speed of the clock and thus corrects the clock drift. For example, in pendulum clocks, the clock drift can be controlled by varying the length of the pendulum. A quartz oscillator is less susceptible to drift due to manufacturing variations than a mechanical clock's pendulum. As a result, most quartz clocks do not have an adjustable drift correction. But atomic clocks are extremely accurate and have almost no clock drift. Due to tidal acceleration and other effects, even the Earth's rotation rate has more drift and variation in drift than an atomic clock. Because of the precision of these oscillations, atomic clocks drift only about one second every hundred million years and as of 2015, the most accurate atomic clock loses one second every 15 billion years.

Clocks based on crystals are commonly found in electronic devices. These clocks are frequently subject to clock drift, which occurs when their time gradually becomes more and more inaccurate. This causes issues for distributed system designers because transactions can be made up of individual server accesses that must occur in a specific temporal order.The drift of crystal-based clocks used in computers, in particular, necessitates some kind of synchronization mechanism for any high-speed communication. There are several options for dealing with this issue. They include algorithms that account for known drift and protocols for keeping clocks in sync with one another.

2.6 Clock Synchronization

Every computer has a physical clock that counts crystal oscillations. The computer's software clock uses this hardware clock to keep track of the current time. However, the hardware clock is prone to drift, which causes the clock's frequency to shift and the time to become inaccurate. As a result, any two clocks at any given time are likely to be slightly different and can lose up to 40 microseconds per second. The difference between two clocks is referred to as the skew.

Clock synchronization attempts to coordinate clocks that would otherwise be independent. Even when initially set precisely, real clocks will differ over time due to clock drift, which is caused by clocks counting time at slightly different rates. There are several issues that arise as a result of clock rate differences, as well as several solutions. As a result of the difficulties in managing time at smaller scales, there are issues with clock skew that become more complex in distributed computing, where multiple computers must realize the same global time. For accurate reproduction of streaming media, synchronization is required. Audio over Ethernet systems rely heavily on clock synchronization.

Physical clocks can be synchronized in a variety of ways. External synchronization means that all computers in the system are synchronized with a time source that is not within the system (e.g., a UTC signal). Internal synchronization means that all computers in the system are in sync with one another, but the time may not always be accurate in relation to UTC.

Synchronization is straightforward and easy in a system that provides guaranteed bounds on message transmission time because the upper and lower bounds on a message's transmission time are known. A message is sent from one process to another indicating the current time, t. The second process adjusts its clock to t + (max+min)/2, where max and min are the upper and lower bounds for message transmission time. This ensures that the skew is no greater than (max-min)/2.

The synchronization solution in a system with a central server is simple; the server will dictate the system time. In this environment, Cristian's algorithm and the Berkeley algorithm are potential solutions to the clock synchronization problem.

In distributed computing, the problem takes on more complexity because a global time is not easily known. The Network Time Protocol (NTP) and Precision Time Protocol (PTP), which are layered client-server architectures based on User Datagram Protocol (UDP) message passing, are the most widely used clock synchronization solutions on the Internet. In distributed computing, Lamport timestamps and vector clocks are concepts of the logical clock.

The problem becomes even more difficult in a wireless network due to the possibility of synchronization packets colliding on the wireless medium and the higher drift rate of clocks on low-cost wireless devices. Here are some important synchronization techniques listed below :

2.6.1 Cristian's Algorithm

Cristian's algorithm is a clock synchronization method that can be used in many fields of distributive computer science but is most commonly used in low-latency intranets. It was developed by Flaviu Cristian in 1989. Cristian discovered that this simple algorithm is probabilistic in the sense that it only achieves synchronization if the request's round-trip time (RTT) is short in comparison to the required accuracy. It also suffers in single-server implementations, making it unsuitable for many distributive applications where redundancy is critical.

Cristian's algorithm communicates with a process P and a time server S that is linked to a time reference source. Simply put:

- At time t0, P requests the time from S.
- S prepares a response after receiving the request from P and appends the time T from its own clock.
- When P receives the response at time t1,we can find the round trip time (RTT) as t1-t0 .then it sets its time to T + RTT/2.

The synchronisation is error-free if the RTT is split equally between request and response. However, due to uncontrollable factors, this assumption is frequently incorrect. Longer RTTs indicate that the interference is generally asymmetrical. By selecting a suitable RTT from a set of many request/response pairs, offset and jitter of the synchronisation are thus minimized. The acceptance of an RTT at a given time is determined by the clock's drift and the RTT's statistics. These quantities can be measured during synchronisation, which optimizes the method on its own.

2.6.2 Berkeley Algorithm

The Berkeley algorithm is a distributed computing clock synchronization method that assumes no machine has an accurate time source. Gusella and Zatti created it in 1989 at the University of California, Berkeley. It is intended for use within intranets, as is Cristian's algorithm.

Unlike Cristian's algorithm, the Berkeley algorithm's server process, known as the leader, polls other follower processes on a regular basis. In general, the algorithm is as follows:

- An election process, such as the Chang and Roberts algorithm, is used to select a leader.
- In a manner similar to Cristian's algorithm, the leader polls the followers, who respond with their time.
- The leader observes the message round-trip time (RTT) and estimates the time of each follower and its own.
- The leader then averages the clock times, ignoring any values that are significantly different from the values of the others.
- Instead of sending the updated current time back to the other process, the leader sends out the amount (positive or negative) by which each follower's clock must be adjusted. This eliminates additional uncertainty caused by RTT at the follower processes.

The average cancels out the drifting tendencies of individual clocks using this method. Gusella and Zatti published results from 15 computers whose clocks were synchronized to within 20-25 milliseconds using their protocol.

When a negative clock change is received from the leader, computer systems normally avoid rewinding their clock. This would violate the monotonic time property, which is a fundamental assumption in certain algorithms in the system itself or in programs like make. A simple solution to this problem is to stop the clock for the duration specified by the leader, but this simple solution can also cause problems, albeit less severe ones. Most systems slow the clock for minor corrections, allowing the correction to be applied over a longer period of time.

When averaging the results, any client whose clock differs by a value outside of a given tolerance is frequently ignored. This prevents the overall system time from being significantly skewed as a result of a single erroneous clock.

2.6.3 Network Time Protocol

Network Time Protocol (NTP) is a highly reliable protocol that is widely used on the Internet. It has been extensively tested over the years and is widely regarded as the cutting-edge in distributed time synchronization protocols for unreliable networks. It can reduce synchronization offsets to a few milliseconds on the public Internet and to sub-millisecond levels on local area networks.

Simple Network Time Protocol (SNTP), a simplified version of the NTP protocol, can also be used as a pure single-shot stateless primary/secondary synchronization protocol, but it lacks the sophisticated features of NTP and thus has much lower performance and reliability levels.

Clock Synchronization Algorithm :

A typical NTP client polls one or more NTP servers on a regular basis. The client is responsible for calculating the time offset and round-trip delay. Time offset is the difference in absolute time between two clocks that is positive or negative (client time > server time). It is depicted in figure 2.3 and defined as follows:

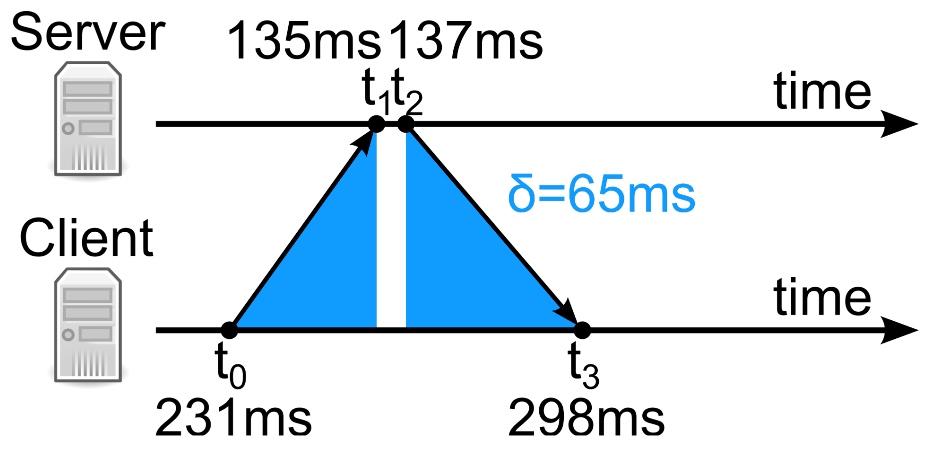


Figure 2.3: NTP Clock Synchronization Algorithm .

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \qquad [\mu\text{s}] \qquad (2.1)$$

as well as the round-trip delay of

$$\delta = (t_3 - t_0) + (t_2 - t_1) \qquad [\mu\text{s}] \qquad (2.2)$$

where

- t0 is the client's request packet transmission timestamp
- t1 is the server's request packet reception timestamp
- t2 is the server's response packet transmission timestamp and
- t3 is the client's response packet reception timestamp

To derive the offset expression, remember this one for the request packet,

$$t_1 = t_0 + \theta + \frac{\delta}{2} \qquad [] \qquad (2.3)$$

and for the response packet,

$$t_2 \;=\; t_3 + \theta - \frac{\delta}{2} \qquad [] \quad (2.4)$$

Solving for θ yields the definition of the time offset.

The values for θ and δ are filtered and statistically analyzed. Outliers are eliminated, and the best three remaining candidates are used to calculate the time offset. The clock frequency is then gradually reduced to reduce the offset, resulting in a feedback loop.When both the incoming and outgoing routes between the client and the server have symmetrical nominal delay, accurate synchronization is achieved. If the routes do not share a common nominal delay, a systematic bias of half the difference in forward and backward travel times exists.

2.6.4 Precision Time Protocol

The Precision Time Protocol (PTP) is a computer network protocol that is used to synchronize clocks across a network. It achieves clock accuracy in the sub-microsecond range on a local area network, making it suitable for measurement and control systems. PTP is currently used to synchronize financial transactions, cell phone tower transmissions, subsea acoustic arrays, and networks that require precise timing but do not have access to satellite navigation signals.

Clock Synchronization Algorithm : The best master clock (BMC) algorithm chooses the best candidate clock based on the following clock properties: unique identifier, accuracy, class, quality, priority 1 and priority 2.

PTP selects a master source of time for an IEEE 1588 domain and each network segment in the domain using the BMC algorithm. Clocks determine their own offset from their master. Let the variable t stand in for physical time. The offset o(t) at time t for a given follower device is defined as follows:

$$o(t) \;=\; s(t) - m(t) \qquad [\mu\text{s}] \quad (2.5)$$

where s(t) is the time measured by the follower clock at physical time t and m(t) is the time measured by the master clock at physical time t

The master clock broadcasts the current time to the other clocks on a regular basis. According to IEEE 1588-2002, broadcasts can occur up to once per second. IEEE 1588-2008 allows for up to ten per second.

Each broadcast begins at time T_1 with the master sending a Sync message to all clocks in the domain. When a clock receives this message, it records the local time T_1' at the time the message is received.the figure 2.4 depicts sample data exchanges for PTP synchronization.

Following that, the master may send a multi-cast Follow Up with an accurate T_1 timestamp. Not all masters can provide an accurate timestamp in the Sync message.

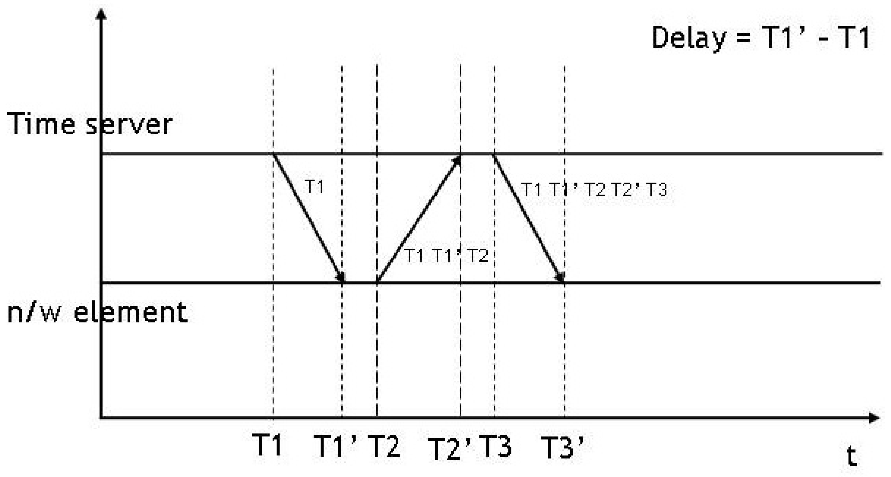


Figure 2.4: IEEE 1588 Synchronisation Mechanism and Delay Calculation .

Only after the transmission has finished can they obtain an accurate timestamp for the Sync transmission from their network hardware. Masters who have this limitation use the Follow Up message to communicate T_1. Masters with PTP capabilities built into their network hardware can present an accurate timestamp in the Sync message, eliminating the need for Follow Up messages.

Clocks must individually determine the network transit time of the Sync messages in order to accurately synchronize to their master. The transit time is calculated indirectly by measuring the round-trip time between each clock and its master. The network element clocks begin an exchange with their master in order to calculate the transit time δ. The exchange begins with a clock sending a Delay Req message to the master at time T_2. At time T_2', the master receives and timestamps the Delay Req message and responds with a Delay Resp message. The timestamp T_2' is included in the Delay Resp message by the master.

The network element clock learns from these exchanges T_1, T_1', T_2, T_2'

If δ is the Sync message's transit time and theta is the constant offset between master and follower clocks, then

$$T_1' - T_1 \;=\; \theta + \delta \; and \; T_2' - T_2 \;=\; -\theta + \delta \qquad [\mu\text{s}] \quad (2.6)$$

By combining the two equations above, we can derive that

$$\theta \;=\; \frac{1}{2}(T_1' - T_1 - T_2' + T_2) \qquad [\mu\text{s}] \quad (2.7)$$

The network element clock now understands the offset θ during this transaction and can correct itself by this amount to align with their master.

One assumption is that this message exchange occurs over such a short period of time that this offset can be safely considered constant over that time. Another assumption is that the transit time from the master to the follower is equal to the transit time from the follower to the master. Finally, it is assumed that both the master and the follower can precisely measure the time it takes to send or receive a message. The accuracy of the clock at the follower device is determined by how well these assumptions hold true.

2.7 Libraries

The Arduino IDE and ESP-IDF include an excellent library package manager that allows you to download and install library versions. The following are some important libraries that we will be using in this project:

- **Bluetooth API:** It has various types of functionality depending on the application requirements with separate libraries for each special function,which includes Bluetooth controller and Virtual Host Controller Interface (VHCI), Bluetooth common, Bluetooth Low Energy, Bluetooth Classic, NimBLE, and ESP-BLE-MESH for device communication.
- **Wi-Fi API:** The Wi-Fi libraries allow you to configure and monitor the ESP32's Wi-Fi networking functionality. This includes configuration for:
  - Station mode (aka STA mode or Wi-Fi client mode). ESP32 connects to an access point.
  - AP mode (aka Soft-AP mode or Access Point mode). Stations connect to the ESP32.
  - Combined AP-STA mode (ESP32 is concurrently an access point and a station connected to another access point).

  Wi-Fi API also has a variety of functionalities based on the application requirements with separate libraries for each special function,which includes Wi-Fi, SmartConfig, ESP-NOW, and ESP Mesh.
- **HTTP Client:** The http client provides an API for making HTTP/S requests from ESP-IDF/Arduino programs.
- **HTTP Server:** The HTTP Server component provides an ability for running a lightweight web server on ESP32.
- **NTP Client:** To obtain time from an NTP Server, the ESP32 must be connected to the Internet, and no additional hardware is required (like an RTC clock).Using an NTP Client library is the simplest way to obtain date and time from an NTP server.
- **Time library:** One need not to install any libraries to get the date and time from the ESP32. All you have to do is include the time.h library in your code.There are NTP servers like "pool.ntp.org" that anyone can use to request time as a client using the function below:

  ```
  configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
  ```

- **Asynchronous Web Server:** To create the web server, we'll use the ESPAsyncWebServer library, which makes it simple to create an asynchronous web server. Building an asynchronous web server has several benefits, which are listed below:
  - Handle multiple connections at the same time
  - When you send the response, you are immediately prepared to handle other connections while the server handles the response in the background.
  - To handle templates, a simple template processing engine is used.
- **ESP32Time:** It is a library for setting and retrieving internal RTC time on ESP32 boards.No need for external RTC module or NTP time synchronization.

3 Conceptual Design

In this project three or more ESP32 devices are used as nodes and clock synchronization is performed to achieve a time difference of less than 5 ms between the boards.The synchronization can be achieved either by getting the timestamp from server using internet access on all the devices or one device broadcasts it's local time added with one way latency of Wi-Fi communication to all the devices in the network. The above two ideas can illustrated more detailed in the following two sections.

3.1 All ESP32 Devices Synchronized with NTP Server

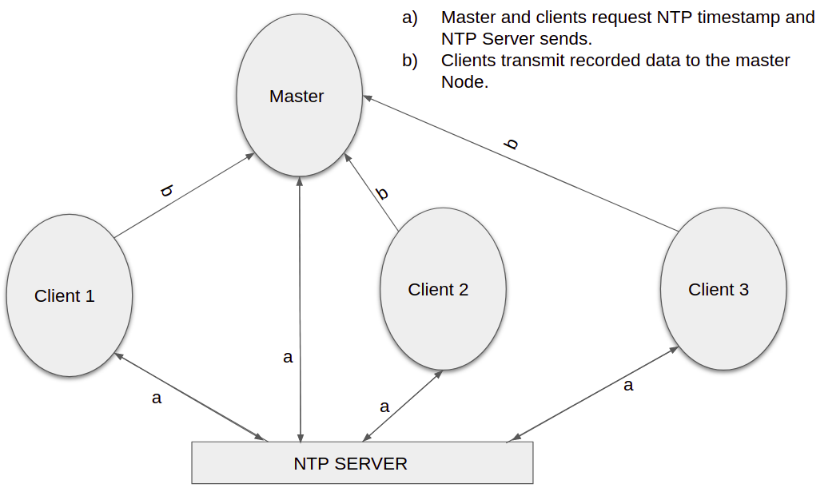
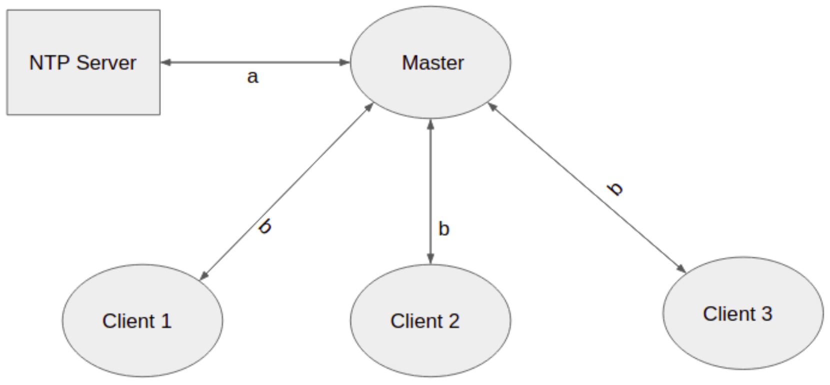

a) Master and clients request NTP timestamp and NTP Server sends.
b) Clients transmit recorded data to the master Node.
Master
Client 1
Client 2
Client 3
NTP SERVER
a
b

Figure 3.1: All devices synchronized with NTP Timestamps.

In this design, all nodes in the network, including the master and all clients, request a timestamp from the NTP server. The timestamps are then sent to the Master and all clients by the NTP Server. After synchronization is complete, the clients periodically transmit recorded data and timestamps at their respective nodes as measured by the sensors attached to the clients. Finally, the received data is sorted at the master node based on the timestamps received from clients. Accurate information can be deduced based on data received from all clients. Figure 3.1 shows a schematic diagram of the system. However,this project does not include data measurement using sensors attached to clients and the development of accurate information based on the received signals.

3.2 One Device as Central Node

In this design, only one of the network's master nodes requests a timestamp from the NTP server. The timestamp is then sent to the master by NTP Server. Here the master node acts as the central node, when the master node obtains the time, it sends the timestamp to all clients. Because we use Wi-Fi or Bluetooth for device communication, there will be a latency between the master's send timing and the client's receive timing. We can calculate the round trip latency between the master and each client by establishing two-way communication, which means that whenever the master sends a signal, the client responds with a signal to the master after receiving data from the master node. Then, based on the round trip latency of each client, we calculate the one-way latency and send the corrected timestamps to each client. Finally, the client nodes are brought into sync with the master node. Figure 3.2 shows a schematic diagram of the system.



a) Master request time from NTP Server and NTP Server Sends Time Stamp.
b) Master send time stamp to all clients. After Synchronization, Clients Periodically transmit the record data to the master node.

Figure 3.2: Master Gets Timestamp from NTP Server and Clients Get Timestamp from Master Node .

After synchronization is complete, the clients transmit recorded data and timestamps at their respective nodes as measured by sensors attached to the clients on a regular basis. Finally, at the master node, the received data is sorted based on the timestamps received from clients. Based on the data received from all clients, accurate information can be deduced. However, this project does not include data measurement using sensors attached to clients and the development of accurate information based on the received signals.

4 Implementation

In implementation, we must decide whether to use the ESP-IDF or Arduino IDE as a programming platform, whether to use Bluetooth or WiFi for wireless communication between devices, and which of the time synchronization protocols to use among all the clock synchronization protocols in the section 2.6. Following the above decisions, we will discuss the implementation of both conceptual designs mentioned in the previous chapter.

4.1 Design Decisions

4.1.1 Programming Platform

In our theoretical section, we discussed various programming platforms or environments for ESP32 devices. However, of all the available environments, Visual Studio Code with ESP-IDF and Arduino IDE with ESP32 Arduino core are the most commonly used.

The toolchain-xtensa32 package is used in both cases. Toolchain is just the compiler (and linker). However, the dependencies differ in both cases. The ESP-IDF framework includes an API for more technically savvy users. Arduino is designed for people who are just starting out (though you can achieve the same thing in every framework with enough of your own code). Because it is easier to develop and understand in the Arduino IDE, it's better to use it for this project.

4.1.2 Time Synchronization Protocol

We discussed various time synchronization protocols in our theoretical section. However, the synchronization solution in a system with a central server is simple, as the server will dictate the system time. In this scenario, Cristian's algorithm and the Berkeley algorithm are potential solutions to the clock synchronization problem. When we use one node as the central node we will use the Cristian's algorithm over the Berkeley algorithm because we can fix one node as the central time server and the remaining as clients.

In distributed computing, the problem takes on more complexity because a global time is not easily known. The Network Time Protocol (NTP) and Precision Time Protocol (PTP), which are layered client-server architectures based on User Datagram Protocol (UDP) message passing, are the most widely used clock synchronization solutions on the Internet.

NTP can reduce synchronization offsets on the public Internet to a few milliseconds and to sub-millisecond levels on local area networks. PTP, on the other hand, achieves sub-microsecond clock accuracy on a local area network but requires continuous synchronization. We have an easily accessible library for NTP synchronization, which allows nodes to be used as NTP clients.

4.1.3 Wireless Communication

WiFi is a radio wireless local area networking technology based on the IEEE 802.11 standards. WiFi operates on 2.4GHz or 5GHz frequencies and can typically cover greater distances than Bluetooth. Bluetooth is designed to send small data chunks, whereas WiFi can send data at speeds of up to 150 Mbps in esp32 devices.

Bluetooth used to have serious security flaws, but most of them have been addressed by newer standards. WiFi, on the other hand, was previously concerned with security, so it developed security protocols such as Wired Equivalent Privacy (WEP), Wi-Fi Protected Access (WPA), WPA2, and WPA3. It is preferable to use the most recent security protocol, WPA3. In almost all cases, properly configured Bluetooth connection security would suffice, but if the data you want to send is sensitive and you want extra security, consider using WiFi. For now we will opt the WiFi for communication between the devices.

ESP-NOW communication protocol will be used among all WiFi communication protocols available because it is a connection less wireless communication protocol that allows paired devices to communicate directly with one another through the data-link layer defined by Espressif. The entire pairing process does not require a Wi-Fi connection or a third-party device such as a mobile phone. The main difference is that ESP-NOW reduces the OSI model's five layers to just one. In other words, data does not need to pass through the network layer, transport layer, session layer, presentation layer, or application layer. Furthermore, there is no need for packet headers or unpackers on each layer, resulting in a faster response and a reduction in the delay caused by packet loss in congested networks.

After powering on, the devices can transmit data and control other paired devices without requiring a wireless connection and with a response time of milliseconds. ESP-NOW reduces the OSI model's five layers to just one, resulting in truly simple communication and low power consumption. A control button can run for two years on two AA batteries.

We will use the ESP-NOW protocol for wireless communication for synchronization because the devices can transmit data quickly, resulting in less latency when sending to other nodes and receiving their responses. As a result, determining the precise round trip latency over WiFi communication becomes easier for our task. Furthermore, ESP-NOW consumes less power.

4.2 All ESP32 Devices Synchronized with NTP Server

In this scenario, all the nodes gets the NTP timestamp from the NTP server "pool.ntp.org". Refer to below listing 4.1 for code snippet used to get the timestamp from the NTP server to all the devices.

Listing 4.1: Code Snippet for Getting NTP Timestamp on Nodes

```
#include <ESP32Time.h>
#include "WiFi.h"

// Give access to your router which has internet connection
const char* ssid0        = "YourOwnWifi";
const char* password0    = "YourPassword";

// Give NTP server address, GMT Offset and Day light Offset
const char* ntpServer = "pool.ntp.org" ;
const long  gmtOffset_sec = 3600;
const int   daylightOffset_sec = 3600;

void setup(){
    // Init Serial Monitor
    Serial.begin(115200);
    Serial.println();

    // Connect to WiFi Router
    Serial.printf("Connecting to %s ", ssid0);
    WiFi.begin(ssid0, password0);

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println(" CONNECTED");

    //init and get the timestamp from NTP Server
    configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);

    //disconnect WiFi as it's no longer needed
    WiFi.disconnect(true);
    WiFi.mode(WIFI_OFF);
}
```

For the client nodes send the measured data through WiFi using the ESP-NOW protocol use the code snippet in the below listing 4.2, for initializing and adding other devices as peer device.

Listing 4.2: Code Snippet for Adding a Device as Peer in ESP-NOW

```
#include "WiFi.h"
#include "esp_now.h"

// REPLACE WITH THE MAC Address of your receiver
uint8_t AddressOfClient[] = {0x7C, 0x9E, 0xBD, 0x91, 0xEB, 0x90};

void setup(){
    // Init Serial Monitor
    Serial.begin(115200);
    Serial.println();

    // Set device as a Wi-Fi Station
    WiFi.mode(WIFI_STA);

    // Init ESP-NOW
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }

    // Once ESPNow is successfully Init, we will register for
    // Send CB to get the status of Transmitted packet
    esp_now_register_send_cb(OnDataSent);

    // Register peer
    esp_now_peer_info_t peerInfo;
    peerInfo.channel = 0;
    peerInfo.encrypt = false;

    // Add first peer
    memcpy(peerInfo.peer_addr, AddressOfClient, 6);
    if (esp_now_add_peer(&peerInfo) != ESP_OK){
        Serial.println("Failed to add peer");
        return;
    }

    // Register for a callback function that will be called
    // when data is received
    esp_now_register_recv_cb(OnDataRecv);
}
```

4.3 One Device as Central Node

In this design, only one of the network's master nodes requests a timestamp from the NTP server. The timestamp is then sent to the master by NTP Server. When the master node obtains the time, it sends the timestamp to all clients.

Listing 4.1 shows a code snippet for getting the NTP timestamp on the master node. To enable the master to send time stamps, one must add all clients as peer devices to the master. Listing 4.2 illustrates how to add a device as a peer device.

Listing 4.3: Code Snippet for Calculating One Way Latency Between The Master and Each Client

```
// Callback when data is received
void OnDataRecv(const uint8_t * mac_addr, const uint8_t \
*incomingData, int len) {
    // Copy received data from the client
    memcpy(&myData_rev, incomingData, sizeof(myData_rev));

    // Round trip latency including the processing time
    // at other board
    long time_diff_2_way= ((rtc.getEpoch() - \
    boardCurrentSendTimes[myData_rev.id - 1].epoch) * \
    1000000L) + (rtc.getMicros() - \
    boardCurrentSendTimes[myData_rev.id - 1].time_us);

    // Copying processing time of client
    boardsStruct[myData_rev.id -1].time_diff = \
    myData_rev.time_diff;

    // calculating one way latency excluding the
    // processing time at other board
    if (clients[myData_rev.id - 1].iteration < 10){
        clients[myData_rev.id - 1].cur_latency = \
        (((time_diff_2_way) - (myData_rev.time_diff))/2);

    // Storing the latencies for example 10 here
        clients[myData_rev.id - 1].last_ten_latencies \
        [clients[myData_rev.id - 1].iteration]= \
        clients[myData_rev.id - 1].cur_latency;

    // increment the iteration
        clients[myData_rev.id - 1].iteration= \
        (clients[myData_rev.id - 1].iteration)+1;
    }
}
```

For calculating the one way latency, use the time stamp of master before sending the sample data to the clients and the timestamp at the time when the master receives the response from the client. Refer to the listing 4.3 for the code snippet.

After calculating the one way latency for certain iterations. One needs to calculate the average one way latency and send the updated timestamps to each device by using the code in the listing 4.4.

Listing 4.4: Code Snippet for Sending Timestamps With Added Latency to The Clients

```
void loop {
    // we can use any no.of iterations to estimate latency
    if (cur_iteration == 10) {
        // Calculating the average one way latency
        clients[0].avg_latency = \
        average(clients[0].last_ten_latencies, 10);

        // get the current epoch
        myData_send.epoch = rtc.getEpoch();

        // Get the present micros and Add the
        // average one way latency
        myData_send.time_us = (long)(rtc.getMicros()) + \
        clients[0].avg_latency ;

        // check overflow of microseconds
        if (myData_send.time_us >= 1000000L){
            myData_send.time_us = myData_send.time_us - 1000000L;
            myData_send.epoch = myData_send.epoch +1 ;
        }

        // Sending the corrected timestamp information to clients
        esp_err_t result = esp_now_send(broadcastAddressClient1,\
        (uint8_t *) &myData_send, sizeof(myData_send));

        if (result == ESP_OK) {
            Serial.println("Sent with success \r\n");
        }
        else {
            Serial.println("Error sending the data");
        }
    }
}
```

At the clients, set the time by using the timestamp sent by the master node and use ESP32time library to get and set time. Refer to the below code in listing 4.5 to set the time to a device.

Listing 4.5: Setting Time to The Client Nodes

```
#include "time.h"
#include <ESP32Time.h>

// Callback when data is received
void OnDataRecv(const uint8_t * mac_addr, const uint8_t \
*incomingData, int len) {
    // Copy the received time stamp locally
    memcpy(&myData_rev, incomingData, sizeof(myData_rev));

    // Set the rtc time
    rtc.setTime(myData_rev.epoch, myData_rev.time_us );
    // time offset day light and timezone
    setenv("TZ", "CET-1", 1);
    tzset();
}
```

For the client nodes to send the measured data through WiFi using the ESP-NOW protocol use the code snippet in the below listing 4.2 , which is self explanatory.

5 Evaluation

The designed and implemented system must be evaluated to ensure that all devices are synchronized within +/- 5 milliseconds. For testing, we use one of the GPIO pins on the master node in output mode as a trigger to request timestamps from all of the clients. To read the trigger sent by the master node, we use one of the GPIO pins on all of the clients in input mode. When clients detect a trigger from the master node, they respond by transmitting their respective timestamps. The master then compares the received timestamps to the timestamp captured by the master node at the time the master node sent the request to the clients.Figure 5.1 shows a schematic diagram of the system.

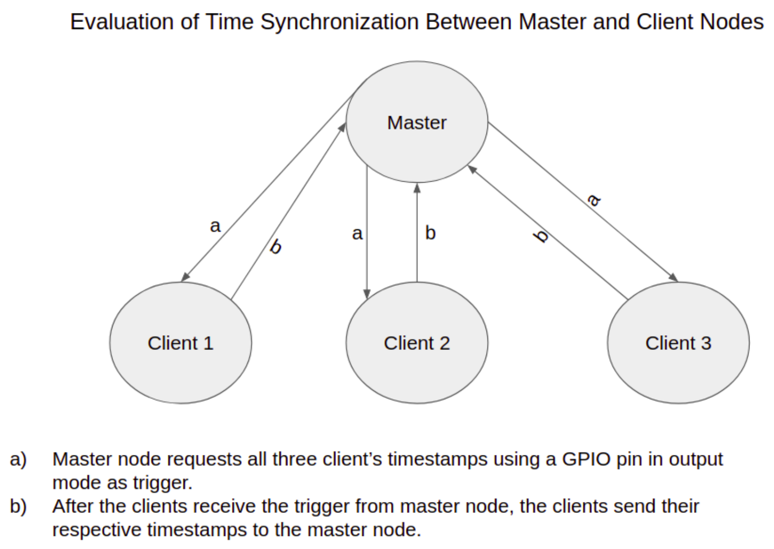

Evaluation of Time Synchronization Between Master and Client Nodes
Master
a
b
a
b
a
b
Client 1
Client 2
Client 3
a) Master node requests all three client's timestamps using a GPIO pin in output mode as trigger.
b) After the clients receive the trigger from master node, the clients send their respective timestamps to the master node.

Figure 5.1: Evaluation of Time Synchronization Between Master and Client Nodes

.

To draw conclusions, one has to evaluate the implemented system using various methods. To evaluate here, one has to use each device separately, all three with NTP synchronization, all three with concurrent latency check, and all three with separate latency check. The different methods are as follows: with no drift correction, with drift correction every 30 seconds, 60 seconds, 120 seconds, 300 seconds, every 600 seconds, every 900 seconds, every 1200 seconds, and every 1800 seconds.

5.1 Without Drift Correction

In this scenario, we evaluate without applying correction in all the cases as depicted in figure 5.2. We have evaluated individual boards separately, which can be illustrated from figure 5.2 d, 5.2e and 5.2f for board 1 , board 2 and board3 respectively. Figure 5.2d shows that the board 1 has a clock drift of 0.3 $\mu s/s$, which is almost negligible drift when compared to the master board, whereas figure 5.2e and 5.2f are the offset of boards 2 and 3, which deviates significantly from the master. By examining the clock drift of board 2 and board 3 are 3.5 $\mu s/s$ and 3.12 $\mu s/s$, we can conclude that boards 2 and 3 have faster clocks than the master and board 1.

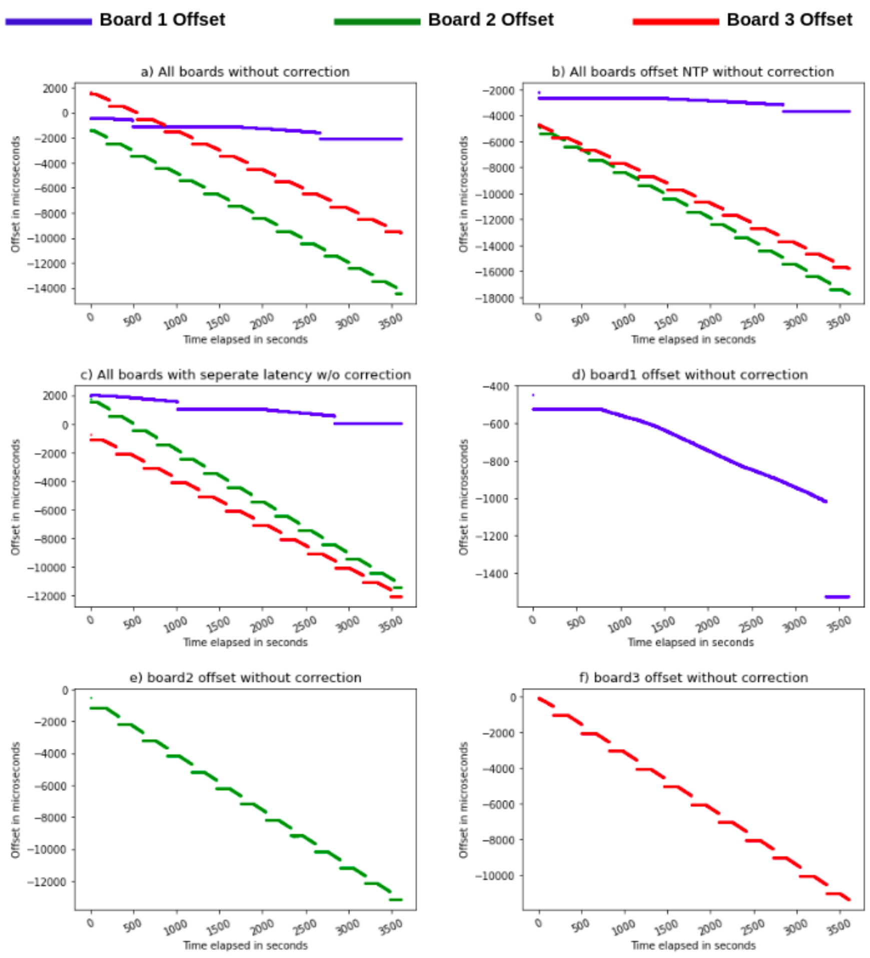


Figure 5.2: All Scenarios Without Drift Correction .

When we synchronized all of the devices with the NTP server, the initial latency can exceed 5 ms as seen from figure 5.2b. In case of board 1 there are no outliers whereas as board 2 and board 3 has 99.47 % and 97.25 % with a starting latency of -4872 μs and -4729 μs respectively, which is less effective synchronization solution.

To accurately estimate the latency between devices, we implemented one device as a central node / master node and the rest as clients, who receive the timestamp from the master node after the latency between nodes has been properly estimated. As illustrated in Figure 5.2a and 5.2c.

When synchronized all the devices with master as central node in case of figure 5.2a, where latency of each device is concurrently checked and synchronized by master node. We the checked the time differences for an hour and found that board 1 has an average offset of -1318 μs, board 2 has an average latency of -7783 μs and board 3 has an average latency of -4037 μs. As the board 1 has almost the same clock frequency as master node so the outliers are zero, in case of board 2 we have an outliers of 71.2 % as its initial latency is -1318 μs and in case of board 3 we have an outliers of 40.5 % as its initial latency is 1647 μs.

In case of figure 5.2c, where latency of each device is estimated separately and synchronized by master node. We the checked the time differences for an hour and found that board 1 has an average offset of 1020 μs, board 2 has an average latency of -4807 μs and board 3 has an average latency of -6420 μs. The board 1 has no the outliers, in case of board 2 we have an outliers of 48.17 % as its initial latency is 1769 μs and in case of board 3 we have an outliers of 40.5 % as its initial latency is -745 μs.

We can conclude that different devices has different clock speed compared to each other. Synchronizing all the devices from NTP server is less effective because the initial latency can be more and with clock drift it can drift away. Whereas in case of one node acting as central node we can see the initial latency is more precise compared to earlier case.

5.2 Drift Correction at 300 Seconds

In this scenario, we evaluate with applying correction every 300 seconds in all the cases depicted in figure 5.3. We have evaluated individual boards separately, which can be illustrated from figure 5.3c , 5.3d and 5.3e that the board 1, board 2 and board 3 has an average latency of 693 μs , -772 μs and -904 μs respectively with no outliers, which means all the boards are precisely synchronized individually with master node. we can clearly see that with applying correction the board 1, 2 and 3 offset has been kept within range of +/-5 ms.

When synchronized all the devices with master as central node in case of figure 5.3a, where latency of each device is concurrently checked and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of 162 μs, -2376 μs and 1268 μs respectively with no outliers.

In case of figure 5.2c, where latency of each device is estimated separately and synchronized by master node.We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of 656 μs, -707 μs and -750 μs respectively with no outliers.

We implemented two cases with one node as the central node, where the central node checks the latency of the devices simultaneously in the first case and the latency of each device separately in the second case, so that no congestion occurs at the central node's receive buffer when the client sends the response to the master. We can see that the latency is within the limits in both cases, but in the second case we have achieved greater accuracy and consistency. It can be illustrated form figure 5.3a and 5.3b.

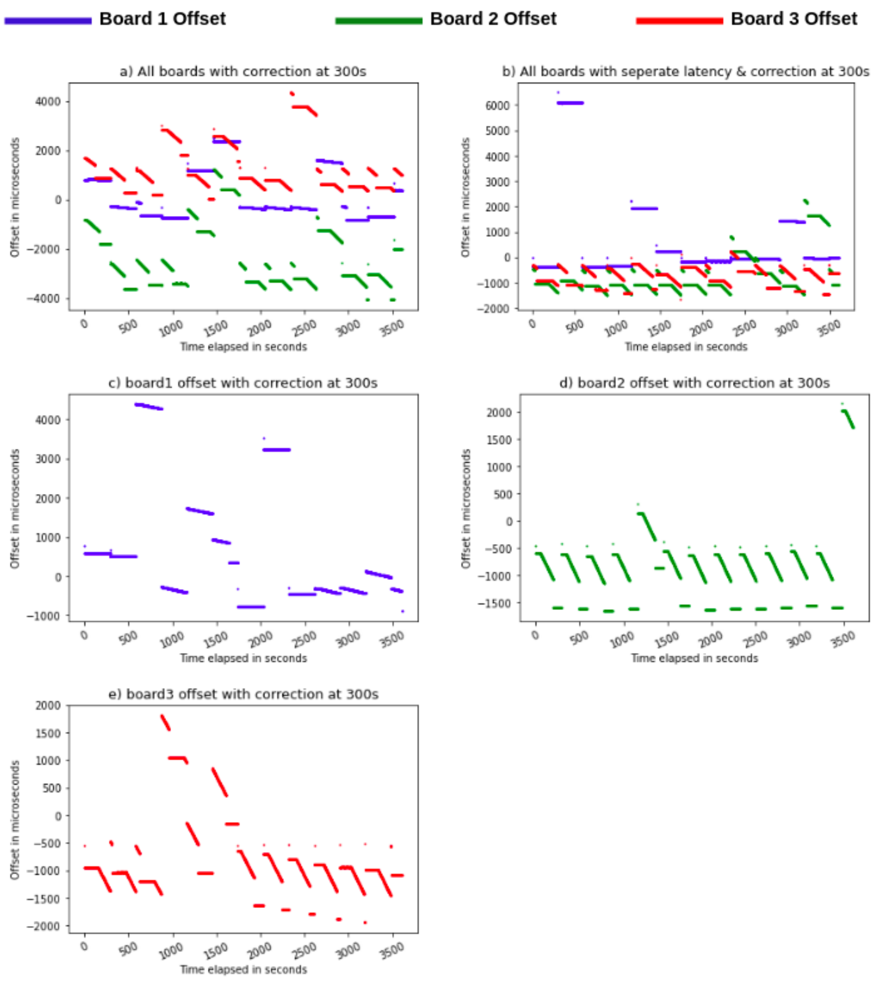


Figure 5.3: All Scenarios With Drift Correction at 300 Seconds .

5.3 Drift Correction at 600 Seconds

In this scenario, we evaluate with applying correction every 600 seconds in all the cases depicted in figure 5.4. We have evaluated individual boards separately, which can be illustrated from figure 5.3c , 5.3d and 5.3e that the board 1, board 2 and board 3 has an average latency of -196 μs , -882 μs and -1107 μs respectively with no outliers, which means all the boards are precisely synchronized individually with master node. We can clearly see that with applying correction the board 2 and 3 offset has been kept within range of +/-5 ms but has more offset compare to the earlier case when correction applied at every 300 seconds because of drift.

We have attempted to control drift in the case where all devices are synchronized with the NTP server by retrieving the timestamp on the nodes every 600 seconds.

But there is still no drift correction in this case. In this case the board 1 has no outliers whereas as board 2 and board 3 has 42.54 and 38.67 percents with a starting latency of 1263 μs and 382 μs respectively, which is less effective synchronization solution. Figure 5.4b shows an example of this.

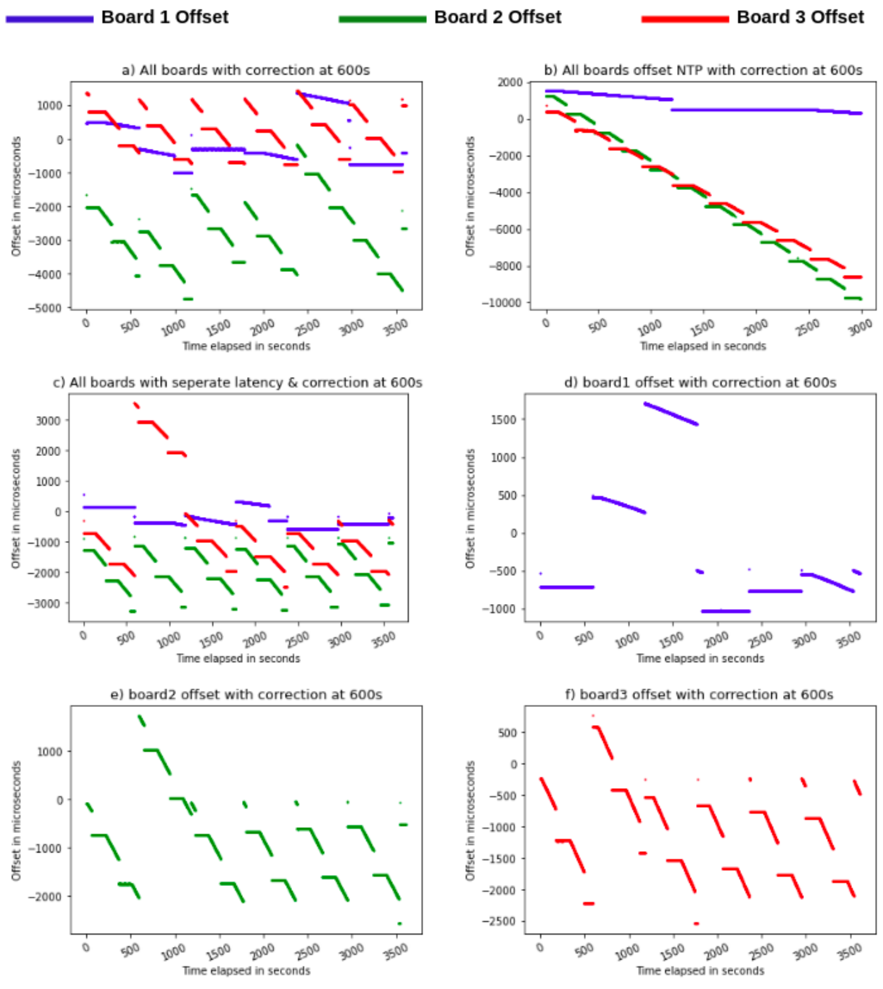


Figure 5.4: All Scenarios With Drift Correction at 600 Seconds .

When synchronized all the devices with master as central node in case of figure 5.4a, where latency of each device is concurrently checked and synchronized by master node. We have checked the time differences for an hour and found that board 1 , board 2 and board 3 has an average offset of -86 μs, -2830 μs and 188 μs respectively with no outliers.

In case of figure 5.4c, where latency of each device is estimated separately and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -247 μs, -1980 μs and -631 μs respectively with no outliers.

We can see that the latency is within acceptable limits in both cases, but the second case has achieved higher accuracy and consistency. It can be illustrated form figure 5.4a and 5.4c. But synchronizing all the devices from NTP server can be less effective because the initial latency can be more and with clock drift it can drift away from master node.

5.4 Drift Correction at 900 Seconds

In this scenario, we evaluate with applying correction every 900 seconds in all the cases depicted in figure 5.5. We have evaluated individual boards separately, which can be illustrated from figure 5.5c , 5.5d and 5.5e that the board 1, board 2 and board 3 has an average latency of 186 μs , -1602 μs and -1441 μs respectively with no outliers, which means all the boards are precisely synchronized individually with master node. We can clearly see that with applying correction the board 2 and 3 offset has been kept within range of +/-5 ms but has even more offset compare to the earlier case when correction applied at every 600 seconds because of drift.

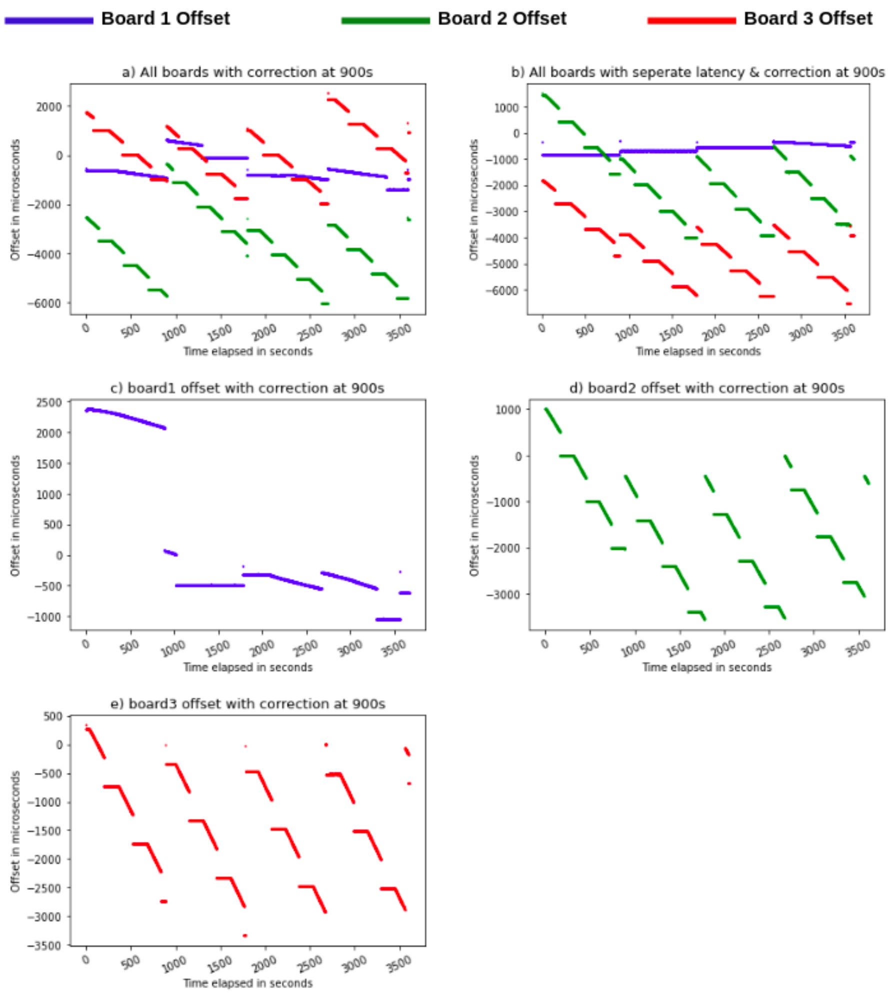


Figure 5.5: All Scenarios With Drift Correction at 900 Seconds .

When synchronized all the devices with master as central node in case of figure 5.5a, where latency of each device is concurrently checked and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -577 μs, -3714 μs and 108 μs respectively with no outliers in case of board 1 and board 3, but in case of board 2 outliers are around 21.09 %.

In case of figure 5.5b, where latency of each device is estimated separately and synchronized by master node.We have checked the time differences for an hour and found that board 1 , board 2 and board 3 has an average offset of -614 μs, -1804 μs and -4534 μs respectively with no outliers in case of board 1 and board 2, but in case of board 3 outliers are around 35.67 %.

We can also see that the latency is not within the limits in both cases, and any of the devices can drift outside of the limits, but in the second case, we have achieved greater accuracy and consistency. Figure 5.5 shows an example of this. Basically with longer synchronization periods allows more drift between the devices but stays within the limits if only the devices has very small initial offset.

5.5 Drift Correction at 1200 Seconds

In this scenario, we evaluate with applying correction every 1200 seconds in all the cases depicted in figure 5.6. We have evaluated individual boards separately, which can be illustrated from figure 5.5c and 5.5d that the board 2 and board 3 has an average latency of -2283 μs and -1958 μs respectively with no outliers, which means all the boards are precisely synchronized individually with master node. We can clearly see that with applying correction the board 2 and 3 offset has been kept within range of +/-5 ms but has even more offset compare to the earlier case when correction applied at every 900 seconds because of drift.

When synchronized all the devices with master as central node in case of figure 5.5a, where latency of each device is concurrently checked and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -221 μs, -2276 μs and 419 μs respectively with no outliers in case of board 1 and board 3, but in case of board 2 outliers are around 16.93 %.

In case of figure 5.5b, where latency of each device is estimated separately and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -294 μs, -2840 μs and -2470 μs respectively with no outliers.

We can also see that the latency is not within the limits in both cases, and any of the devices can drift outside of the limits, but in the second case, we have achieved greater accuracy and consistency. Figure 5.6 shows an example of this. Basically with longer synchronization periods allows more drift between the devices but stays within the limits if only the devices has very small initial offset.

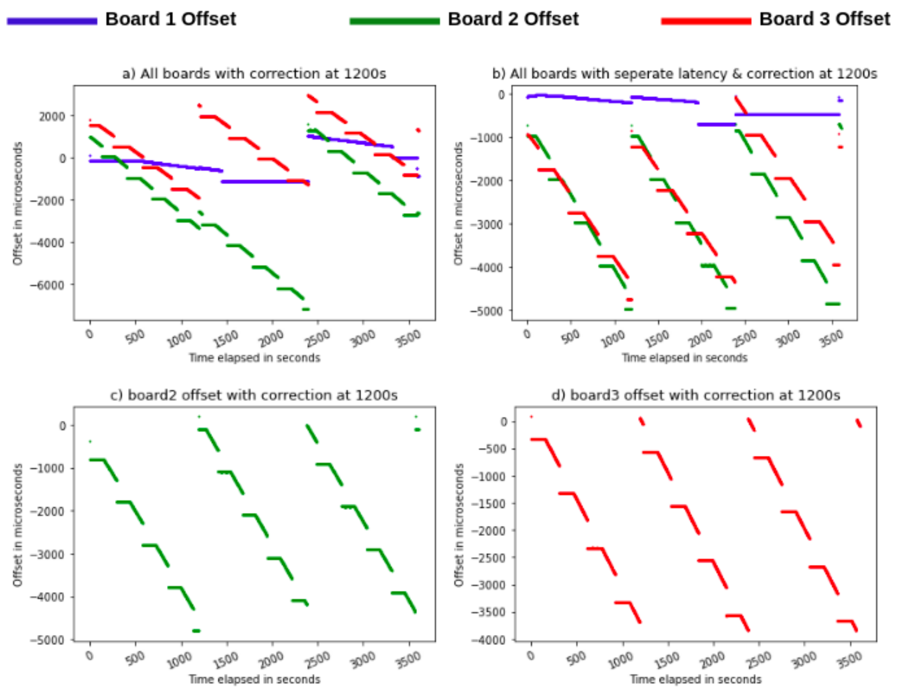


Figure 5.6: All Scenarios With Drift Correction at 1200 Seconds .

5.6 Drift Correction at 1800 Seconds

In this scenario, we evaluate with applying correction every 1800 seconds in all the cases depicted in figure 5.7. We have evaluated individual boards separately, which can be illustrated from figure 5.7c , 5.7d and 5.7e that the board 1, board 2 and board 3 has an average latency of -477 μs , -1863 μs and -1380 μs respectively with no outliers in case of board 1 and has 11.72 % and 7.07 % in case of board 2 and board 3 respectively, which means all the boards are not precisely synchronized individually with master node during 1800 seconds. We can clearly see that with applying correction the board 2 and 3 offset has been kept within range of +/-5 ms but has even more offset compare to the earlier case when correction applied at every 1200 seconds because of drift.

When synchronized all the devices with master as central node in case of figure 5.7a, where latency of each device is concurrently checked and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -416 μs, -5247 μs and -1233 μs respectively with no outliers in case of board 1 and board 3, but in case of board 2 outliers are around 51.86 %.

In case of figure 5.5b, where latency of each device is estimated separately and synchronized by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -294 μs, -2840 μs and -2470 μs respectively with no outliers in case of board 1 , but in case of board 2 and board 3 outliers are around 33.54 % and 8.32 % respectively.

We can also see that the latency is not within the limits in both cases, and both

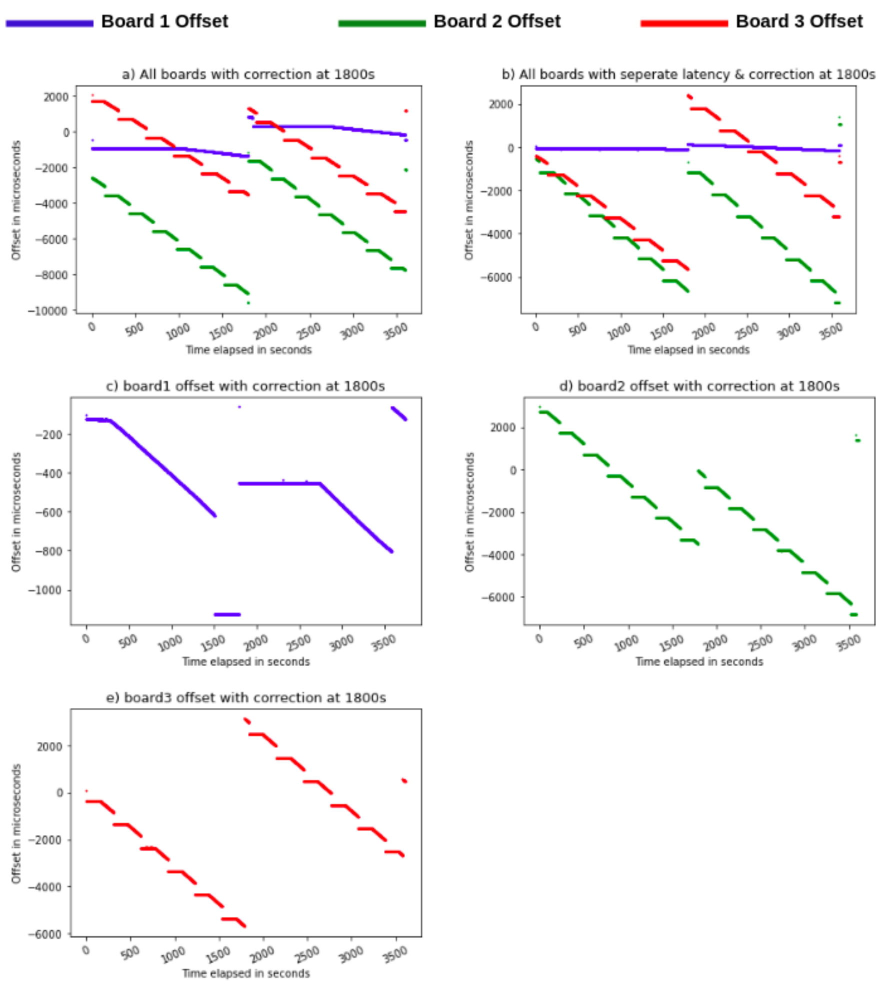


Figure 5.7: All Scenarios With Drift Correction at 1800 Seconds .

boards 2 and 3 are drifting outside of the limits. Figure 5.7 shows an example of this.

5.7 Drift Correction at 30, 60 and 120 Seconds

We implemented two cases with one node as the central node with applying correction every 30, 60 and 120 seconds depicted in figure 5.8, where the central node checks the latency of the devices simultaneously in the first case and the latency of each device separately in the second case, so that no congestion occurs at the central node's receive buffer when the client sends the response to the master.

When synchronized all the devices with master as central node in case of figure 5.8a, where latency of each device is concurrently checked and synchronized every 30 seconds by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of 484 μs, 438 μs and 3397 μs with outliers of 9.17, 10.92 and 24.25 % respectively.

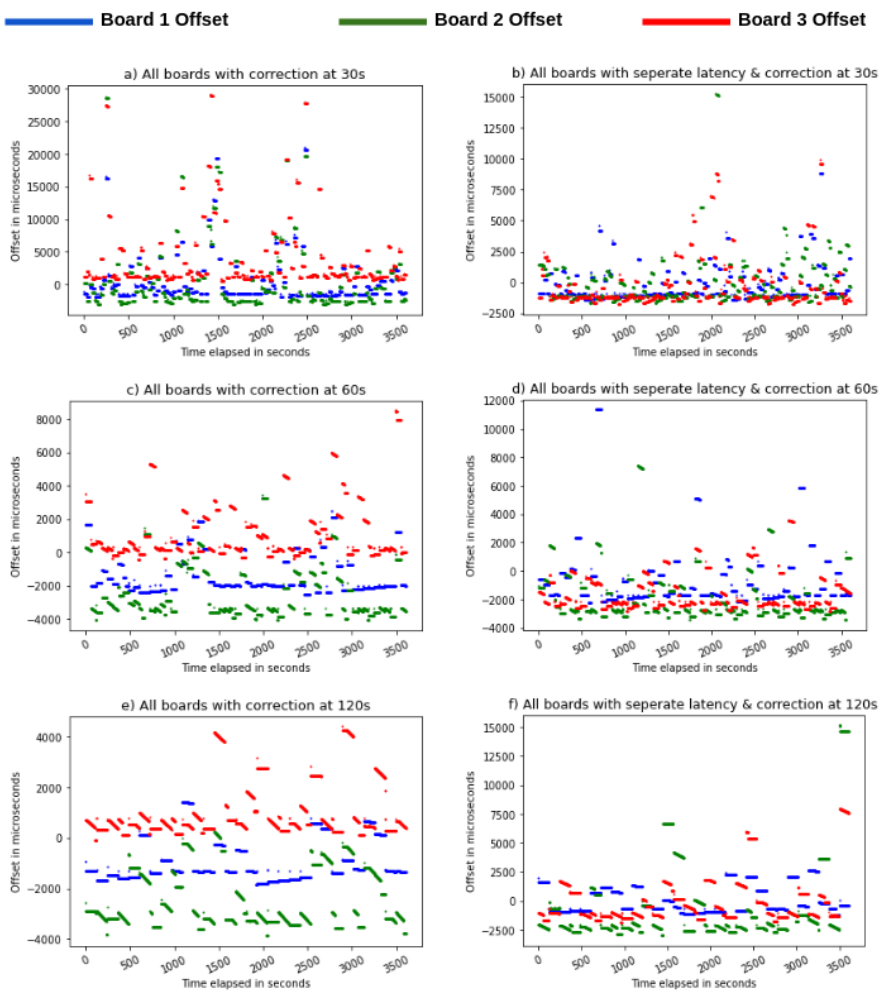


Figure 5.8: Applying Corrections for Drift Frequently at 30, 60 and 120 Seconds .

In case of figure 5.8b, where latency of each device is estimated separately and synchronized every 30 seconds by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -237 μs, -159 μs and -338 μs with outliers of 0.86, 1.64 and 2.63 % respectively.

When synchronized all the devices with master as central node in case of figure 5.8c, where latency of each device is concurrently checked and synchronized every 60 seconds by master node . We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -1386 μs, -2479 μs and 1094 μs respectively with no outliers in case of board 1 and board 2 but in case of board 3 outliers are of 4.99 %.

In case of figure 5.8d, where latency of each device is estimated separately and synchronized every 60 seconds by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -684 μs, -1912 μs and -1595 μs respectively with no outliers in case of board 3 but in case of board 1 and board 3 outliers are of 4.96 and 1.6 % respectively.

When synchronized all the devices with master as central node in case of figure

5.8e, where latency of each device is concurrently checked and synchronized every 120 seconds by master node . We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of -957 μs, -2454 μs and 986 μs respectively with no outliers.

In case of figure 5.8f, where latency of each device is estimated separately and synchronized every 120 seconds by master node. We have checked the time differences for an hour and found that board 1, board 2 and board 3 has an average offset of 51 μs, -690 μs and -56 μs respectively with no outliers in case of board 1 but in case of board 2 and board 3 outliers are of 6.32 and 6.37 % respectively.

We can see that the latency is within acceptable limits in both cases, but the second case has achieved higher accuracy and consistency. Figure 5.8 also clearly illustrates this. Also we see some outliers because of frequent synchronization or latency correction.

5.8 Results and Discussion

From Section 5.1 , we can illustrate that various devices has different clock speed compared to each other. The board 1 has a clock drift of 0.3 $\mu s/s$, which can stay synchronized with master for longer time, whereas board 2 and 3 has clock drift of 3.5 $\mu s/s$ and 3.12 $\mu s/s$. That implies board 2 and 3 have faster clocks compared to master and board 1 , which are most used boards.

From section 5.2 and 5.3, when we evaluated the offset for an hour and we can see that with regular synchronization of devices at 300 and 600 seconds, all devices are within +/- 5 ms offset from the master node. However, the offset between the master and clients is more dispersed in the case of 600 seconds than in the case of 300 seconds because more time between synchronization intervals allows all devices to drift away from the master.

From section 5.4 and 5.4, when we evaluated the offset for an hour and we can illustrate that with regular synchronization of devices at 900 and 1200 seconds, sometimes all devices are not within the +/- 5 ms offset from the master node. However, the offset can be within the range if all of the node's initial offsets are close to zero or positive because all of the device clocks are faster than the master node, causing the clock to drift for a longer period of time in order to drift out of range.

From section 5.5, when we evaluated the offset for an hour, we can see that with regular device synchronization at 1800 seconds, all of the devices always drift away and go out of range because the time interval between synchronization intervals is too long and insufficient to control offset of board 2 and board 3 from master due to their faster clocks compared to master.

Form section 5.6, we have evaluated the offset for an hour, We can see that with regular synchronization at faster intervals like 30, 60, and 120 seconds, all devices almost always stay in range because the shorter duration between synchronization intervals reduces clock drift but some outliers are present. We can also illustrate that second case, where the latency between all the devices and master is checked separately has achieved higher accuracy and consistency.

Overall, we discovered that using one node as the central node and checking the latency of each device separately yields more accurate and consistent results. Besides that, we can conclude that the device's aging causes the clock speed to slow down when compared to new devices. Furthermore, in all cases, we can see that the offset of all devices and the master is constant for some time, then there is some linear drift for some time, and finally there is a sudden jump in the offset. The above phenomenon has to be investigated further in detailed.

6 Conclusions

In more often all the scenarios, was able to achieve time synchronization between the master and the clients within +/- 5 milliseconds initially after synchronization, as specified in the problem statement. There will be outliers, but they will be rare. One need to synchronize frequently because of clock drift between the devices. In this case, without any time synchronization, the time offset increases to more than 5 ms after 10 minutes of use with multiple wireless network devices. If one can find the devices with identical clock speed , then we can avoid synchronizing frequently and achieve the offset within +/- 5 ms with a possible error of less than 5%. Furthermore, One can conclude that the master and board 1 are older boards that are slower than the new boards 2 and 3, implying that aging of the ESP32 device can reduce clock speed.

This project was able to achieve time synchronization between the master and the clients by using a single device as the central node / master node, which sends time stamps to the clients after calculating the latency of each client. In this case, the central node checks the latency of the devices simultaneously in the first case and the latency of each device separately in the second case, to ensure that there is no congestion at the central node's receive buffer when the client sends the response to the master. Based on the results of the evaluation section, One can conclude that the second case achieved greater overall accuracy and consistency.

Further work on the project should be focused on the offset of all devices and the master is constant for some time, then there is some linear drift for some time, and finally there is a sudden jump in the offset. Which means there are sudden jumps in clock drift between the devices at regular intervals, as shown in the graphs of the evaluation section. One should delve deeper into the sudden jumps and constant for some duration. Furthermore, the outliers of latency or offset between devices should be statistically eliminated. In addition to the preceding discussion, one can calculate the drift of each device over a longer period of time, and making drift corrections on the device itself on a regular basis can potentially eliminate or at least increase the synchronization interval.

List of Figures

Listings

List of Equations

Bibliography

[1] ESP32 Technical Reference Manual (2021) *: a document preparation by Espressif systems* *https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf*, version 4.6.

[2] ESP32 series data sheet (2021) by Espressif Systems, *: a document preparation by Espressif systems* *https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf*, version 3.8 .

[3] Arduino Library List, `https://www.arduinolibraries.info/architectures/esp32`, accessed on 16/09/2021.

[4] Arduino esp32, `https://github.com/espressif/arduino-esp32`, accessed on 19/09/2021.

[5] "Clock Synchronization: Open Problems in Theory and Practice" written by Christoph Lenzen, Thomas Locher, Philipp Sommer, and Roger Wattenhofer accessed on accessed on 17/11/2021.

[6] "Timing-sync Protocol for Sensor Networks" by Ganeriwal, Kumar, and Srivastava accessed on 15/11/2021.

[7] "Fine-Grained Network Time Synchronization using Reference Broadcasts" by Elson, Girod, and Estrin accessed on 12/11/2021.

[8] "Probabilistic clock synchronization" (1989) by Flaviu Cristian in IBM Almaden Research Center accessed on 22/11/2021.

[9] "The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD" (1989) by Gusella R , Zatti S accessed on 22/11/2021.

[10] "Computer Network Time Synchronization: The Network Time Protocol" by David L. Mills accessed on 07/10/2021

[11] LUMINEX PTPv2, `https://www.luminex.be/improve-your-timekeeping-with-ptpv2/`, accessed on 07/10/2021.

[12] Arduino core for the ESP32, `https://github.com/espressif/arduino-esp32`, accessed on 01/10/2021.

[13] Espressif IoT Development Framework, `https://github.com/espressif/esp-idf`, accessed on 03/10/2021.

[14] ESP-IDF Programming Guide Getting Started, `https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/get-started/index.html`, accessed on 05/10/2021.

[15] Arduino-ESP32 Programming Guide Getting Started and examples, `https://docs.espressif.com/projects/arduino-esp32/en/latest/getting_started.html`, accessed on 13/10/2021.

[16] Bluetooth API, `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/bluetooth/index.html`, accessed on 05/09/2021.

[17] WiFi , `https://www.verizon.com/info/definitions/wifi/`, accessed on 10/09/2021.

[18] WiFi API, `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_wifi.html`, accessed on 12/09/2021.

[19] ESP-NOW user guide (2021),*: a document preparation by Espressif systems* ***https://www.espressif.com/sites/default/files/documentation/esp-now_user_guide_en.pdf***, accessed on 13/09/2021.

[20] ESP32 Bluetooth Architecture (2019) *: a document preparation by Espressif systems* ***https://www.espressif.com/sites/default/files/documentation/esp32_bluetooth_architecture_en.pdf***, version 1.1 accessed on 15/09/2021.

[21] ESP32 Bluetooth Networking user guide (2019) *: a document preparation by Espressif systems* ***https://www.espressif.com/sites/default/files/documentation/esp32_bluetooth_networking_user_guide_en.pdf***, version 1.2 accessed on 15/09/2021.