εχνολογικό Εκπαιδευτικό Ίδρυμα Κρήτης

Σχολή Τεχνολογικών Εφαρμογών

Τμήμα Μηχανικών Πληροφορικής

Πτυχιακή εργασία

Τίτλος: One-Way Delay Computations for Real Time Clock Synchronization

Νταλλαρής Ευστράτιος (ΑΜ: 4291)

Επιβλέπων εκπαιδευτικός: Γραμματικάκης Μιλτιάδης

Επιτροπή Αξιολόγησης: Γραμματικάκης Μιλτιάδης, Παναγιωτάκης Σπυρίδων και Παπαγρηγορίου Αντώνιος

Ημερομηνία παρουσίασης : 19-12-2018

Abstract

Global time agreement and clock synchronization in a network of nodes are important issues which cannot be easily measured due to unstable delays and asymmetric paths. An important metric used to obtain an estimation of the network delays is round-trip time. Round-trip time is often for approximating one-way delay. Specifically, under a common, assumption that the path is symmetric, one-way delay is estimated as half the round-trip time.

However, very often this estimation is not very accurate. For example, the path that the packet follows to reach its destination is not always the same as reaching its source. Even if the reverse path is symmetric, unstable delays due to operating system intervention, such as interrupts while the packet is waiting at the socket buffer, and/or asymmetries in the processing time of messages make this estimate inaccurate.

The aim of this thesis is to review and implement more accurate one-way delay estimation methods. We implement Yoo and Choi's technique that focuses on measuring round-trip time on both sides with a ping pong procedure to estimate *one-way delay* using network calculus. We compare divergence of this method to the common practice of using "half the round-trip time". We also implement Aoki, Oki, and Cessa's model to measure *one-way delay variation* by relating it to inter-packet delay, providing linear regression to remove the clock skew. Experiments were performed on CAN Bus with two AVR microprocessors, and on Ethernet with two Odroid XU4 connected to a fast Ethernet router. Results show that one-way delay can, be computed much more accurately than earlier approximations. Moreover, CAN network offers a much smaller one-way delay variation than Ethernet.

Notice that further more complex experiments related to computing one-way delay (and especially round trip time) in the presence of other CAN bus traffic in a 6-node configuration have been reported independently at the APPLEPIES conference "Applications in Electronics Pervading Industry, Environment" in Pisa, Italy (September 26-27, 2018). Proceedings are to appear in Lecture Notes in Electrical Engineering.

Acknowledgments

I would like to express my gratitude to my thesis advisor Dr. Miltiadis Grammatikakis, who guided me and motivated me during my thesis. Also, I would like to thank Mr. Antonis Papagrigoriou, and my colleague Mr. Mouzakitis Nikolaos who supported me in critical moments of this thesis. Finally, I would like to thank Antonis Papagrigoriou, and Mr. Panagiotakis Spyros for participating in my diploma defense committee.

Table of Contents

1.0 Introduction 5
1.1 Motivation 5
1.2 Outline 5

2. Background and Related Work 6
2.1 Digital Clocks 6
2.2 Atomic Clocks 8
2.3 Clock Synchronization 9
2.3.1 Clock Synchronization with physical clocks 9
2.3.2 Clock Synchronization with logical clocks 11
2.3.3 External Clock Synchronization 14

3. One-way delay Algorithms 17
3.1 Describing Round-Trip Time and One-Way Delay 17
3.2 Ping-Pong-based Technique 20
3.2.1 Implementation using POSIX Socket API on Ethernet 22
3.2.2 Implementation on CAN Bus Protocol 27
3.3 Aoki, Oki, Cessa Model 31
3.3.1 Implementation using POSIX Socket API on Ethernet 33
3.3.2 Implementation on CAN Bus Protocol 35

4. Experimental Framework and Results 36

5. Conclusions and Future work 41

References: 43

Appendix A1. One-way Delay: TCP 44

Appendix A2. One-way Delay: CAN Bus 51

Appendix A3. One-way Delay Variation: TCP 53

Appendix A4. One-way Delay Variation: CAN Bus 60

1.0 Introduction

1.1 Motivation

A distributed system is a group of end-nodes in a network, sharing resources, information and capabilities to complete the same goal. Each node has a clock with quartz accuracy. Sharing a common sense of time in a network is critical for tasks that have to be cooperative. There are three basic reasons for having nearly the same clock value in distributed systems:

1. To start events at nearly the same time.
2. To timestamp tasks and events in such a way that the order can be kept.
3. The time between the tasks can be accurately estimated.

The most accurate way to synchronize clocks nowadays in a network is using Global Positioning System. However, GPS is not that usual in computer networks nowadays, as it requires additional hardware equipment, such as antennas for every end point (or a group of hosts), making its use impractical.

Another way to synchronize clocks within the same network is by applying internal clock synchronization. This can be done only if there is a node with a high precision reference clock that is explicitly dedicated to this purpose. A major concern in internal clock synchronization is network delay. As an estimation of the network delay many implementations assume that the network path is symmetric and they estimate one-way delay as RTT/2, e.g. [9], [10]. In this thesis, we examine methods that provide a more accurate estimation of one-way delay and one-way delay variation.

1.2 Outline

The thesis is organized as follows:

Section II: **Background** work related to digital and atomic clock operation, round-trip time and one-way delay. There is also a brief explanation of the basic clock synchronization algorithms.

Section III: **One-way Delay Estimation** based on the model by Yoo and Choi is examined and implementation on CAN Bus and POSIX sockets is described and analyzed, as well as One-Way Delay Variation based on the model by Aoki, Oki and Cessa model [11].

Section IV: **Results** concentrate on the experiment framework and analysis.

Section V: **Coclusion and Future Work** examines how this work can be further extended and improved.

2. Background and Related Work

2.1 Digital Clocks

Operation of a digital clock is based on the qualities of the quartz crystal that facilitates the accuracy of the device. To understand the working of a quartz clock, it is important to identify its primary components.

(a) Battery

This is the power source of the device. It supplies current to the microchip circuit.

(b) Quartz crystal oscillator

Oscillates to facilitate electron movement to other components. Technically, quartz crystal is the basic component of digital clocks. Crystals have the ability to vibrate back and forth with a constant frequency whenever an electric charge is passed through them, and exhibit a phenomenon called piezoelectric effect. This means that whenever a crystal is being deformed (stretched or compressed), an electrical charge is developed across its surface. It is also what we call quartz. The quartz crystal is usually made in the shape of a tuning fork, whereas the quartz is sandwiched between two conducting plates. These devices can relatively accurately vibrate with fixed frequency, e.g. 32,768 Hz.

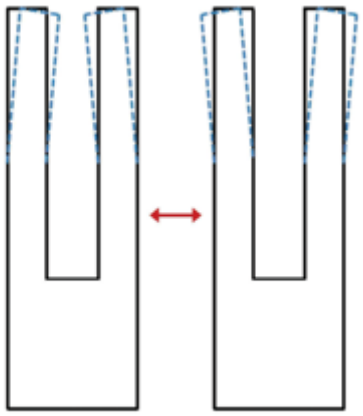

Figure 1. Shape of tuning forks used in a quartz clock. [2]

(c) Counters

Counters keep track of the number of vibrations (oscillations) the oscillator has made.

An electric charge is sent from the source of the circuit which is usually a battery, to the oscillator, making it vibrate 32.768 times a second. That means that the counter counts a second every 32,768 vibrations. The vibrations are converted into an electrical pulse, and the counters keep track of the time. Dividing the frequency multiple times results in parts of the second, e.g. ms, us, etc. In general, higher frequencies lead to more accurate clocks.

However, quartz clocks are not absolutely accurate. For instance, most of the quartz clocks lose or gain 1 or 2 seconds per day. In fact, the oscillator also changes its frequency slightly in different temperatures. This actually depends on factors such as local temperature, variations in the power supply, and noise coupling to other signals. But even if the oscillator keeps a constant temperature, the electrical circuit causes the clock to lose/gain time for other reasons, such as device aging.

Losing time is measured with a scale of *parts per million*. An accuracy of one hundred parts per million (ppm), means the clock is losing/gaining up to one hundred seconds for every millions of seconds. Based on this, we can assume that two or more clocks cannot stay synchronized forever. Comparing the relative speed of two clocks, we distinguish between

- Clock Skew (or timing skew): The phase difference between two clocks. Clock skew is caused by differences in wire-interconnect length, temperature variations, variation device imperfections, and differences in capacitance. This leads to different ways of measuring the time. Clock skew can lead to Setup and Hold Violation in flip flops. Hold violation is when clock travels slower than the path between two registers, allowing data to be set in the registers while at the same clock tick. As a result, the data can be assigned in two registers at the same time, and possibly, as a consequence, the integrity of data can be destroyed. Setup violation is when the destination flop receives a clock tick earlier than a source flop but the data signal has not reached the destination flop before next clock tick. As a result, the new data would not be assigned to the registers.
- Clock Drift: The clock frequency difference between two clocks or between a clock and a nominal perfect reference clock per unit of time of the reference clock. Clock drift can be caused by different effects, such as variations in device characteristics due to component aging, or temperature change. This can be a positive if Clock1(s) – Clock2(s) > 0, or negative if Clock1(s) - Clock2(s) < 0. If Clock1 - Clock2 is higher than zero then we can say that Clock1 is preceded by N seconds, where N = Clock1(s)-Clock2(s).

- Clock Jitter: It is the deviation of a clock edge from its ideal time point.

Ideally two clocks are totally synchronized, if both run at the same frequency, their offset to be zero, and their accuracy (jitter) is 0 ppm.

2.2 Atomic Clocks

Atomic Clocks are made of Cesium or Rubidium isotopes that have the ability to oscillate at a high frequency. For example, Cesium-133 oscillates at a frequency 9,192,631,770 Hz. That is the most precise definition of a second in SI units nowadays, although more accurate atomic clocks based on strontium (not part of current standard) have an accuracy of one second in 15 billion years, i.e. longer than the estimated age of the universe. Cesium atoms have two states, the high energy state (also known as Cesium I Ground State), and a lower energy state (also known as Cesium II Ground State). Atoms at the lower energy state can change to the higher energy state by absorbing radiation energy.

A Cesium atomic clock has a quartz oscillator that oscillates at the frequency of 9,192,631,770 Hz. However, as mentioned in a previous section, quartz oscillators can deteriorate because of parameters such as temperature, aging etc. Whenever the quartz oscillator is loses or gains oscillations, this is detected by a detector which controls the quartz oscillator.

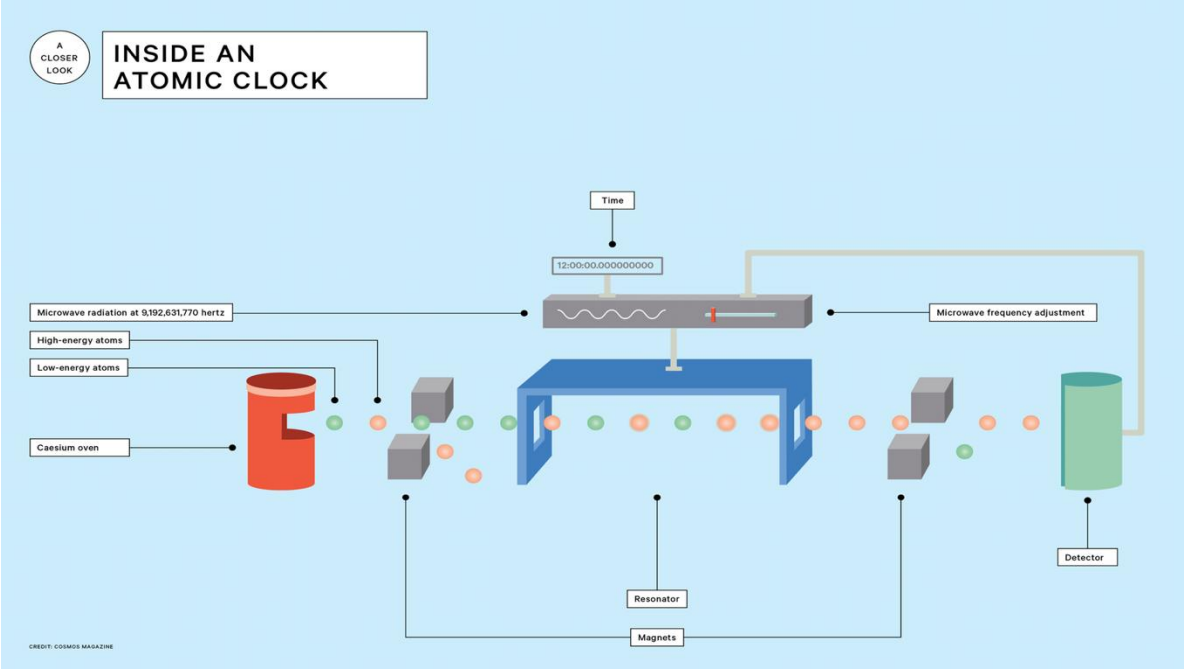

Figure 2. Implementation of an atomic clock [7].

The Caesium oven shown in the middle of Figure 2, heats the caesium-133 atoms. The magnets after the Caesium oven distinguish the low energy state atoms from the high energy state atoms. The low energy atoms then pass through the resonator (only the low energy atoms are used as these are the only that can change state). The resonator hits the low energy atoms with microwaves of frequency 9,192,631,770 Hertz, and converts them to high-energy atoms if the frequency is right. Then, the atoms pass through magnets again but this time only the high-energy atoms hit the detector. If the detector traces gaps, this means that not all the low-energy atoms were converted to high-energy atoms. Therefore, it means that the wave transmitter of the resonator does not transmit the correct frequency. The detector then sends an electrical signal correcting the frequency to the waves of the resonator until a constant fleet of caesium-133 hits the detector. Caesium clocks have an accuracy of 1×10^{-6} ppm.

2.3 Clock Synchronization

2.3.1 Clock Synchronization with physical clocks

In systems which need high timing accuracy, such as hard real-time systems maintaining a global clock is important. It is impossible to synchronize clocks in a network perfectly, as the delay that the message is being transported to another node is unstable. Even if the delay was measured, the clocks cannot be perfectly synchronized, due to timing variations.

The basic goal of physical clock synchronization, is to minimize the clock skew, i.e. the phase offset of the two clocks, no matter what distance they are apart. Even so, this "network distance" is unstable, and there is no way to predict the maximum delay that packets in a network take to be transmitted. This is because, depending on network congestion, packets have to be re-transmitted again and again (by the system or the application) until the receiver finally gets an acknowledgment. In fact, network packet delay not only depends on network congestion, but also on random system events such as kernel interruptions, scheduling etc.

Hence, a synchronization algorithm attempts to limit the clock skew by placing an upper and lower bound. Whenever the synchronized clock surpasses an upper or lower threshold value, the system tries to synchronize the clock again.

Assuming $\rho > 0$ is a constant known as the maximum drift rate, the upper bound can defined as:

$$\frac{\mathrm{dC}}{\mathrm{dt}} \leq 1 + \rho$$

And the lower bound is defined as:

$$\frac{dC}{dt} \geq 1 - \rho$$

A perfect clock would ideally have:

$$\frac{dC}{dt} = 1$$

This means that for any universal time *t*, an ideal clock *C* would have $C(t)=t$.

.

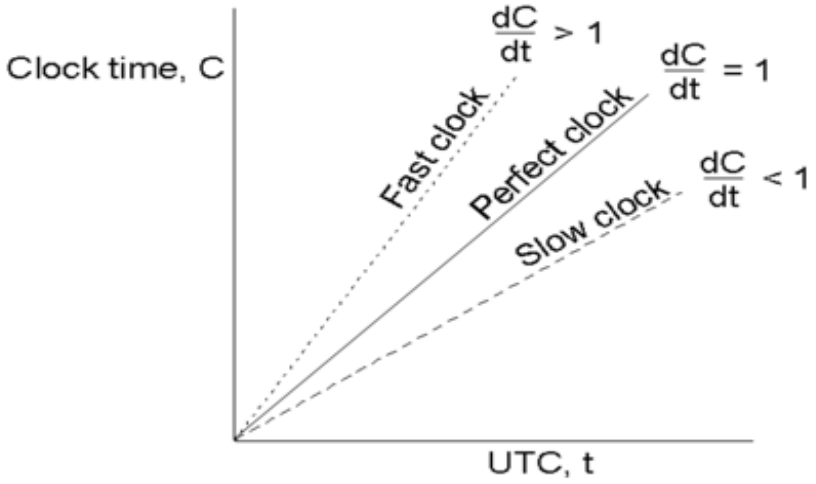

Figure 3. Variations in clock drift [4].

2.3.1.1 Cristian Algorithm for Physical Clock Synchronization

Cristian algorithm limits the drift by sending messages with timestamps and synchronizing clocks periodically. Assuming, there is a server, which has the reference clock, a client is connected to the server and sends a message. When the server receives a message, it immediately replies including his timestamp Ts. Without loss of generality, if we assume that

minRd, < $RTTi$ - *minRd,* the client receives the server's timestamp, then the correct synchronized time is in the range:

$$[t + minRd, t + RTTi - minRd],$$

where $minFd$ is the minimum forward delay, $minRd$ is the minimum reverse delay, and $RTTi$ is the round-trip time of the Mi message.

In this algorithm, the client sets his clock as the average of this range:

$$T' = t + \frac{RTTi}{2}$$

If the error is too high, then the client takes multiple readings to make an estimation. In this algorithm, it is allowed to increase or decrease the speed (frequency) of the clock. Normally, clocks can only be increased, since decreasing a clock may violate event ordering which is critical for certain system algorithms and programs, such as make. Instead, in this case, the time is frozen and an extra delay is made.

2.3.1.2 The Berkeley Algorithm

In this algorithm, no machine has an accurate time source, therefore, the clients ask for the server clock in order to set their clock periodically. Each of the clients send its clock to the server, the server averages the clocks and sends a new time to the client to adjust. The communication delays are measured as in Cristian's algorithm. The clocks are adjusting periodically. In this algorithm, clock adjustment can take negative values. For example, a client can adjust its clock by -1600 ms from its previous time. When a server cannot complete this procedure, another server is elected. This server may also have an atomic clock or wwv receiver.

2.3.2 Clock Synchronization with logical clocks

With logical clocks processes agree on the order of the events, rather than sharing the same global clock time. Every message that is sent to a process has a sequence number,

This type of synchronization can only be applied in systems that a global clock doesn't matter at all, and only the event order is important for consistency. Next sections introduce Lamport's algorithm and vector clocks. Both of these algorithms work with logical clocks.

2.3.2.1 Lamport's algorithm

Leslie Lamport introduced an algorithm for event based synchronization [3]. When physical clock synchronization is not necessary, the synchronization can be implemented using a logic called "Happened Before". This method focuses on the order that the events can occur, rather than synchronizing physical clocks. The expression y → x means that y event happened before x event, or x event must be processed after y event is done. Processes must follow these relations. The relation is transitive meaning if y → x and x→ z then y → z. For processes that do not communicate, this logic cannot be applied as the processes are considered either independent or concurrent.

Tasks must be time stamped in a way that all processes agree and keep the event order. If process P_1 has to complete event E_1 , and process P_2 has to complete event E_2, and E_1 has to be done before E_2 (E_1 → E_2) then E_1, E_2 must be synchronized in such a way $C_{E1} < C_{E2}$, where C_{Ei} is the timestamp that each event happened. Timestamp is an integer that can only be incremented. When the receiver receives a timestamp, it compares its local timestamp with the timestamp received. The maximum of these two timestamps incremented by one, will be the new timestamp of the receiver as shown in the following Figure.

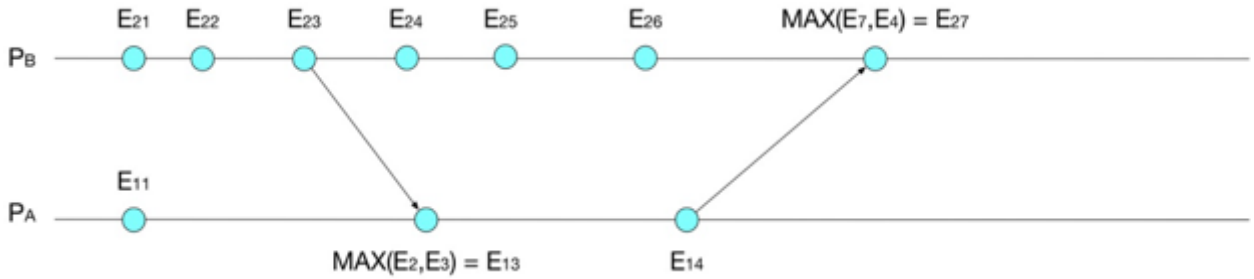

Figure 4. Causal ordering of events per process or among processes.

2.3.2.2 Vector Clocks

The problem with Lamport algorithm is that only every two pair of events can be ordered. Lamport solves the problem of ordering synchronization that keeps causal relationships. For example, if process A sends a message to process B, then, process B send a message to process C, the message that is being sent to C, is causally related to the message that A sent, because

process B could have reacted to that message (as seen on Figure 5). Vector Clock is an extension of Lamport algorithm to impose total ordering among logical clocks.

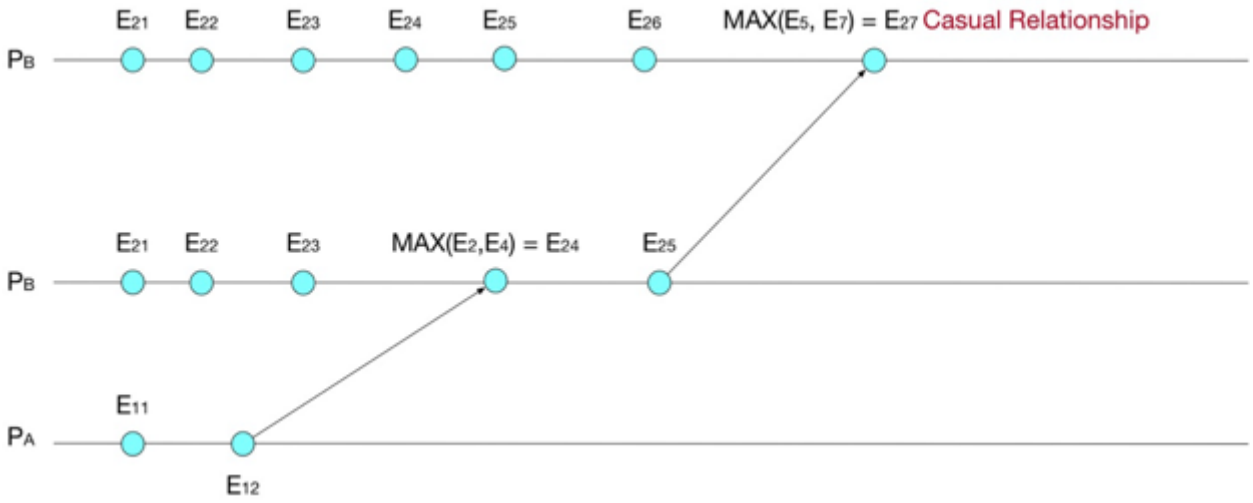

Figure 5. Example of causal relationship.

Each message carries a vector with all the processes clocks. For N processes, N clocks are in the vector V (T_1, T_2, ..., T_N). Vector V is a set of logical clocks. Suppose that process P_s is the sending process and P_r is the receiving process. These two processes have a logical clock on the set V. When process P_r receives an event, the logical clock of P_s in V set is increased by one, and the logical clock of P_r gets the maximum value between the element P_r of the received vector V or the element of the local vector V that P_r has. This algorithm is an extension of Lamport's algorithm. An example of Vector Clocks algorithm is represented in Figure 6.

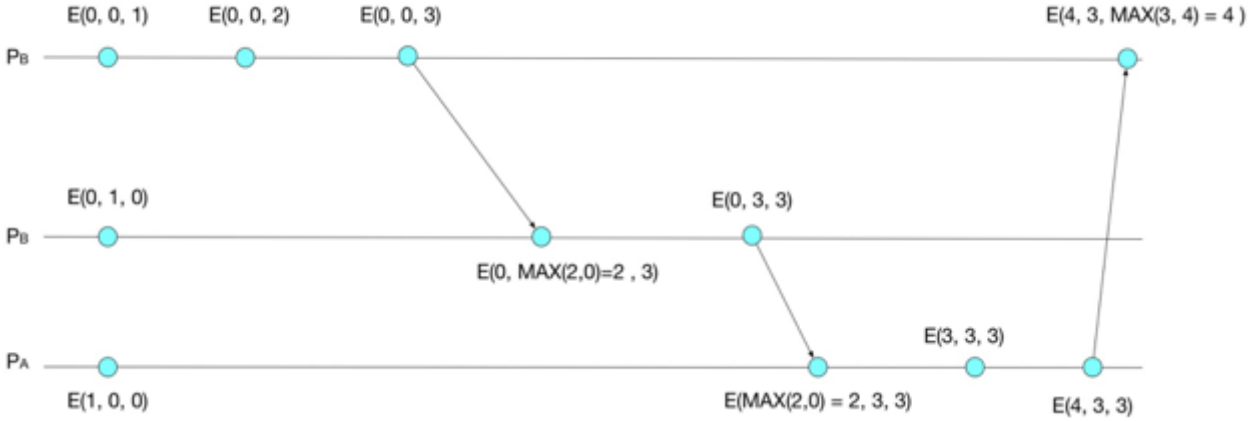

Figure 6. Example with Vector clocks.

2.3.3 External Clock Synchronization

This group is used for a node or a set of nodes which are synchronized with an external time source. The external time source has usually UTC as reference clock.

2.3.3.1 Network Time Protocol

Network Time Protocol (NTP) [9] is used to ensure that all computers in a network are synchronized to a reference clock that can be an atomic clock following the coordinated universal time. But as atomic clocks are expensive, it usually uses Global Positioning Time (GPS) network or wwv receiver. Computers uses NTP nowadays to synchronize time. But not only computers use NTP. It is also used by routers, telephone systems etc. NTP has a hierarchy, where at the highest-level, Stratum-1 has a reference clock. The time is passed to Stratum-2 NTP servers, who in turn synchronize their internal clocks. Similarly, Stratum-3 connects to Stratum-2 devices and synchronizes their clock. This can be done until Stratum-16, after that the time is considered 'false'. The time that passes on each Stratum level increases the error.

T1 timestamp travels from Stratum-N to nodeX, on nodeX the packet will be received at time T2, then nodeX sends back to Stratum-N at time T3 a message (Figure 7 represents the timestamps). When Stratum-N receives that message at time T4, the network delay as:

$$d = (T4 - T1) - (T3 - T2)$$

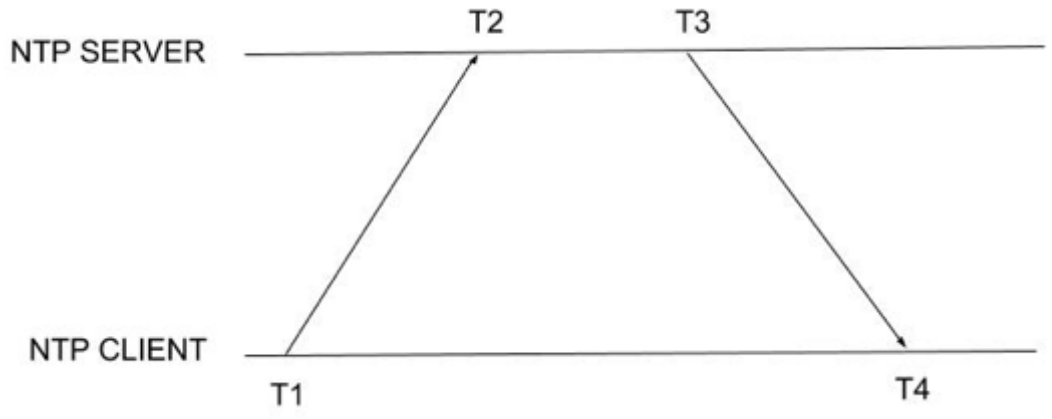

Figure 7. NTP delay measurement.

The NTP packet is 64 bits in total, the first 32 bits contains seconds, and the rest 32 bits is in this format:

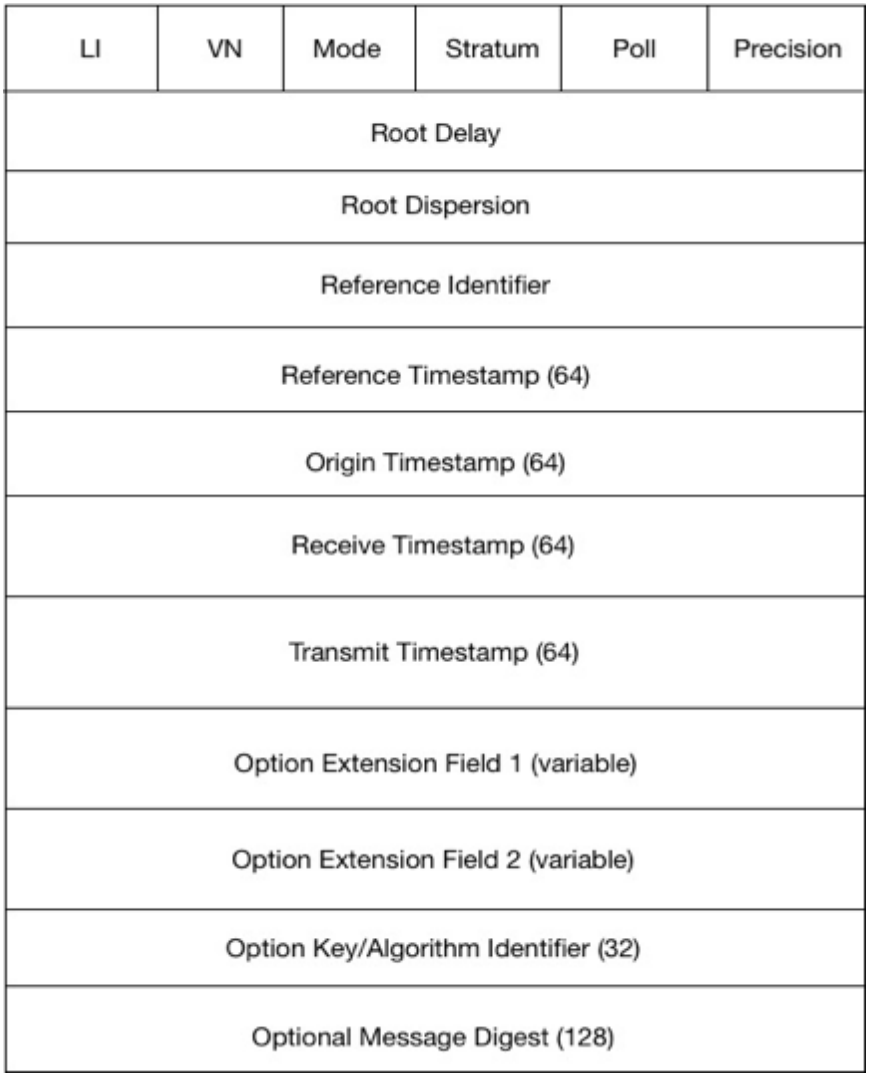

Figure 8. NTP Packet.

- LI Section (2 bits) Contains, if a leap second is applied. This can be 0 if there is no leap adjustment, 1 if the last minute of the day is by a second higher (61 seconds in total), 2 if the last minute of the day is by a second lower (59 seconds in total), 3 if the clock is considered unsynchronized.
- VN Section (3 bits) contains the ntp version.
- Mode Section (3 bits) contains the ntp packet mode, can take values from 0 to 7.
- Stratum Section (8 bits) contains the stratum, can take 0 if its unspecified, 1 if the timestamp comes from the stratum-1, 2-15 is if the timestamp is from the secondary server. Stratum-16 is considered unsynchronized, and provides false clock.

- Poll Section (8 bits) contains the poll interval. This describes the maximum waiting time until the protocol starts synchronizing again.
- The last 8 bits contain the precision of the system clock.

2.3.3.2 Global Positioning System

Global Positioning Systems (GPS) has four atomic clocks that are synchronized as accurately as possible to UTC. It is considered Stratum-0 for the NTP and plays the role of reference clock in many cases. The time signal is received from a time server or the local master and is passed to other devices or networks. Accuracy of GPS synchronization ranges from better than microsecond to few ms depending on the synchronization protocol.

3. One-way delay Algorithms

3.1 Describing Round-Trip Time and One-Way Delay

As mentioned in the previous sections, clock synchronization of nodes in a network is difficult due to network instabilities. In this section, we focus on the network delay.

Network delay is written as:

$$TotalDelay = Transmission\ Delay + Propagation\ Delay + Processing\ Delay + Queueing\ Delay$$

where:

- *Queueing Delay* is the time that the packet spends in the routing queues.
- *Transmission Delay* is the delay that the bits are put into the link.
- *Propagation Delay* is considered the time, when bits are travelling through the wire to another node.
- *Processing Delay* is considered the time that the router takes to process the header.

Processing and queuing delays are unpredictable due to variability in processing routines in routers and network conditions.

When Node A sends a message to Node B through a router (see Figure 9), then the delay becomes:

$$TSend[A \rightarrow B] = PR_{A1} + Q_{A1} + TR_{A1} + PD_{A1} + PR_{AB} + QRouter_{AB} + TR_{AB} + PD_{B1} + Q_{B1} + PR_{B1} \quad (1)$$

where $PR_{A1}$is the Processing Delay on Node A, Q_{A1} is the queuing delay on the socket buffer of node A when sending message to node B, TR_{A1} is the transmission delay when A sends a message to B, PD_{A1} is the propagation delay of A when sending message to node B, PR_{AB} is the processing delay of the router to forward the packet to node B, $QRouter_{AB}$ is the queuing delay of the router when node A sends a message to node B, TR_{B1} is the transmission delay of router when Node A sends a packet to node B, PD_{B1} is the propagation delay of the packet to arrive at

node B from the router, and finally Q_{B1} is the queuing delay of the socket buffer at node B.

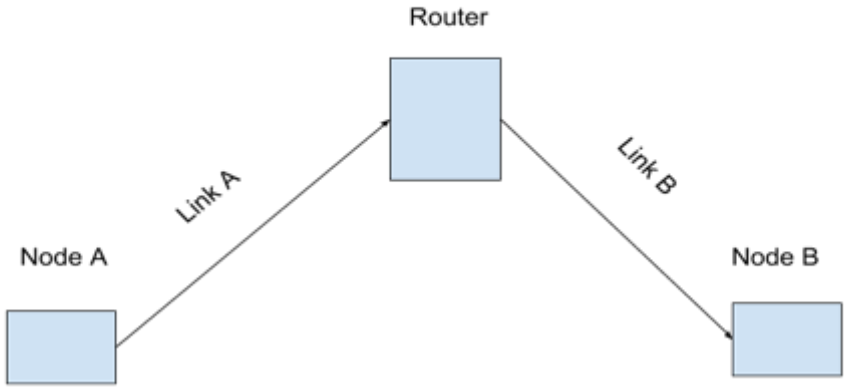

Figure 9. Node A sends message to node B

Similarly, when Node B sends a message to Node A the delay is:

$$TSend[B \rightarrow A] = PR_{B2} + Q_{B2} + TR_{B2} + PD_{B2} + PR_{BA} + QRouter_{BA} + TR_{BA} + PD_{A2} + Q_{A2} + PR_{A2} \quad (2)$$

Where PR_{B2} is the Processing Delay on Node B , Q_{A2} is the queuing delay on the socket buffer of node A when sending message to node B, PD_{A2} is the propagation delay of A when sending message to node B, PR_{BA} is the processing delay of the router to forward the packet to node B, $QRouter_{BA}$ is the queuing delay of the router when node B sends a message to node A, PD_{B2} is the propagation delay of the packet to arrive at node A from the router, and finally Q_{A2} is the queue delay of the socket buffer at node A.

Based on the definitions above, the round-trip time is:

$$Round\ Trip\ Time = TSend[A \rightarrow B] + TSend[B \rightarrow A].$$

where $TSend[A \rightarrow B]$ is considered the forward delay and $TSend[B \rightarrow A]$ the reverse delay, which are also known as one-way delays.

One of the basic assumptions nowadays is that the network path is symmetric. In other words, the assumption is that $TSend[A \rightarrow B]$ and $TSend[B \rightarrow A]$ both equal to half the RTT. Nowadays, this is not an accurate assumption, and often the network delay in the forward

direction is orders of magnitude larger than the reverse [6]. Even if same size packets followed the same route, the delay would be asymmetric, as it passes through different queues.

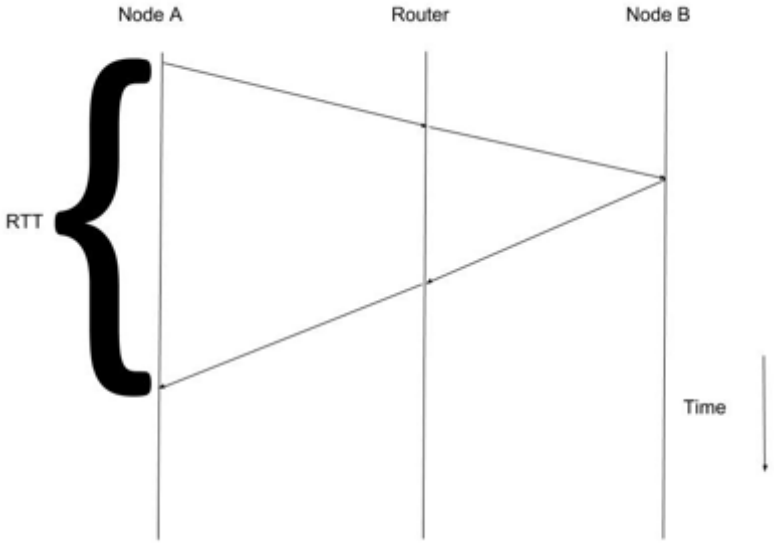

Figure 10. Symmetric delay.

Ideally if the one-way delays are symmetric, forward delay ($TSend[A \rightarrow B]$) and reverse delay ($TSend[B \rightarrow A]$) equal to half of round-trip time (as seen on Figure 10).

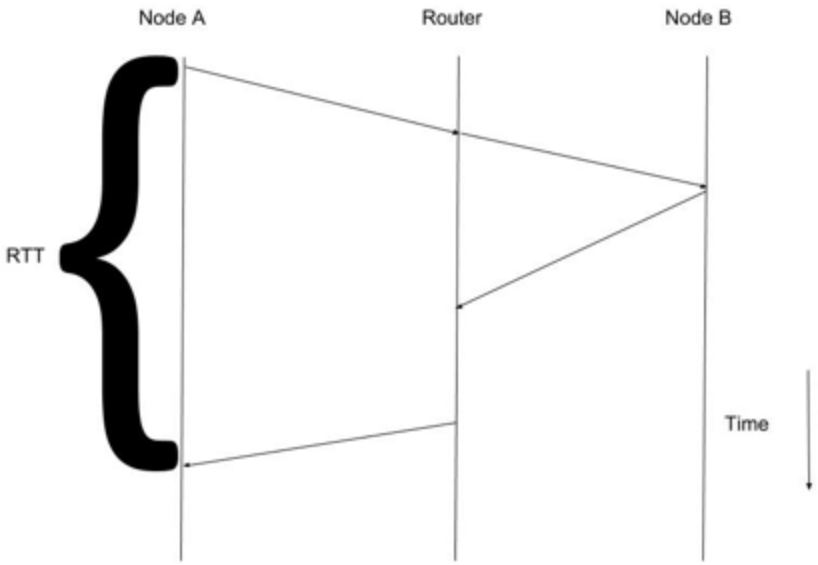

Figure 11. Asymmetric delay.

In Figure 11, an asymmetric example is presented. The packet has a asymmetric delay at the router queue due to processing on the router node. Thus,

$$QRouter_{BA} \neq QRouter_{AB}$$

or

$$PR_{BA} \neq PR_{AB}$$

Hence, $TSend[B \rightarrow A] \neq TSend[A \rightarrow B]$, and the one-way delay cannot be half the RTT. A main difficulty in measuring one-way delay is that network clocks are not synchronized. If the clocks were synchronized, then the exact one-way delay would be measured as follows.
Node A sends a packet with a timestamp T_1 to another end-node. When the other node receives the packet at time T_2, it computes $T_2 - T_1$ which is considered as the one-way delay.
Doing this on unsynchronized nodes creates errors, since the one-way delay measurement will include the clock offset, which is unknown between the two nodes.

3.2 Ping-Pong-based Technique

Yoo and Choi [5] proposed one-way delay estimation in relation to round-trip times. In their method, they did not focus on clock skew, assuming that time delay between sending and receiving packets is independent of clock skew.
By considering network calculus equations [5], they developed a system of two equations:

$$t_n = t_0 - \sum_{i=1}^{n} RTT(s,i) + \sum_{i=1}^{n} RTT(r,i) \qquad (3)$$

where t_n is the forward delay of the nth packet, RTTs is the round-trip time that the sender measures and RTTr is the round-trip time that the receiver measures.

$$k_n = -t_0 + \sum_{i=1}^{n+1} RTT(s,i) - \sum_{i=1}^{n} RTT(r,i) \qquad (4)$$

where k_n is the reverse delay.

Round-trip time $RTTS_i$ on the sender side is measured with a ping-pong procedure. The sender sends a dummy packet at time Ts_i to the client and receives an acknowledgment (same size dummy packet) at time Tr_i. Each round-trip time on the server side is equal to:

$$RTTS_i = Ts_i - Ts_{i-1} \qquad (5)$$

The receiver (client side) receives a message at time Tr_i. The round-trip time on the client side is equal to:

$$RTTR_i = Tr_i - Tr_{i-1},\ i > 1 \qquad (6)$$

For the first packet on the client side, the round-trip time is not measured. Just a timestamp $Tr_1$is taken.

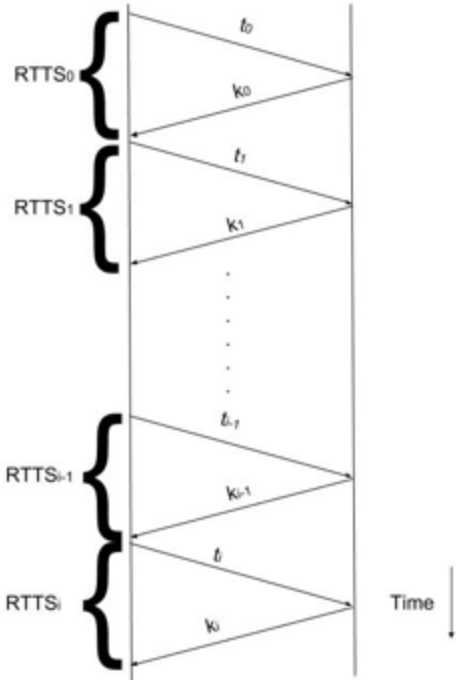

Figure 12. Round-Trip Time measurements based on [5].

Equations (3), (4) can be simplified. By subtracting them we get:

$$t_n + k_n = RTTS_{n+1} \qquad (7)$$

Also k_{n-1} is equal to:

$$k_{n-1} = -t_0 + \sum_{i=1}^{n} RTT(s,i) - \sum_{i=1}^{n-1} RTT(r,i) \qquad (8)$$

By adding (3), (8) we get:

$$t_n + k_{n-1} = RTTR_n \qquad (9)$$

With equations (7), (9) one-way delay estimation can be measured while the algorithm is sending and receiving packets without re-computations. Both sides, client and server, must be symmetric

when doing computations. Asymmetries in any side bring inaccuracies, as the computations in between

Ts_i and Ts_{i-1} or Tr_i and Tr_{i-1} make $RTTS_i > RTTR_i$ or $RTTR_i > RTTS_i$ most of the time.

In the presence of asymmetries, the sums in equations (3) and (4) have an increasing trend, and with n packets forward or reverse delay can get negative values. Equations are also very sensitive to the constant t_0, which can also bring (5) or (7) to negative values. One-way delay can't be negative.

3.2.1 Implementation using POSIX Socket API on Ethernet

For the implementation of one-way delay estimation algorithm, TCP sockets are used. It is important to guarantee that packets are delivered undamaged, in the same order that are sent.

On the server side, we do these 4 stages:

- **Socket** – creates a new socket and a new descriptor that is assigned to the socket. By creating the socket, we must define the protocol that will be used. For the implementation TCP sockets are used.
- **Bind** – connects a local network address with the socket.
- **Listen** – Listen call indicates to the protocol that the server is ready to accept any incoming connections.
- **Accept** – Accepts and establishes a new connection with a client socket wants to connect.

On the client side, we do these 2 stages:

- **Socket**
- **Connect** – it is called from the client process to connect a server process.

After establishing a successful connection `send ()` and `recv()` calls are used to send and receive messages between the processes.

The packet that is being transferred to the network is defined as:

```
struct networkPacket
{
  Char packetType[300];
  Char packetInfo[200];
  uint16_t packetId;
```

```
    uint32_t time;
};
```

Parameter

Description

1. packetType

Describes the type of packet the client or the server is sending, as one of the following values:

PacketType Value	Description	From->To
RTT_REQ	Packet passed to server process for measuring round-trip time	Client->Server
RTT	Packet received from client is a round-trip packet. Client measures Round Trip Time using Choi's formulation and passes it back again to the server.	Server->Client

2. packetInfo

Contains data that a side (client/server) wants to pass to the other side. The value set depends on packetType as follows:

PacketType	***packetInfo*** values separated by commas
RTT_REQ	***NumberOfPackets***, indicates how many packets are going to be transmitted for the one-way delay measurement. ***CorrectionAt*** indicates the number of packets sent after which there will be a correction on the measured RTTs.
RTT	Empty

3. packetId

Contains the packet ID of each packet.

4. time

Contains time values in microseconds.

TCP is a stream based protocol, and for this reason we have set the flag MSG_WAITALL on each recv (). MSG_WAITALL sets recv () to blocking mode and waits until all *(n)* bytes are received.

```
recv (int sock, char *buffer, n, MSG_WAITALL)
```

On the current implementation *n* is equal to the network packet size in bytes, i.e. *sizeof (struct network Packet).*

When declaring a socket, we can use options to affect the way the socket is used. By using **setsockopt** () we can define the socket behavior based on some parameters.

```
int setsockopt( int socket, int level, int option_name,
                const void* option_value, socketlen_t option_length);
```

Parameter

Description

1. `socket`

The socket descriptor.

2. `level`

The protocol level on which the option will be set. The options that we will use will be on TCP level and the corresponding parameter is set to IPPROTO_TCP.

3. `option_name`

The socket option for which the value is to be set. This parameter also depends on the socket level. Different socket levels have different options.

4. `option_value`

A pointer to a buffer from which the value for the `option_name` will be passed.

5. `option_length`

Size of the buffer indicated in `option_value` (in bytes).

The options that we will use are:

- **TCP_NODELAY**

Disables Nagle's algorithm. Nagle's algorithm detects small size packets and "encapsulates" them into bigger TCP packets before sending them across the network. After disabling it, an acknowledgement for each previous packet must be received, before next packet is sent.

In brief, if Nagle's algorithm is enabled, it enqueues a large amount of data and then sends it. However, waiting for a larger amount of data will not benefit some applications. Nagle's algorithm could bring asymmetries if the queues are not of the same size. Even if the queues were of the same size, different processing delay in the nodes could bring asymmetries.

- **TCP_CORK**

TCP_CORK option is to disable Nagle's algorithm. It forces the application to send full frames, and not parts of frames, and get an acknowledgement for the data sent.

For the algorithm implementation, it is important that both sides send the same amount of data, and that data blocks at both sides be equal. Also, packets must not stay in a socket buffer for a long time. As soon as a packet enters the socket buffer, it must be immediately being transmitted. Finally, we must avoid complicated computations on one side, which can bring inaccuracies on measuring time delays due to asymmetry.

The procedure of measuring Round-Trip Time and One-Way Delay on Ethernet based on Yoo and Choi's technique is as follows [5]:

For Sender[server]:

1. Receive an RTT_REQ type packet with the number of packets (NumberOfPackets) for the estimation and the correction (CorrectionAt) value.
2. **For i=0 to NumberOfPackets with step=1 do**
3. Send packet to the Sender.
4. Receive packet from Sender [Packet]
5. Get Timestamp Value T2
6. If i is equal to 0 set RTTs equal to zero [T1=T2]
7. Else:
8. Measure the difference T2-T1 [RTTs]

9. Add to RTTs_sum the measured RTTs
10. Add to RTTr_sum the measured RTTr (Packet.time)
11. T1=T2
12. If i % CorrectionAt equal to zero do
13. RTT_r_Phase = RTTr_sum / CorrectionAt
14. RTT_s_Phase = RTTs_sum / CorrectionAt
15. If firstPhase do
16. k = RTTs/2 [in microseconds]
17. t=k
18. firstPhase = false
19. end if
20. else :
21. t = RTT_r –k
22. k = RTT_s – t
23. end else
24. RTTr_sum = 0
25. RTTs_sum = 0
26. End if
27. End Else
28. **End for**

For the Receiver[client]:

1. Send an RTT_REQ type packet with the number of packets (NumberOfPackets) for the estimation and the correction (CorrectionAt) value.
2. **For i=0 to NumberOfPackets with step=1 do**
3. Receive packet from Receiver [Packet]
4. Get Timestamp Value T2
5. If received packet type is RTT do
6. If received packet id is 0 do
7. set T1=T2

8. end if
9. else:
10. RTTr = T2-T1
11. T1=T2
12. End else
13. Send packet to the Sender containing RTTr.
14. End if
15. **End for**

The procedure of measuring Round-trip time is represented in Figure 13.

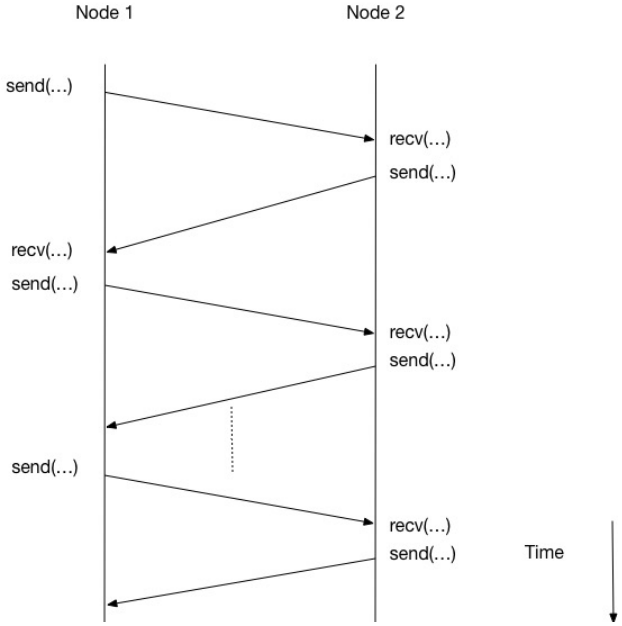

Figure 13.Posix Socket round-trip times based on [5]

The full algorithm implemented on top of TCP is provided in Appendix A1.

3.2.2 Implementation on CAN Bus Protocol

The algorithm by Yoo and Choi was also implemented and tested in a real-time CAN bus network. We have connected AVR microprocessors (Arduino UNO) equipped with a CAN Bus with DFROBOT Can Bus Shield (DE9 connectors). DFROBOT also gives the required libraries in C++. MCPCAN class has a constructor that creates an instance from which you can send and receive messages on CAN-BUS through corresponding member functions.

```
MCPCAN (INT8U _CS);
```

Parameter `_CS`, is the pin that connects the Arduino with the CAN Bus shield which is used to control interrupts during packet (standard frame) communication [Figure 13]. Each Arduino node listens to a set of message IDs. Lower ID values have higher priority for transmission.

The function for sending messages is defined as:

```
INT8U sendMsgBuf (INT32U id, INT8U ext, INT8U len, INT8U *buf);
```

where `id` is the id that the message has been sent, `ext` is 1 for extended packet (29-bit IDs) or 0 for non-extended packet (11-bit IDs), `len` is the length of the buffer to be transmitted, and `*buf` is a pointer where data is strored.

The function for receiving messages is defined as:

```
INT8U readMsgBufID (INT32U *ID, INT8U *len, INT8U *buf);
```

where `*ID` is an INT32U pointer that contains the sender ID that the message is going to be received, `len` is the length of the buffer that is going to be transmitted and `*buf` is a pointer from where the buffer starts.

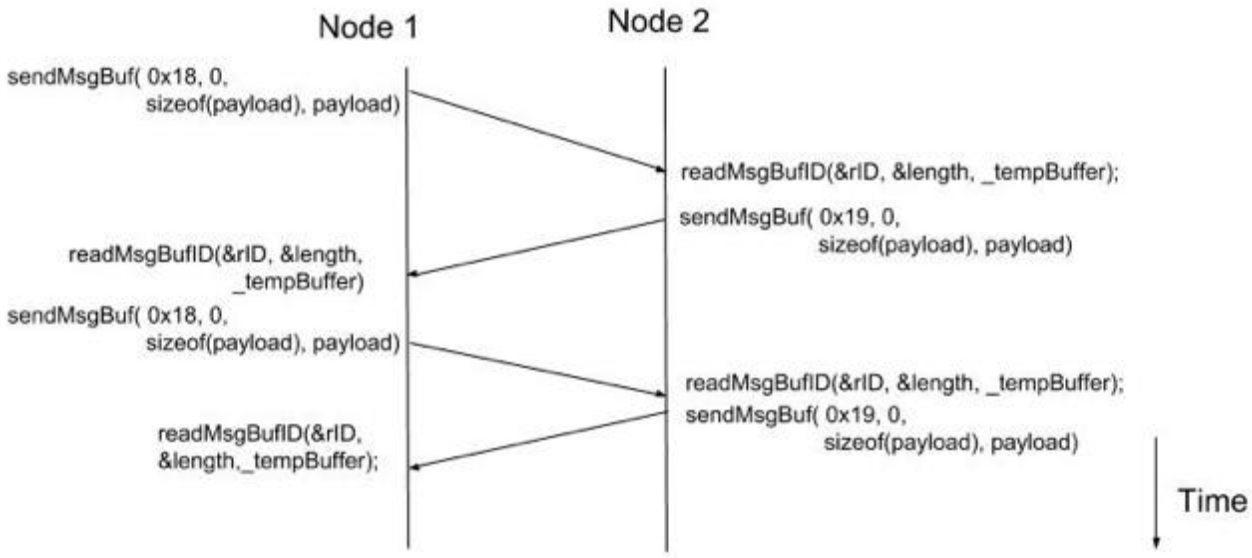

Figure 14. CAN-Bus round-trip times based on [5].

Figure 14, shows the one-way delay computation as implemented on the CAN Bus. Considering that Node 1 sends ID 0x18 and Node 2 sends ID 0x19, and Node 1 measures RTTs and Node 2 measures RTTr.

Due to the fact that the Arduino has limited resources, the packets are divided into phases. Arduino time measurements are very sensitive. If we compute t and k at the same time that the round-trip times are being calculated, even without any recomputation, the algorithm becomes unstable due to code asymmetries (or imbalances between sender (server) and receiver (client), leading to negative predictions for t or k. For this reason, in our algorithm (see pseudocode and C code in Appendix A2), round-trip times are first stored in an array, and then processed for computing one-way delay.

To understand better this issue, consider the alternative of running the commented code in Appendix A2 (which is similar to Ethernet, see Section 3.2.1. Using the same ping-pong process, we computed on-the-fly, without recomputation, both the forward delay t, and reverse delay k on CAN bus. However, as shown in Figure 15, this lead to an asymmetry, and as a result the forward (and reverse delay) was constantly decreasing (resp. increasing), leading to negative values (and reset).

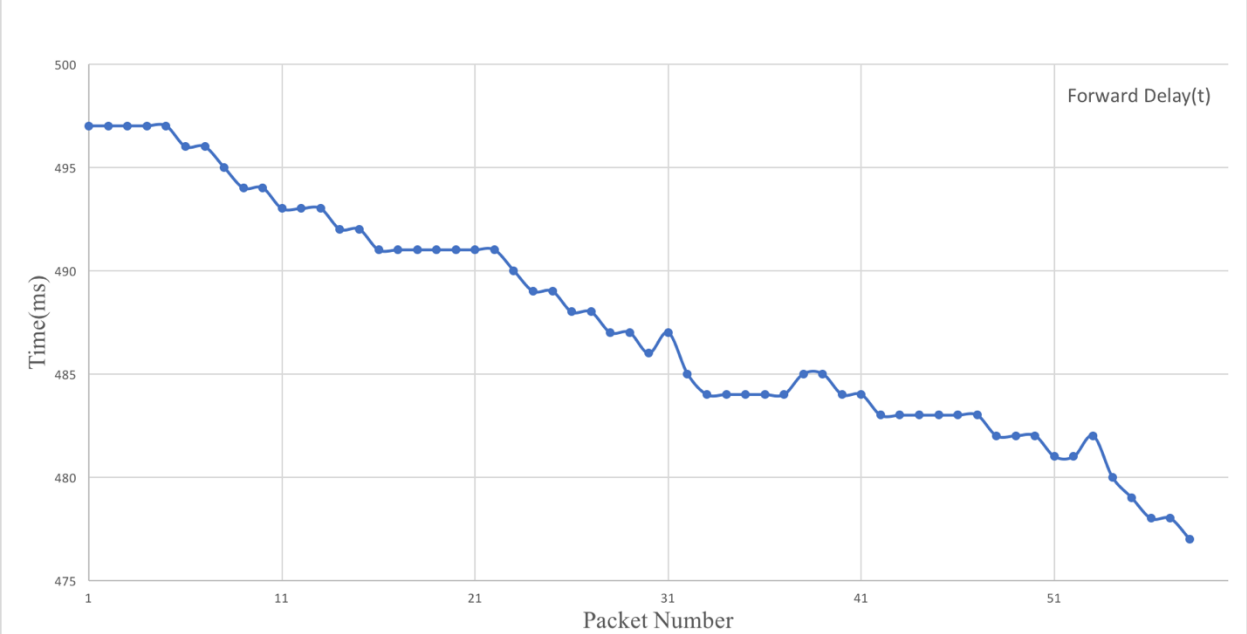

Figure 15. Decreasing Forward Delay(t) trend.

In the future, we hope to resolve these unbalances and develop an intermediate solution that computes the forward delay t, and reverse delay during runtime. However, for now, the new

proposed procedure to measure Round-Trip Time and One-Way Delay estimation on CAN Bus, based on the algorithm by Yoo and Choi [5] is as follows:

For RTTs (Node 1):

1. Send an empty packet to Node 2
2. Take a timestamp value. [timestamp T1]
3. Receive Packet that has been sent from Node 2
4. Send immediately a packet to Node 2
5. Take a timestamp value [timestamp T2]
6. Receive packet from Node 2 [Contains RTTr]
7. RTTs = T2-T1
8. T1=T2
9. rtt_receiver_arr[i] = RTTr
10. rtt_sender_arr[i] = RTTs
11. Goto 2, until i equals to the total number of phase packets.
12. For i=2 to PhasePackets with step = 1 do
13. t = rtt_sender_arr[i] – k
14. k = rtt_receiver_arr[i] – t
15. tsum+= t;
16. ksum+= k;
17. end for
18. T = tsum/ (PhasePackets-2)
19. K = tsum/ (PhasePackets-2)
20. Goto 4, until all the phases are done.

For RTTr (Node 2):

1. Send an empty packet to Node1
2. Receive response from Node 1
3. Take a timestamp value. [timestamp T1]

4. Send a packet that contains RTTr to Node1
5. Receive a response from Node1
6. Take a timestamp T2
7. RTTr = T2-T1
8. T1=T2
9. Goto 4

The full algorithm implemented on top of CAN bus is provided in Appendix A1. Notice that the code has been integrated in a network-layer secure infrastructure called vatiCAN-G which is beyond the scope of this work

3.3 Aoki, Oki, Cessa Model

Aokis, Okis and Cessa measure one-way delay variation [11], using a different method than Yoo and Choi's ping pongs [5]. The sender measures the time difference of between consecutive (timestamped) periodic sending events. For example, the sender (source) sends a packet at time $T_{send(i-1)}$ and then again, sends a packet at time $T_{send(i)}$. The inter-packet delay of the ith packet at source si:

$$TS(i) = T_{send(i)} - T_{send(i-1)}$$

In a similar sense, the receiver measures the difference of each receive event from the previous one. When the receiver receives a packet from the sender at time $T_{receive(i-1)}$ and next packet at time $T_{receive(i)}$, the inter-packet delay of the ith packet at the Receiver will be equal to:

$$IPD(i) = T_{receive(i)} - T_{receive(i-1)}$$

Figure 15 shows the procedure of measuring the inter-packet delays on each side.

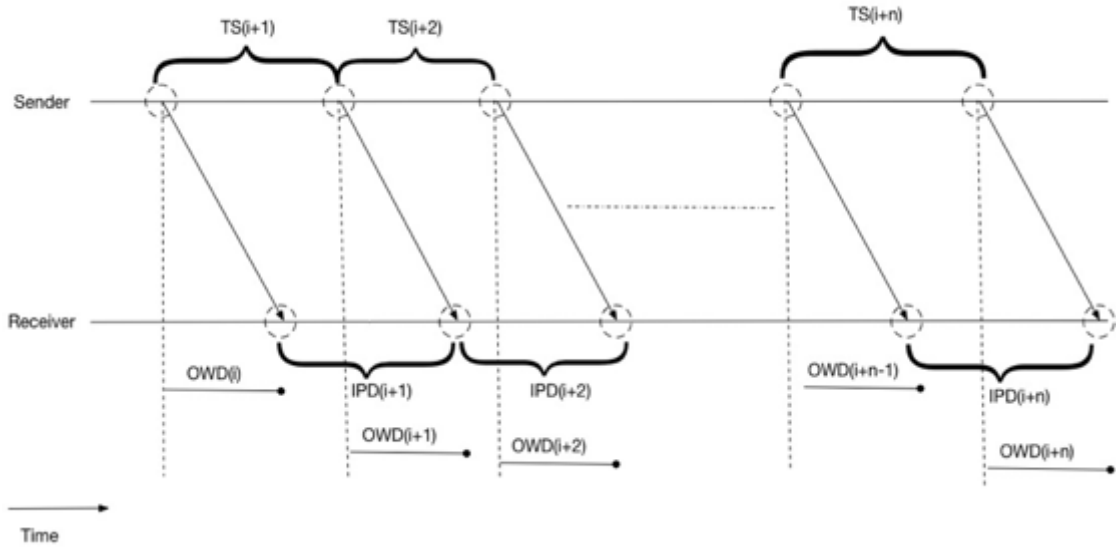

Figure 16. How inter-packet delays are measured [11].

One-way delay can now be expressed as:

$$OWD(i+n) = \sum_{j=1}^{n} (IPD(i+j) - TS(i+j)) + OWD(i)$$

and one-way delay variation as:

$$AOWDV(i+n) = \sum_{j=1}^{n} (IPD(i+j) - TS(i+j))$$

Sending too many packets in a small amount of time causes overhead to the socket buffer and the network. Due to that reason, the procedure of measuring the inter-packet delays is divided into time windows. From each window, the minimum value of AOWDV is selected. Then, applying linear regression to the minimum AOWDV values, we can find a clock skew estimation (the slope of the linear regression line). We then remove the clock skew (the slope of the linear regression line) from each one-way delay (OWD) value.

The algorithm for Ethernet (Figure 16) and CAN Bus (Figure 17) is the same:

The process for Node 1 is:

1. Send a packet to node 2 with TS.
2. previous_timestamp = current_timestamp

3. Get timestamp [current_timestamp]
4. TS = current_timestamp – previous_timestamp
5. Goto 1 until all the packets are sent.

The process for Node 2 is:

1. Receive a packet that has been sent from Node 1 [Node 1 -> Node 2]
2. Extract value TS
3. previous_timestamp = current_timestamp
4. Get timestamp [current_timestamp]
5. IPD = current_timestamp – previous_timestamp
6. OWDV[i] = IPD – TS
7. Add OWDV and Packet ID to the linear regression set.
8. Goto 1 until all the packets are sent
9. Subtract from the OWDV Values the predicted value calculated from linear regression.

3.3.1 Implementation using POSIX Socket API on Ethernet

For socket setup, a similar procedure to section 3.2.1 has been used.

The packet being transferred is defined as:

```
struct networkPacket
{
    char packetType[200];
    char packetInfo[500];
    uint16_t packetId;
    uint32_t time;
};
```

Parameter

Description

6. packetType

Describes the type of packet the client or the server is sending, as one of the following values:

PacketType Value	Description	From->To
TR	Client side sends a packet with the inter-packet delay.	Client->Server

7. packetInfo

Contains data that a side (client/server) wants to pass to the other side.

8. packetId

Contains the packet ID of each packet.

9. time

Contains time values in microseconds precision.

For this implementation, only the sender set to this variable its inter-packet delay.

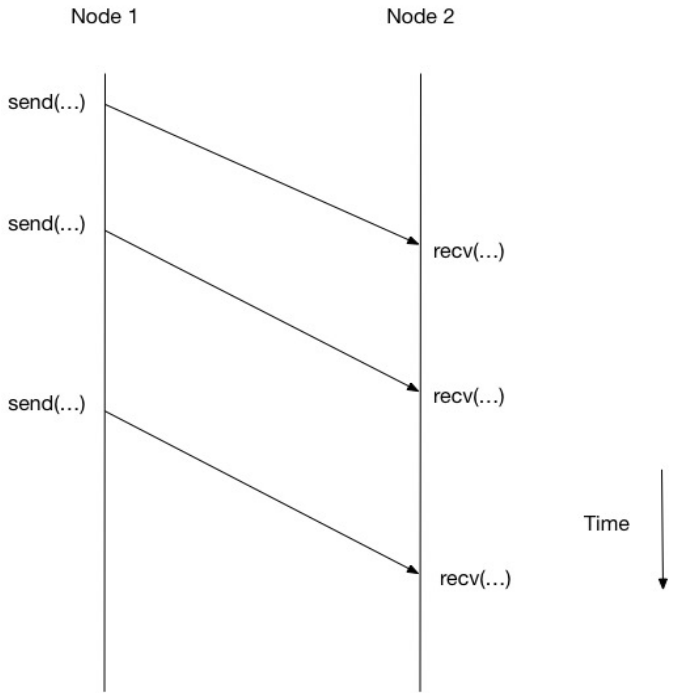

Figure 17. Ethernet implementation of one-way delay variation based on [11].

3.3.2 Implementation on CAN Bus Protocol

The setup in section 3.2.2 has been reused.

Node 1 measures TS and sends a packet with the TS value through CAN Bus to Node 2. Node 2 extracts TS, measures IPD and then doing the computations for OWDV. Next figure represents how the packets are being transmitted:

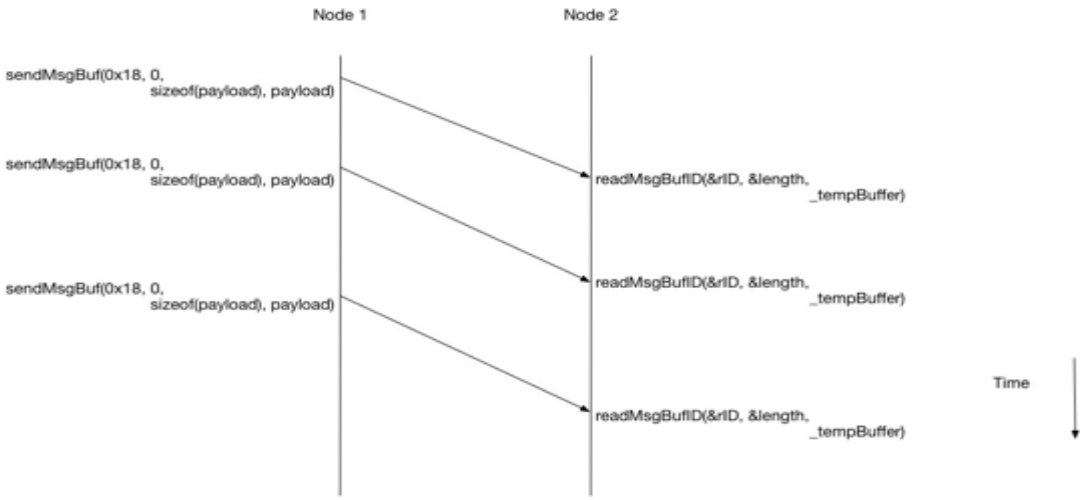

Figure 18. CAN-Bus implementation based on [11].

The full algorithm implementations for TCP and CAN bus are provided respectively, in Appendices A3 and A4.

4. Experimental Framework and Results

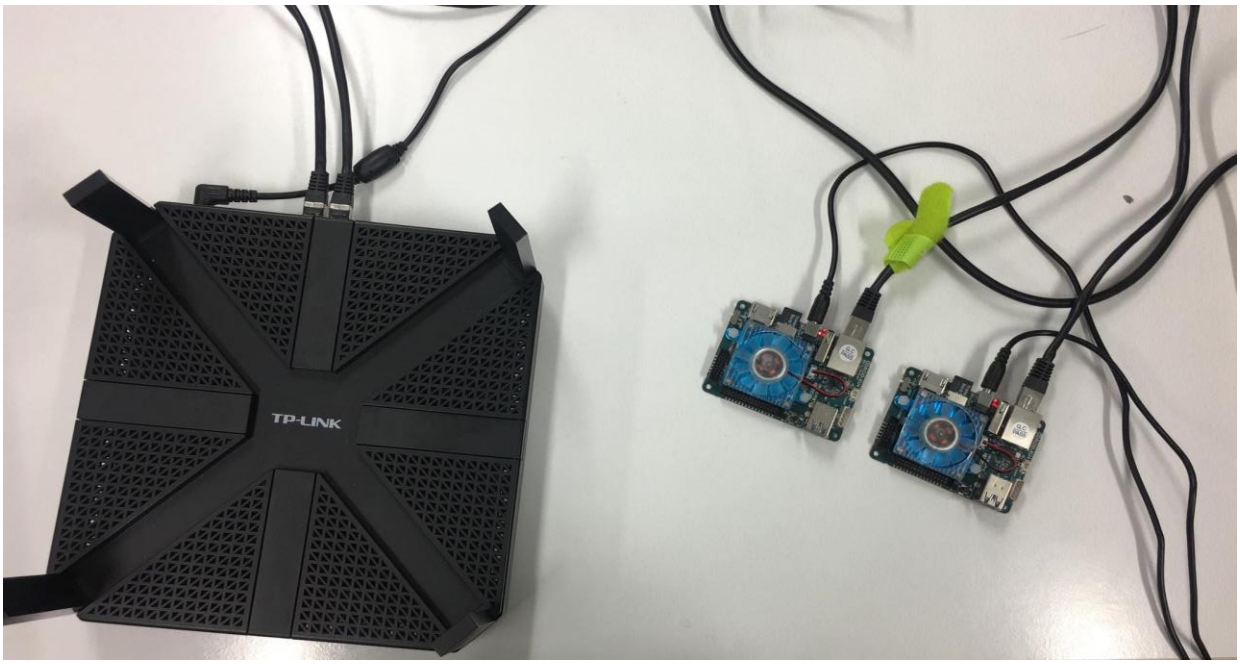

Figure 19. Odroid XU4's and Router experimental setup.

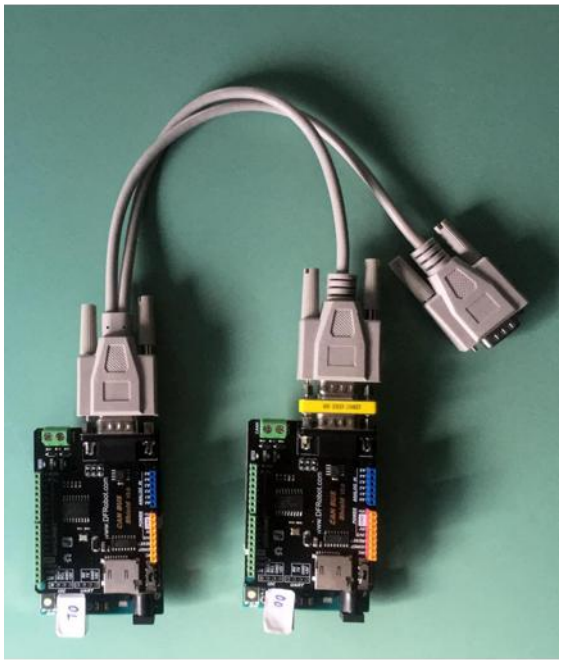

Figure 20. CAN Bus with two Arduino XU4 experimental setup.

For the experiments on the Ethernet, two Odroid XU4 and a router have been used (Figure 19). On the CAN Bus, 2 Arduino UNO R3 are connected together using with DFRobot CAN BUS Shield and DE9 interface cables (Figure 20).

In our first experiment, we compare the forward delay t with RTTs/2 [5], which many implementations tend to use. One of the Arduinos plays the role of a sender, while the other one the role of a receiver (e.g. see Figure 14).

- For CAN bus, we have used 18000 packets to measure round-trip times with 1800 phases, with each phase consisting of 10 packets. During each phase, we compute the one-way delay and compared with RTT/2. Figure 20 compares t and RTT/2 on CAN bus protocol.
- For Ethernet, two Odroid xu4 boards have been connected to a router. One Odroid plays the role of the client and the other the role of the server. We have similarly used 18000 packets to measure round-trip times and a phase window of 10 packets.

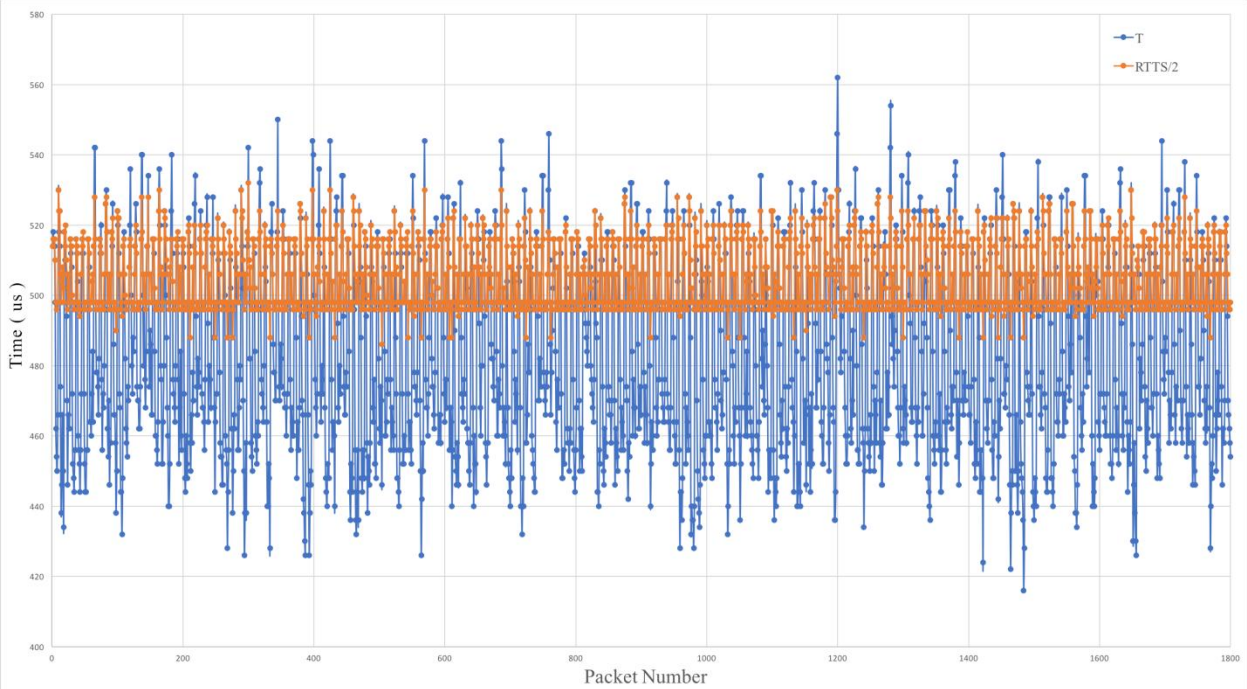

Figure 20. Forward delay (t) vs RTTs/2 on CAN bus.

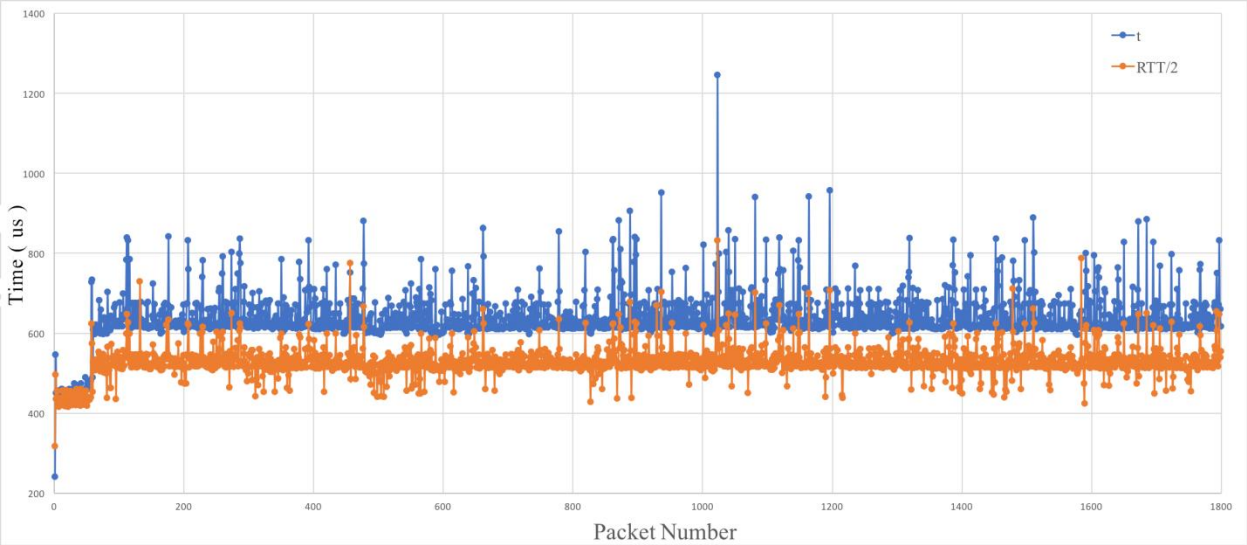

Figure 21. Forward delay (t) vs RTTs/2 on Ethernet.

Figures 20 (and 21) show t and RTT/2 for CAN bus and Ethernet. As explained above, the estimated error of the average forward delay differs from RTT/2 by only ~4.9% on CAN Bus (Figure 19), while on the Ethernet (Figure 20) it is ~20.8%. Thus, on Ethernet, the scale of the error is approximately 4 times higher due to higher dispersion of the values. Notice for example, that the standard deviation of the forward delay on Ethernet is 54.3us was (resp. 19.2us on CAN bus), while the corresponding average forward delay is 640.8us on Ethernet (resp. 481.7us on CAN bus). Therefore, we have an improved bound on the forward (and similarly reverse) delay on the CAN bus. This accuracy loss when using RTT/2 as an approximation can be critical, e.g. for Linux systems with real-time communication requirements.

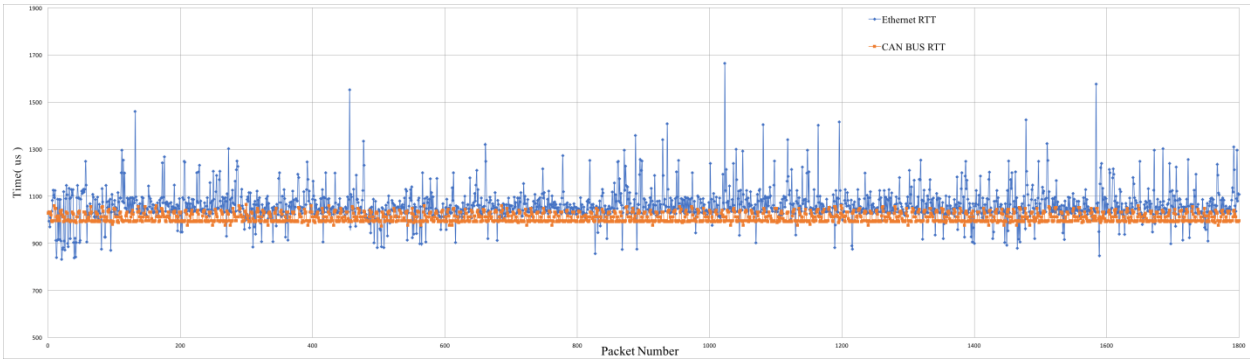

Figure 22. RTT on CAN Bus and Ethernet.

Figure 22 shows the computed round-trip time on the CAN bus and Ethernet. Notice that the round-trip time on CAN bus is more stable than Ethernet. Higher stability in round-trip times mean improved accuracy in one-way delay estimation, due to network calculus relationship explained in Section 3.2.1.

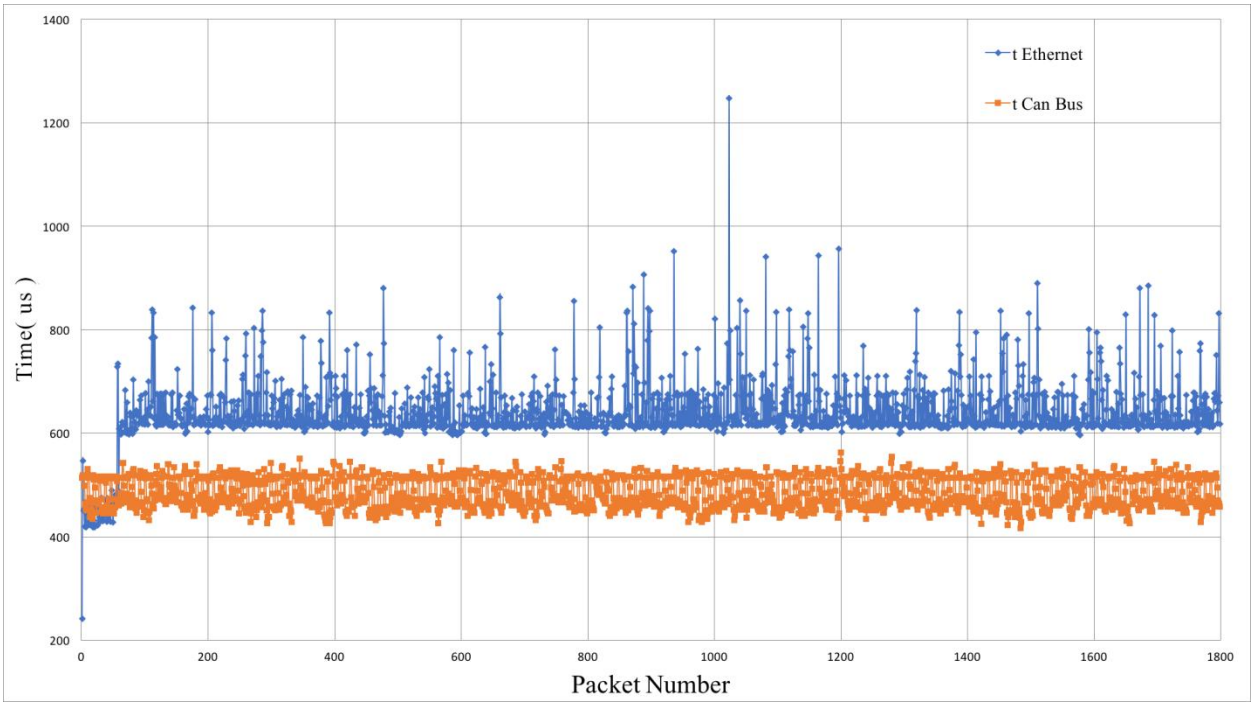

Figure 23. Forward delay on CAN Bus and Ethernet.

This is also confirmed in Figure 23, which shows the forward delay on Ethernet and CAN Bus. As we can see forward delay on Ethernet has higher fluctuations than on CAN bus. Notice that forward delay on CAN Bus is 159us smaller (or 33% smaller) than Ethernet.

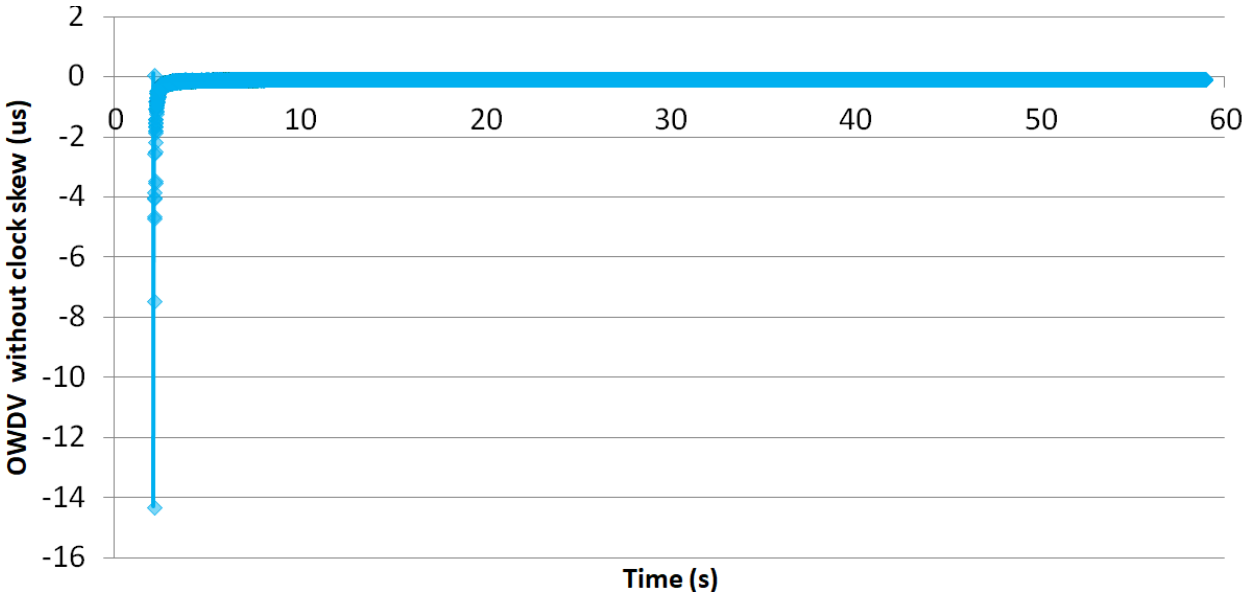

Figure 24. OWDV without skew on CAN Bus.

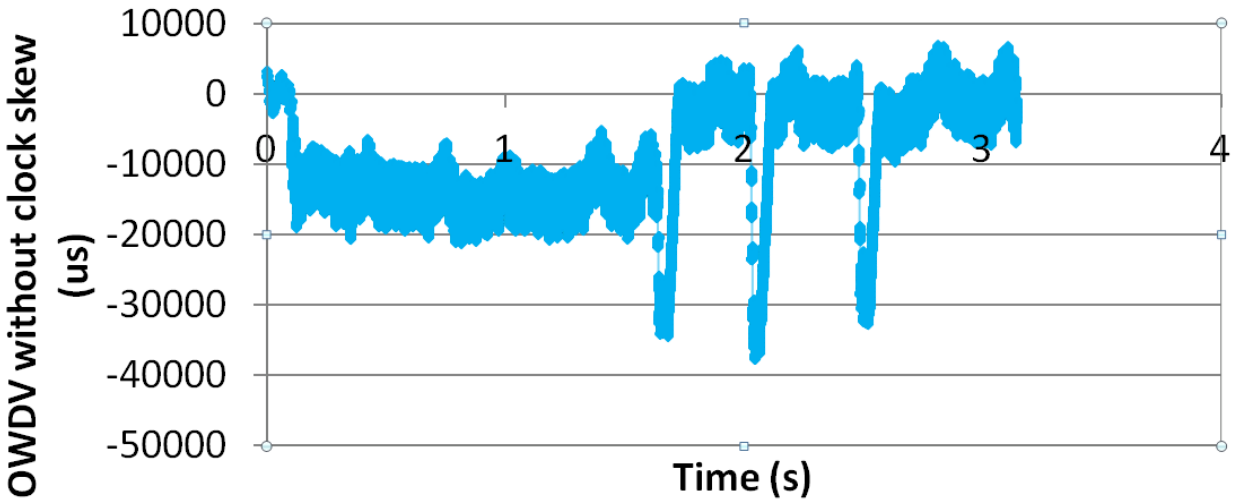

Figure 25. OWDV without skew on Ethernet.

In Figures 24 and 25 we show the one-way delay variation, after the clock skew has been removed, on CAN bus. As we can see in Figure 24, variation of one-way delay on can bus is rather stable without sudden increases. In fact, after the 10th second, one-way delay variation never exceeds the range of -2,2us. However, one-way delay variation on Ethernet is up to four orders of magnitude higher than CAN bus, i.e. a range from -40000us to 10000us.

5. Conclusions and Future work

In this thesis, we implemented two one-way delay estimation algorithms. The first one was measuring forward delay and reverse delay by considering round-trip times at the sender and receiver node. The second one measured on one-way delay variation from inter packet delays, applying also linear regression to remove any clock skew.

When it comes to comparing, the former technique by Choi and Yoo which focuses on one-way delay is more intrusive, since it causes a larger overhead to the network, since packets are sent and received on both sides (sender and receiver). It is also very sensitive to the initial constant t_0. On the other hand, another technique by Aoki, Oki and Cessa does not cause much overhead on the network. However, it uses more complex computations that are influencing the processing time to perform least squares calclulations. In our implementations, special attention was made that both of the techniques a) use code symmetry (in computation time) during the phase of sending and receiving packets, and b) avoid doing complex and especially redundant mathematics computations.

Some extensions to the algorithm by Choi and Yoo can be considered next:

- improving the accuracy of one-way delay estimation in large-scale systems.
- improving the accuracy by making this process self-adjusting. Instead of manually setting the number of packets and the correction values such as available bandwidth, the algorithm can set them by itself to provide a necessary level of accuracy.
- An interesting extension relates to computing end-to-end one-way delay when multiple nodes are in between, when each of the intermediate nodes can individually measure end-to-end delay with its neighbor(s) without much network overhead and computing power.
- Applying encryption to the packets for security reasons. This should be carefully implemented as too much processing time could result in asymmetries.
- Implementing this technique at kernel level. At kernel level, we must trace the socket buffers to estimate better RTTr and RTTs, and therefore obtain more accurate timestamps.
- On the CAN bus implementation forward and reverse delay can be computed on-the-fly, i.e. in parallel with round-trip times.

Some extensions related to Aoki, Oki and Cessa's technique relate to:

- improving the accuracy of one-way delay variation estimation,
- extending this technique to measure one-way delay and one-way delay variation on both sides, and
- applying encryption to the packets.

Both our implementations will be released as open source either separately or as part of other tools focusing on delay metrics, such as pathload.

One-way delay is also related to distance. The larger the distance in network hops of end-to-end nodes, usually the larger the one-way delay. It would be interesting, if one can accurately compute the location of an object at a specific time using one-way delay computations based on laser beams. Some algorithms provide methodologies on how you can localize objects in space by measuring Cartesian coordinates [12], [13], [14].

References:

1. C. Lenzen, T. Locher, P. Sommer, and R. Wattenhofer, “Theory and Practice of Computer Science,” vol. 4910, SOFSEM Conference, 2008.
2. R. Essen, “Greenwich Time - from Pendulum to Quartz. Greenwich Time: From Pendulum to Quartz 1”, Horological Journal, British Horological Institute, **Vol. 154**, 2012
3. L. Lamport “Time, Clocks, and the Ordering of Events in a Distributed System”, Communications of the ACM, **21 (7),** 1978, pp. 558-565.
4. A. S. Tanenbaum “*Distributed Operating Systems*”, Prentice Hall, 1995.
5. J.-H. Choi and C. Yoo, “One-way delay estimation and its application,” *Comput. Commun.*, **vol. 28**, 2005, pp. 819–828.
6. H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz, “The effects of assymetry on TCP performance,” *Mobile Networks and Applications*, **vol. 4**, 1999, pp. 219–241.
7. Cosmos Magazine, “https://cosmosmagazine.com/technology/how-atomic-clocks-keep-perfect-time-for-billions-of-years”.
8. D.L. Mills. Internet time synchronization: The network time protocol. IEEE Transactions on Communications, **39(10)**, 1991, pp.1482–1493.
9. R. Baldoni, A. Corsaro, L. Querzoni, S. Scipioni, and S. T. Piergiovanni, “Coupling-based internal clock synchronization forlLarge-scale dynamic distributed systems,” *IEEE Trans Parallel Distr. Systems*, vol. 21, no. 5, 2010, pp. 607–619.
10. F. Cristian. “Probabilistic clock synchronization”. Tech report RJ 6432(62550), IBM Almaden Research Center, Sept 1988.
11. M. Aoki, E. Oki, and R. Rojas-Cessa, “Measurement scheme for one-way delay variation with detection and removal of clock skew,” *ETRI J.*, **vol. 32, no. 6**, 2010, pp. 854–862.
12. Y. Zhou, “An efficient least-squares trilateration algorithm for mobile robot localization”, *IEEE/RSJ Int. Conf. Intell. Robot. Syst. IROS 2009*, 2009, pp. 3474–3479.
13. F. Thomas and L. Ros, “Revisiting trilateration for robot localization,” *IEEE Trans. Robot.*, **vol. 21**, no. 1, 2005, pp. 93–101.
14. S. Safavi, and U.A. Khan, "An opportunistic Linear–Convex algorithm for localization in mobile Robot networks", *IEEE Trans. on Robotics*, **33-4**, August 2017, pp. 875-888.

Appendix A1. One-way Delay: TCP

Client:

```
1.  /*
2.
3.
4.                                  time_t          tv_sec      seconds 1 second = 10^3 ms
5.                                  suseconds_t     tv_usec     microseconds 1 microsecond = 10^
    -6 seconds
6.  */
7.
8.
9.  #include<stdio.h>
10. #include<string.h>
11. #include<sys/socket.h>
12. #include<arpa/inet.h>
13. #include <sys/time.h>
14. #include<unistd.h>
15. #include <malloc.h>
16. #include <stdbool.h>
17. #include <netinet/in.h>
18. #include <netinet/tcp.h>
19.
20.
21. char *getPacketFromBuffer(char *, bool*);
22. int processRequest(struct networkPacket, int, struct timeval );
23. struct timeval getRTTDiff(struct timeval, struct timeval);
24.
25. #define numberOfPackets 18000
26. #define correctionValue 10
27.
28. struct networkPacket
29. {
30.     char packetType[200];
31.     char packetInfo[500];
32.     uint16_t packetId;
33.     uint32_t time;
34.
35. };
36.
37.
38. int main(int argc , char *argv[])
39. {
40.
41.     // SET PRIORITY
42.     nice(-20);
43.
44.     int sock;
45.     struct sockaddr_in server;
46.
47.     int read_size;
48.     // RTT_packet gets a timestamp on each receive. This is done to avoid any delay tha
    t the code can add.
49.     struct timeval RTT_packet,RTT_start,RTT_end,RTT_client;
50.     // recvPacketServer is the packet that client receives from the server
51.     // replyPacket is the packet that the client sents back to the client.
```

```
52.     struct networkPacket recvPacketServer, replyPacket;
53.
54.     //Create socket
55.
56.     sock = socket(AF_INET , SOCK_STREAM , 0);
57.     //Set socket options
58.     int i = 1;
59.     setsockopt( sock, IPPROTO_TCP, TCP_NODELAY, (void *)&i, sizeof(i));
60.
61.     int corki = 0;
62.     setsockopt( sock, IPPROTO_TCP, TCP_CORK, (void *)&corki, sizeof(corki));
63.
64.     int quickacki = 1;
65.     setsockopt( sock, IPPROTO_TCP, TCP_QUICKACK, (void *)&quickacki, sizeof(quickacki))
   ;
66.
67.
68.
69.     if (sock == -1){
70.         printf("Socket Failed!");
71.     }
72.     puts("Socket Success!");
73.        //Set The server IP [127.0.0.1]
74.        //AND PORT [7777]
75.     server.sin_addr.s_addr = inet_addr("127.0.0.1");
76.     server.sin_family = AF_INET;
77.     server.sin_port = htons( 7777 );
78.
79.     //Connect to remote server
80.     if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0){
81.         puts("\nConnect Failed!\n");
82.         return 1;
83.     }
84.
85.     puts("\nConnect Success\n");
86.
87.     // Start RTT Procedure.
88.     struct timeval recvPacketTimestamp;
89.     struct networkPacket sendToServerPacket, receivedFromServerPacket;
90.     sprintf(sendToServerPacket.packetType, "RTT_REQ");
91.     sprintf(sendToServerPacket.packetInfo, "%d,%d", numberOfPackets, correctionValue);
92.
93.     if( send(sock , &sendToServerPacket , sizeof(struct networkPacket) , 0) < 0){
94.                 puts("Send failed");
95.                 return 1;
96.     }
97.        // buffer reading, MSG_WAITALL waits for a sizeof(struct networkPacket) bits
98.     while( (read_size = recv(sock , &receivedFromServerPacket, sizeof(struct networkPac
   ket) , MSG_WAITALL) ) > 0 ){
99.         gettimeofday(&recvPacketTimestamp,NULL);
100.                processRequest(receivedFromServerPacket, sock, recvPacketTimestamp);
101.
102.           }
103.
104.
105.           return 0;
106.       }
107.
108.
109.       // gets two structures and finds the diffrence
```

```
110.        struct timeval getRTTDiff(struct timeval t1, struct timeval t2){
111.            int diffSec;
112.            long int diffUsec;
113.            struct timeval finalStruct;
114.            if(t2.tv_usec-t1.tv_usec<0){
115.              diffSec=t2.tv_sec - t1.tv_sec - 1;
116.              diffUsec = t2.tv_usec - t1.tv_usec + 1000000;
117.            }
118.            else{
119.              diffSec=t2.tv_sec - t1.tv_sec;
120.              diffUsec = t2.tv_usec - t1.tv_usec;
121.            }
122.            finalStruct.tv_sec=diffSec;
123.            finalStruct.tv_usec=diffUsec;
124.            return finalStruct;
125.        }
126.
127.
128.        int processRequest(struct networkPacket receivedFromServerPacket,int sock, struc
    t timeval recvPacketTimestamp){
129.          struct timeval RTT_start, RTT_end, RTT_client;
130.          struct networkPacket sendPacketToServer;
131.          int read_size;
132.          if(strcmp(receivedFromServerPacket.packetType,"RTT")==0){
133.            if(receivedFromServerPacket.packetId==0){
134.              RTT_start = recvPacketTimestamp;
135.            }else{
136.              RTT_end=recvPacketTimestamp;
137.              RTT_client= getRTTDiff(RTT_start,RTT_end);
138.              sendPacketToServer.time = (RTT_client.tv_sec * 1000000) + RTT_client.tv_us
    ec;
139.              sendPacketToServer.packetId=receivedFromServerPacket.packetId;
140.              RTT_start = RTT_end;
141.            }
142.
143.          send(sock, &sendPacketToServer, sizeof(struct networkPacket), 0);
144.
145.
146.
147.
148.          }
149.
150.
151.        }
```

Server:

```c
/*
 */

#include<stdio.h>
#include<string.h>    //strlen
#include<sys/socket.h>
#include<arpa/inet.h> //inet_addr
#include<unistd.h>    //write
#include <sys/time.h>
#include <stdlib.h>
#include <stdbool.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
struct networkPacket
{
    char packetType[200];
    char packetInfo[500];
    uint16_t packetId;
    uint32_t time;

};



int processRequest(struct networkPacket, int);


int main(int argc , char *argv[])
{
    // SET PRIORITY
    nice(-20);

    int socket_desc , client_sock , c , read_size;
    struct sockaddr_in server , client;


    //Create socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1)
    {
        puts("Socket Failed");
    }
    puts("\nSocket Success!\n");

    //Prepare the sockaddr_in structure
    //Server Listens on 7777
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( 7777 );

    // Port Reusability.
    int reusePort = 1;
    setsockopt(socket_desc, SOL_SOCKET, SO_REUSEPORT ,  &reusePort, sizeof(reusePort));


    int reuseAddr = 1;
    setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR ,  &reuseAddr, sizeof(reuseAddr));

    // Disable Nagles Algorithm.
    int i = 1;
    setsockopt( socket_desc, IPPROTO_TCP, TCP_NODELAY, (void *)&i, sizeof(i));
```

```
61.
62.     int corki = 0;
63.     setsockopt( socket_desc, IPPROTO_TCP, TCP_CORK, (void *)&corki, sizeof(corki));
64.
65.     int quickacki = 1;
66.     setsockopt( socket_desc, IPPROTO_TCP, TCP_QUICKACK, (void *)&quickacki, sizeof(quic
    kacki));
67.
68.
69.
70.     //Bind The Socket
71.     if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
72.     {
73.
74.         puts("\nBind Failed!\n");
75.         return 1;
76.     }
77.     puts("\nBind Success!\n");
78.
79.     //Listen
80.     listen(socket_desc , 10); // TODO IMPROVE THE SERVER
81.
82.     puts("Waiting for incoming connections...");
83.     c = sizeof(struct sockaddr_in);
84.
85.     //accept connection from an incoming client
86.     client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c);
87.     if (client_sock < 0) {
88.         puts("\nConnection with client failed\n");
89.         return 1;
90.     }
91.     puts("\nConnection accepted\n");
92.     struct networkPacket receivedPacket;
93.     while( (read_size = recv(client_sock , &receivedPacket, sizeof(struct networkPacket
    ) , MSG_WAITALL) ) > 0 ){ // buffer reading.
94.
95.         processRequest(receivedPacket,client_sock);
96.
97.     }
98.
99.     return 0;
100.       }
101.
102.      // gets two timeval structures and find's the diffrence
103.      struct timeval getRTTDiff(struct timeval t1, struct timeval t2){
104.          int diffSec;
105.          long int diffUsec;
106.          struct timeval finalStruct;
107.          if(t2.tv_usec-t1.tv_usec<0){
108.            diffSec=t2.tv_sec - t1.tv_sec - 1;
109.            diffUsec = t2.tv_usec - t1.tv_usec + 1000000;
110.          }
111.          else{
112.            diffSec=t2.tv_sec - t1.tv_sec;
113.            diffUsec = t2.tv_usec - t1.tv_usec;
114.          }
115.          finalStruct.tv_sec=diffSec;
116.          finalStruct.tv_usec=diffUsec;
117.          return finalStruct;
118.      }
119.
```

```
120.
121.        int processRequest(struct networkPacket receivedPacket,int client_sock){
122.
123.            int read_size;
124.            struct timeval packet_RTTr, start_RTTr,end_RTTr,RTT_Diff;
125.            struct networkPacket serverPacket, sendToClientPacket,recvFromClientPacket;

126.            int numberOfPackets;
127.            int correctionAt;
128.            int correctedPacketId; // i/correctionAt
129.            long int t,k;
130.            long int RTT_r;
131.            long int RTT_s;
132.            bool firstPhase = true;
133.            // For the equations  //
134.            long int RTTs_sum, RTTr_sum;
135.            RTTs_sum=0;
136.            RTTr_sum=0;
137.            if(strcmp(receivedPacket.packetType,"RTT_REQ")==0){
138.
139.                sscanf(receivedPacket.packetInfo, "%d,%d", &numberOfPackets, &correction
    At);
140.                //MEASURING RTT'S
141.                for(int i=0;i<numberOfPackets;i++){
142.                    // Prepare the packet
143.                    sprintf(sendToClientPacket.packetType, "RTT");
144.                    sendToClientPacket.packetId=i;
145.                    sendToClientPacket.time = 0;
146.                    if( send(client_sock , &sendToClientPacket , sizeof(struct networkPa
    cket) , 0) < 0 ){
147.                        puts("Send failed");
148.                        return 1;
149.                    }
150.
151.                    read_size = recv(client_sock , &recvFromClientPacket , sizeof(struct
     networkPacket) , MSG_WAITALL);
152.                    gettimeofday(&packet_RTTr,NULL);
153.                    if(i==0){
154.                        start_RTTr = packet_RTTr;
155.
156.                    }else{
157.
158.                        // Measuring time on each ack, packet_RTTr is a timestamp for ea
    ch received packet.
159.                        end_RTTr = packet_RTTr;
160.                        RTT_Diff = getRTTDiff(start_RTTr,end_RTTr);
161.                        RTTs_sum+=(RTT_Diff.tv_sec*1000000)+RTT_Diff.tv_usec;
162.                        RTTr_sum+=recvFromClientPacket.time;
163.                        start_RTTr = end_RTTr;
164.                        if(i%correctionAt==0){
165.
166.                            RTT_r = RTTr_sum/( (float) correctionAt );
167.                            RTT_s = RTTs_sum/( (float) correctionAt );
168.                            printf("RTT_R is : %ld RTT_S is : %ld\n",RTT_r,RTT_s);
169.                            // initialize K
170.                            if( firstPhase ){
171.                                k = ( (RTT_Diff.tv_sec*1000000)+RTT_Diff.tv_usec )/2.0;

172.                                t = k;
173.                                firstPhase = false;
174.                            }else{
```

```
                                t = RTT_r - k;
                                k = RTT_s - t;
                                printf(" t + k = RTT_s | %ld + %ld = %ld|\n",t,k,RTT_s);

                            }
                            RTTs_sum=0;
                            RTTr_sum=0;
                        }

                    //puts(recvPacket); // We can now manage the packet
                }
            }


            }
            return 1;
        }
```

Appendix A2. One-way Delay: CAN Bus

Sender:

```
1.  void VatiCAN::owd_sender(){
2.    int i = 0 ;
3.    CANSENDER ID_RECEIVED;
4.    uint64_t received_rtt;
5.    uint32_t phasesumt=0, phasesumk=0;
6.    int correctionAt = 300;
7.
8.    // OWD WITH PHASES AS DESCRIBED IN SECTION 4
9.    //spliting OWD_NUMBER_OF_PACKETS into phases of 10's
10.   int outSideLoop = OWD_NUMBER_OF_PACKETS/10;
11.   for(int j=0; j<outSideLoop; j++){
12.     for(int k=0; k<10; k++){
13.       _can.sendMsgBuf(OWD_SENDER_ID, 0, sizeof(payload), payload);
14.       waitForMessage(recv_payload, OWD_RECEIVER_ID);
15.       PREV_RECV_ACK = RECV_ACK;
16.       RECV_ACK = micros();
17.       RTT_MSG_RAW =  RECV_ACK - PREV_RECV_ACK;
18.       convert_uint8arr_to_uint64(&received_rtt,recv_payload);
19.       rtt_receiver_arr[k] = received_rtt;
20.       rtt_sender_arr[k] = RTT_MSG_RAW;
21.     }
22.   tsum = 0;
23.   ksum = 0;
24.   k = rtt_sender_arr[1]/2.0;
25.   for(int index=2; index<10; index++){
26.     t = rtt_sender_arr[index] - k;
27.     k = rtt_receiver_arr[index] - t;
28.     tsum+= t;
29.     ksum+= k;
30.   }
31.   //The estimated forward delay and reverse for each window
32.   phasesumt+= (tsum / 8);
33.   phasesumk+= (ksum / 8);
34.   }
35.   averagek = phasesumk/outSideLoop;
36.   averaget = phasesumt/outSideLoop;
37.
38.   /* OWD WITH PHASES AS DESCRIBED IN SECTION 3.2.2
39.   rtt_r_sum = 0;
40.   rtt_s_sum = 0;
41.
42.   firstPhase = true;
43.   int index2;
44.   for(int i=0;i<OWD_NUMBER_OF_PACKETS;i++){
45.     _can.sendMsgBuf(OWD_SENDER_ID, 0, sizeof(payload), payload);
46.     waitForMessage(recv_payload, OWD_RECEIVER_ID);
47.     convert_uint8arr_to_uint64(&rtt_r,recv_payload);
48.     PREV_RECV_ACK = RECV_ACK;
49.     RECV_ACK = micros();
50.     RTT_MSG_RAW = RECV_ACK - PREV_RECV_ACK;
51.     if(i>0){
52.         rtt_r_sum += rtt_r;
53.         rtt_s_sum += RTT_MSG_RAW;
54.     }
55.
```

```
    if(i % correctionAt == 0 && i >= correctionAt){
      rtt_r = rtt_r_sum / correctionAt;
      rtt_s = rtt_s_sum / correctionAt;

      if(firstPhase){
        k = (rtt_r+rtt_s)/4.0;
        t = k;
        firstPhase = false;
      }else{

        // forward delay t and reverse delay k
        t = rtt_r - k;
        k = rtt_s - t;

        }
      rtt_r_sum = 0 ;
      rtt_s_sum = 0 ;

    }
    */

  }
  Serial.println("Done");


}
```

Receiver:

```
void VatiCAN::owd_receiver(){
  int i;
  CANSENDER ID_RECEIVED;
  for(i = 0; i<OWD_NUMBER_OF_PACKETS; i++){
    waitForMessage(payload, OWD_SENDER_ID);
    PREV_RECV_ACK = RECV_ACK;
    RECV_ACK = micros();
    RTT_MSG_RAW =  RECV_ACK - PREV_RECV_ACK;
    convert_uint64_to_uint8arr(RTT_MSG_RAW,payload);
    _can.sendMsgBuf(ID, 0, sizeof(payload), payload);

  }


}
```

Appendix A3. One-way Delay Variation: TCP

Sender:

```
/*


                            time_t          tv_sec      seconds 1 second = 10^3 ms
                            suseconds_t     tv_usec     microseconds 1 microsecond = 10^
-6 seconds
*/


#include<stdio.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include <sys/time.h>
#include<unistd.h>
#include <stdbool.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#define numberOfPackets 500
#define windowSize 10

uint32_t timevalToMicros(struct timeval);
struct timeval getTimevalDiff(struct timeval , struct timeval );

struct networkPacket
{
    char packetType[200];
    char packetInfo[500];
    uint16_t packetId;
    uint32_t time;

};


int main(int argc , char *argv[])
{

    struct networkPacket sendPacketToServer;
    struct timeval sendPacketTime,prevSendPacketTime;
    //send(sock, &sendPacketToServer, sizeof(struct networkPacket), 0);


    // SET PRIORITY
    nice(-20);

    int sock;
    struct sockaddr_in server;

    int read_size;
    // RTT_packet gets a timestamp on each receive. This is done to avoid any delay tha
t the code can add.
```

```
51.     struct timeval RTT_packet,RTT_start,RTT_end,RTT_client;
52.     // recvPacketServer is the packet that client receives from the server
53.     // replyPacket is the packet that the client sents back to the client.
54.     struct networkPacket recvPacketServer, replyPacket;
55.
56.     //Create socket
57.
58.     sock = socket(AF_INET , SOCK_STREAM , 0);
59.
60.
61.
62.     if (sock == -1){
63.         printf("Socket Failed!");
64.     }
65.     puts("Socket Success!");
66.
67.     server.sin_addr.s_addr = inet_addr("127.0.0.1");
68.     server.sin_family = AF_INET;
69.     server.sin_port = htons( 7777 );
70.     int i = 1;
71.     setsockopt( sock, IPPROTO_TCP, TCP_NODELAY, (void *)&i, sizeof(i));
72.
73.     int corki = 0;
74.     setsockopt( sock, IPPROTO_TCP, TCP_CORK, (void *)&corki, sizeof(corki));
75.
76.     int quickacki = 1;
77.     setsockopt( sock, IPPROTO_TCP, TCP_QUICKACK, (void *)&quickacki, sizeof(quickacki))
    ;
78.
79.
80.
81.     //Connect to remote server
82.     if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0){
83.         puts("\nConnect Failed!\n");
84.         return 1;
85.     }
86.
87.     puts("\nConnect Success\n");
88.
89.     /* Sending windows Size to server */
90.     sprintf(sendPacketToServer.packetType, "START_MEASURING");
91.     sprintf(sendPacketToServer.packetInfo, "%d,%d", numberOfPackets,windowSize);
92.     send(sock, &sendPacketToServer, sizeof(struct networkPacket), 0);
93.
94.     sprintf(sendPacketToServer.packetType, "TR");
95.     for(int i = 0 ; i<numberOfPackets ; i++){
96.       sendPacketToServer.packetId=i;
97.       sendPacketToServer.time = timevalToMicros(getTimevalDiff(prevSendPacketTime, send
    PacketTime));
98.       send(sock, &sendPacketToServer, sizeof(struct networkPacket), 0);
99.       prevSendPacketTime = sendPacketTime;
100.             gettimeofday(&sendPacketTime, NULL);
101.             #if defined DEBUG
102.                 printf("sendPacketTime.id :%d |", sendPacketToServer.packetId);
103.                 printf("sendPacketTime.packetType is :%s |", sendPacketToServer.packet
    Type);
104.                 printf("sendPacketTime.time is :%d |", sendPacketToServer.time);
105.                 printf("\n");
106.             #endif
107.
108.
```

```
            }

            return 0;
        }


        struct timeval getTimevalDiff(struct timeval t1, struct timeval t2){
            int diffSec;
            long int diffUsec;
            struct timeval finalStruct;
            if(t2.tv_usec-t1.tv_usec<0){
              diffSec=t2.tv_sec - t1.tv_sec - 1;
              diffUsec = t2.tv_usec - t1.tv_usec + 1000000;
            }
            else{
              diffSec=t2.tv_sec - t1.tv_sec;
              diffUsec = t2.tv_usec - t1.tv_usec;
            }
            finalStruct.tv_sec=diffSec;
            finalStruct.tv_usec=diffUsec;
            return finalStruct;
        }

        uint32_t timevalToMicros(struct timeval t1){
            uint32_t millis;
            millis= (t1.tv_sec * 1000000) + t1.tv_usec;
            return millis;

        }
```

Receiver:

```
#include<stdio.h>
#include<string.h>    //strlen
#include<sys/socket.h>
#include<arpa/inet.h> //inet_addr
#include<unistd.h>    //write
#include <sys/time.h>
#include <stdlib.h>
#include <stdbool.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <sys/types.h>          /* See NOTES */

struct networkPacket
{
    char packetType[200];
    char packetInfo[500];
    uint16_t packetId;
    uint32_t time;

};


uint32_t linear_reg(int , int ,int *,  int *, int *, int *,int *, float *, float *);

void processRequest(struct networkPacket, int,struct timeval );
```

```
uint32_t timevalToMicros(struct timeval);
struct timeval getTimevalDiff(struct timeval , struct timeval );
/* PUBLIC VARIABLES */

struct timeval prevRecvPacketTimestamp;
uint32_t sumIPD;
uint32_t sumTS;
uint32_t dmin;
uint32_t firstOWD;


int windowSize;
int numberOfPackets;
uint32_t min_sum_in_window=100000;
int *allOwds;
int *minD;


/* No need for = 0 it is already assigned.*/

int sumx;
int sumy;
float xmean;
float ymean;
int sum_xxmea0;
int sum_yymea0;
int sum_xmeanymean;
int sumsrqrtxmxmean;

int sumepisum;
float a;
float b;



/* END OF PUBLIC VARIABLES */

int main(int argc , char *argv[])
{
    int newy;
    uint32_t *timestamp_at_receive;
    int *OWDVEstimation;
    #if defined DEBUG
        printf("%s\n", "DEBUG MODE ON");
    #endif

    struct timeval recvPacketTimestamp;
    sumIPD=0;
    sumTS=0;
    dmin=99999;
    // SET PRIORITY
    nice(-20);

    int socket_desc , client_sock , c , read_size;
    struct sockaddr_in server , client;
    //Create socket
    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
    if (socket_desc == -1)
    {
        puts("Socket Failed");
```

```
88.     }
89.     puts("\nSocket Success!\n");
90.
91.     //Prepare the sockaddr_in structure
92.     server.sin_family = AF_INET;
93.     server.sin_addr.s_addr = INADDR_ANY;
94.     server.sin_port = htons( 7777 );
95.
96.
97.
98.     int i = 1;
99.     setsockopt( socket_desc, IPPROTO_TCP, TCP_NODELAY, (void *)&i, sizeof(i));
100.            int corki = 0;
101.            setsockopt( socket_desc, IPPROTO_TCP, TCP_CORK, (void *)&corki, sizeof(corki
    ));
102.            int quickacki = 1;
103.            setsockopt( socket_desc, IPPROTO_TCP, TCP_QUICKACK, (void *)&quickacki, size
    of(quickacki));
104.
105.
106.
107.
108.            //Bind The Socket
109.            if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
110.            {
111.
112.                puts("\nBind Failed!\n");
113.                return 1;
114.            }
115.            puts("\nBind Success!\n");
116.
117.            //Listen
118.            listen(socket_desc , 10); // 10 Connections are accepted (We only need 1)
119.
120.            puts("Waiting for incoming connections...");
121.            c = sizeof(struct sockaddr_in);
122.
123.            //accept connection from an incoming client
124.            client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c
    );
125.            if (client_sock < 0) {
126.                puts("\nConnection with client failed\n");
127.                return 1;
128.            }
129.            puts("\nConnection accepted\n");
130.
131.            struct networkPacket receivedPacket;
132.            while( (read_size = recv(client_sock , &receivedPacket, sizeof(struct networ
    kPacket) , MSG_WAITALL) ) > 0 ){ // buffer reading.
133.                prevRecvPacketTimestamp = recvPacketTimestamp;
134.                gettimeofday(&recvPacketTimestamp,NULL);
135.                timestamp_at_receive[receivedPacket.packetId] = timevalToMicros(recvPack
    etTimestamp);
136.                processRequest(receivedPacket,client_sock,recvPacketTimestamp);
137.
138.            }
139.         //  for(int i = 0 ; i <numberOfPackets; i++){
140.         //      newy = a * timestamp_at_receive[i] + b;
141.         //      OWDVEstimation[i] = y - newy;
142.
143.         //}
```

```
144.            //IN CASE WE COUNT RTT_S
145.            //Free Pointers.
146.
147.            free(timestamp_at_receive)
148.            free(OWDVEstimation);
149.            free(allOwds);
150.            return 0;
151.        }
152.
153.        void processRequest(struct networkPacket receivedPacket, int client_sock, struct
     timeval recvPacketTimestamp){
154.            uint32_t diffSum;
155.            uint32_t ipd;
156.            #if defined DEBUG
157.               printf("Im in processRequest|");
158.               printf("packetType is : %s\n", receivedPacket.packetType);
159.            #endif
160.
161.            if(strcmp(receivedPacket.packetType,"START_MEASURING")==0){
162.                sscanf(receivedPacket.packetInfo, "%d,%d", &numberOfPackets,&windowSize)
    ;
163.                allOwds = (int *) malloc(numberOfPackets * (sizeof(int)));
164.                timestamp_at_receive = (uint32_t * ) malloc(numberOfPackets * sizeof(uin
    t32_t));
165.                OWDVEstimation = (int *) malloc(numberOfPackets *(sizeof(int)));
166.            }
167.            if(strcmp(receivedPacket.packetType,"TR")==0){
168.                if(receivedPacket.packetId>1){
169.                    // measuring ipd
170.                    ipd=timevalToMicros(getTimevalDiff(prevRecvPacketTimestamp,recvPacke
    tTimestamp));
171.                    sumIPD+=ipd;
172.                    // measuring TS
173.                    sumTS+=receivedPacket.time;
174.                    diffSum = sumIPD - sumTS;
175.                    allOwds[receivedPacket.packetId] = diffSum;
176.                   //linear_reg(receivedPacket.packetId, diffSum, &sumx, &sumy,&sum_xmea
    nymean, &sumsrqrtxmxmean,&sumepisum, &a, &b);
177.                    #if defined DEBUG
178.                        // printf("ipd is :%d |", ipd);
179.                        // printf("min D is :%d |", dmin);
180.                        // printf("sumIPD is :%d |", sumIPD);
181.                        // printf("sumTS is :%d", sumTS);
182.                        // printf("AOWDV IS:%d", sumIPD - sumTS);
183.                        // printf("OWD IS:%d\n", sumIPD - sumTS + firstOWD);
184.                        // printf(" A is : %f\n", a);
185.                        // printf(" B is : %f\n", b);
186.                    #endif
187.                }
188.            }
189.        }
190.
191.        struct timeval getTimevalDiff(struct timeval t1, struct timeval t2){
192.            int diffSec;
193.            long int diffUsec;
194.            struct timeval finalStruct;
195.            if(t2.tv_usec-t1.tv_usec<0){
196.              diffSec=t2.tv_sec - t1.tv_sec - 1;
197.              diffUsec = t2.tv_usec - t1.tv_usec + 1000000;
198.            }
199.            else{
```

```
            diffSec=t2.tv_sec - t1.tv_sec;
            diffUsec = t2.tv_usec - t1.tv_usec;
          }
          finalStruct.tv_sec=diffSec;
          finalStruct.tv_usec=diffUsec;
          return finalStruct;
      }

      uint32_t timevalToMicros    (struct timeval t1){
          uint32_t millis;
          millis= (t1.tv_sec * 1000000) + t1.tv_usec;
          return millis;

      }

      uint32_t linear_reg(int x, int y,int *sumx,  int *sumy, int *sum_xmeanymean, int *sumsrqrtxmxmean,int *sumepisum , float *a, float *b){
          float xmean,ymean;

          int xmxmean;
          int ymymean;
          *sumy += y;
          *sumx += x;
          xmean = *sumx/x;
          ymean = *sumy/x;
          *sumsrqrtxmxmean+= ((x-xmean) * (x-xmean));
          *sum_xmeanymean+= (x-xmean) * (y-ymean);

          *a = (float) *sum_xmeanymean/(*sumsrqrtxmxmean);
          *b = (float) ymean - *a * xmean;
          // printf("a  is %f\n", *a);
          // printf("b  is %f\n", *b);
          // printf("sum xmeanymean  is %d\n", *sum_xmeanymean);
          // printf("sum sqrt  is %d\n", *sumsrqrtxmxmean);
          return 0;

      }
```

Appendix A4. One-way Delay Variation: CAN Bus

Sender:

```
1.  void VatiCAN::owdv_sender(){
2.    int i = 0 ;
3.    CANSENDER ID_RECEIVED;
4.    uint64_t received_rtt;
5.    uint32_t phasesumt=0, phasesumk=0;
6.    //phases = OWDV_NUMBER_OF_PACKETS / correctionWindow;
7.    for(int i=0; i<phases; i++){
8.      _can.sendMsgBuf(OWD_SENDER_ID, 0, sizeof(payload), payload);
9.        PREV_RECV_ACK = RECV_ACK;
10.       RECV_ACK = micros() ;
11.       RTT_MSG_RAW =  RECV_ACK - PREV_RECV_ACK ;
12.       convert_uint64_to_uint8arr(RTT_MSG_RAW,payload);
13.
14.   }
15. }
```

Receiver:

```
1.  void VatiCAN::owdv_receiver(CANSENDER ID){
2.    int i;
3.    CANSENDER ID_RECEIVED;
4.
5.    uint64_t TS;
6.    for(i = 0; i<OWD_NUMBER_OF_PACKETS; i++){
7.      waitForMessage(payload, OWD_SENDER_ID);
8.      PREV_RECV_ACK = RECV_ACK;
9.      RECV_ACK = micros();
10.     IPD =  RECV_ACK - PREV_RECV_ACK ;
11.     convert_uint8arr_to_uint64(&TS,payload);
12.     if(i>1){
13.       sumIPD+=IPD;
14.       sumTS+=TS;
15.       OWDV = sumIPD - sumTS;
16.       // Serial.print((int)IPD);
17.       // Serial.print("|");
18.       // Serial.print((int)TS);
19.       Serial.print((int)(OWDV));
20.       Serial.print("|");
21.       Serial.println(micros());
22.     }
23.
24.     // if(i>1 && TS!=0){
25.     //   linear_reg(i, (int) IPD-
    TS, &sumx, &sumy,&sum_xmeanymean, &sumsrqrtxmxmean,&sumepisum, &a, &b);
26.     // }
27.
28.
29.   }
30.   // Serial.print("Slope A:");
31.   // Serial.println(a);
32.   // Serial.print("B:");
33.   // Serial.println(b);
```

```
}
```

Linear Regression:

```
void VatiCAN::linear_reg(int x, int y,int *sumx,  int *sumy, int *sum_xmeanymean, int *sumsrqrtxmxmean,int *sumepisum , float *a, float *b){
    float xmean,ymean;
    *sumy += y;
    *sumx += x;
    xmean = *sumx/x;
    ymean = *sumy/x;
    *sumsrqrtxmxmean+= ((x-xmean) * (x-xmean));
    *sum_xmeanymean+= (x-xmean) * (y-ymean);
    *a = (float) *sum_xmeanymean/(*sumsrqrtxmxmean);
    *b = (float) ymean - *a * xmean;
    // printf("a  is %f\n", *a);
    // printf("b  is %f\n", *b);
    // printf("sum xmeanymean  is %d\n", *sum_xmeanymean);
    // printf("sum sqrt  is %d\n", *sumsrqrtxmxmean);
    return ;

}
```