NTNU
Norwegian University of
Science and Technology

Bluetooth Mesh and low-power data acquisition in real time

Bernt Johan Damslora

2018-12-18

Specialization Project Report
Engineering Cybernetics, Specialization Project (TTK4550)
Master of Science in Cybernetics and Robotics
Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor: Prof. Geir Mathisen

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology
and Electrical Engineering
Department of Engineering Cybernetics

Specialization Project Report

Candidate: **Bernt Johan Damslora**

Course: **TTK4550 Engineering Cybernetics, Specialization Project**

Thesis title (Norwegian) **Blåtann maskenettverk og lavenergi datainnsamling i sann tid**

Thesis title (English): **Bluetooth Mesh and low-power data acquisition in real time**

Work desctiption: In this report we want to investigate the properties of the Bluetooth protocols used for connectiong IoT (Internet of Things) units used for data acquisition.

Nordic Semiconductor's NRF52840 SoC (System on Chip) supports different configurations of Bluetooth network. We want to use this chip to explore the properties (energy consumption / efficiency, scalability, bandwidth, range of communication inside and outside buildings, communication robustness) of the different configurations.

In important issue both for IoT data acquisition and exploration of Bluetooth network properties is to have synchronized real-timers on the nodes. Thus, a priorized task will be to get a system for synchronization for timers on the nodes.

The application areas for the data acquisition will be power control in buildings, smart grid and communication within other critical infrastructures.

The tasks will be:

1. Conduct a literature search in the area of low-power wireless networks, especially Bluetooth-based networks.
2. Suggest and implement a system for synchronization of distributed timers
3. As far as time permits, configure different Bluetooth-based networks and explore the real-time properties of the networks.

Start date: August 18th, 2018
Due date: December 18th, 2018

Thesis performed at: Department of Engineering Cybernetics
Supervisor: Professor Geir Mathisen

Preface

I would like to thank my supervisor, Geir Mathisen, for his valuable guidance and feedback during this project.

The project was carried out in cooperation with Nordic Semiconductor. I would like to thank Nordic for providing access to state-of the-art competence in the field of low-power radio communication. In particular, I would like to thank Trond Einar Snekvik for providing an excellent in-depth explanation of Bluetooth Mesh, and David Edwin for providing literature and valuable insights on time synchronization.

Abstract

There is an increasing demand for low-power Internet-connected sensor systems. Technological advances have enabled new wireless communication technologies used to realize such systems. In this project, we survey some of these technologies, focusing on the recent developments of Bluetooth. In order to facilitate further experimentation with these technologies, we have developed a C implementation of a synchronization algorithm that can be used to estimate, with tight upper and lower bounds, the state of a timer on another connected device. We present simulated synchronization error measurements using this implementation with latencies inspired by those in a large Bluetooth Mesh network. The results suggest that millisecond-level synchronization can be achieved across such a network without tight integration with the radio hardware.

Contents

Preface . . . . . **ii**
Abstract . . . . . **iii**
Contents . . . . . **iv**
Acronyms . . . . . **vi**
List of Figures . . . . . **vii**
Listings . . . . . **viii**
1 Introduction **1**
1.1 Background and motivation 1
1.2 Limitations 2
1.3 Main contribution 2
1.4 Structure of this report 2
2 Literature Search **3**
2.1 Overview of different wireless technologies 3
2.1.1 3GPP cellular technologies 3
2.1.2 LoRa and LoRaWAN 3
2.1.3 2.4 GHz local networks 4
2.2 Details of Bluetooth Mesh (BT Mesh) 4
2.2.1 Working principles 4
2.2.2 Power efficiency 5
2.2.3 Latency and real-time properties 5
2.3 Acquiring a common time reference 5
2.4 Litterature search summary 6
3 The Tiny-sync Time Synchronization Scheme **7**
3.1 Background 7
3.2 Algorithm for constraint selection 8
4 An Implementation of the Tiny-Sync Constraint Selection Algorithm in C **10**
4.1 Data structures and state 10
4.2 Initializing the algorithm 11
4.3 Constraint selection 12
5 Testing the Implemented Algorithm **13**
5.1 Simulation framework 13
5.2 Simulation results 14
6 Discussion **18**
6.1 Precision of the estimate 18

6.2 Constant error given asymmetric delays . . . 18
6.3 Guaranteed upper and lower bounds . . . 18
6.4 Extending the synchronization from a pair to a network . . . 19
6.5 Response to non-linearity . . . 19
7 Conclusion . . . 20
7.1 Future work . . . 21
Sources . . . 22
A Tiny-sync Implementation Source Files . . . 23
A.1 Interface: estimator.h . . . 23
A.2 Implementation: estimator.c . . . 25
A.3 Test framework: main.c . . . 27

Acronyms

BLE Bluetooth Low Energy

BT Mesh Bluetooth Mesh

IoT Internet of Things

ISM Industrial Scientific Medical

LTE-M Long-Term Evolution Machine Type Communication

NB-IoT Narrowband IoT

TTL Time To Live

List of Figures

1 The possible values for a new upper constraint 9
2 Simulated synchronization errors of tiny-sync (run 1) 16
3 Simulated synchronization errors of tiny-sync with asymmetric delays (run 2) 16
4 Simulated synchronization errors of tiny-sync with nonlinear clock (run 3) 17

Listings

4.1 Definition of the basic data structures used in tiny-sync 10
4.2 Definition of the data structures held as state between iterations 11
4.3 Initialization of the algorithm, affecting the first two iterations 11
4.4 Constraint selection algorithm responsible for setting A_1 and B_2 12
5.1 Test configuration constants 13
5.2 Data structure for storing clock states 13
5.3 Function for generating new data points 14
5.4 Main program used to test the implementation 15
A.1 Tiny-sync implementation interface 23
A.2 Tiny-sync implementation 25
A.3 Tiny-sync implementation test framework 27

1 Introduction

1.1 Background and motivation

In our modern world, there are many use cases for Internet-connected distributed embedded systems (popularly called Internet of Things (IoT)). As an example, remotely monitored and controlled thermostats have been used by cottage owners to enable a warmer welcome for themselves for decades. More recently, networked power meters are being installed in homes across Norway and other countries to enable finer – not to mention automatic – monitoring of power use. There have also been experiments with using consumer water heaters for distributed energy storage, in order to smooth power consumption.

In the cases mentioned so far, there is usually an abundant power source available, for the purposes of computation and communication; if an embedded computer is switching a water heater of 2000 W, it does not matter much if the computer itself uses 5 mW or 500 mW. However, there are still reasons to minimize power consumption in IoT devices. There are use cases where an abundant source is not available. A remote solar powered cottage or rest area, for example, may have a severely limited power budget. In a battery-powered system, it is almost always desirable to further minimize power consumption, as it will either extend the usable time, or allow for a smaller battery.

With the above in mind, a number of technologies have been developed to address the need for low-power wireless communication. For the long range cellular communication required to keep a device connected to the Internet, some popular technologies are Narrowband IoT (NB-IoT), Long-Term Evolution Machine Type Communication (LTE-M) and LoRa. For the shorter range communication within a site, some popular technologies are Thread, Zigbee, Bluetooth Low Energy (BLE) and BT Mesh. One goal for this project is to acquire a bird's eye view of these technologies. They will be further distinguished, detailed, and discussed in Chapter 2. We will focus specifically on BT Mesh.

When using IoT devices to collect data, the time dimension is often important. One example where exact time-stamping is important is indoor location systems. Another would be monitoring the synchronization in an electric grid. This project has a goal of finding, implementing and testing some method of acquiring a common time reference across a BT Mesh sensor network.

1.2 Limitations

As a consequence of the difference between the envisioned scope of the project and available time, the algorithm implemented in Chapter 4 was tested in a simulated environment, rather than over a BT Mesh network on the Nordic nRF52840 System on Chip, as originally planned. As such, the testing done in this project can be used as a demonstration of the implementation's qualitative properties and an indicator of the possible performance, but not as a precise evaluation of the performance.

1.3 Main contribution

The contribution of this project is twofold. First, the literature search provides an overview of different low-power wireless technologies from an IoT perspective, focusing on the advantages and limitations of BT Mesh. This overview can be a useful resource when faced with the choice of which technology to use in a project.

Secondly, the implementation part provides a usable C implementation of the tiny-sync algorithm proposed by [1]. This implementation can be used to estimate – with tight upper and lower bounds – the state of a timer on another connected device.

1.4 Structure of this report

The rest of this report is structured as follows: It starts with the literature search in Chapter 2, providing background on the different wireless technologies discussed. The literature search continues with details on BT Mesh, and on time synchronization.

After the literature search, the rest of the report pertains to the tiny-sync implementation. Chapter 3 explains the details of the algorithm that are critical to understanding the implementation, and then goes on to specify the algorithm as pseudo-code. Chapter 4 covers the C implementation itself, and discusses implementation-specific properties like data types.

Chapter 5 presents the method used for testing the implementation, and presents the results from various test runs. Chapter 6 discusses these results, focusing on the different properties of the algorithm. Finally, Chapter 7 wraps up the report.

2 Literature Search

The literature search part of this project has several goals:

- Acquire a bird's eye view of current low-power wireless technologies from an IoT perspective
- Focusing on the properties of BT Mesh in terms of scalability, range, power efficiency and bandwidth
- Finding some method of acquiring a common time reference across a wireless sensor network

The rest of this chapter covers these areas in order, and ends with a short summary of the findings.

2.1 Overview of different wireless technologies

2.1.1 3GPP cellular technologies

NB-IoT and LTE-M (including the "enhanced" eMTC) are standards released by the 3rd Generation Partnership Project for communication with a terrestrial cellular network. [2] Both are designed to provide a low power service for IoT applications. The most notable differences are presented in Table 1. In short, NB-IoT is designed to provide better coverage and uses up less of the RF spectrum, but provides an order of magnitude slower maximum data rates.

NB-IoT and LTE-M are both cellular technologies operating in restricted radio bands. This confers the limitation that infrastructure is controlled by approved cellular operators, and it is not possible to run one's own network. On the other side, if the infrastructure is already in place, the only cost is the device itself, and the cost of cellular service. Since they operate in the LTE band, much of the existing infrastructure hardware is usable. At the time of writing, networks have been deployed in several European countries, the United States, Japan, China and others [3].

2.1.2 LoRa and LoRaWAN

LoRa is a modulation technique based on chirp spread spectrum radio technology. [4] LoRaWAN defines the upper layers designed to provide a low power, long range communication network based on that modulation. It operates in the open Industrial Scientific Medical (ISM) frequencies

	LTE-M	**NB-IoT**
Peak data rate (up/down)	1 Mbps	50 kbps
Bandwidth	1.08 MHz	180 KHz
Coverage	155.7 dB	164 dB

Table 1: Simplified summary of the differences between LTE-M and NB-IoT, from [2]. Coverage is specified in terms of Maximum Coupling Loss target (higher is better).

under 1 GHz, enabling private entities to set up their own networks without a license. Data rates and power consumption is advertised as even lower than NB-IoT. Semtech is the only company producing the radio hardware that use this modulation.

2.1.3 2.4 GHz local networks

There are multiple technologies for establishing low-power networks local to a site for IoT applications in the 2.4 GHz ISM band. The most commonly cited use case is a "smart home". Zigbee [5] and Thread [6] build on the IEEE 802.15.4 [7] radio standard, while BT Mesh builds on the Bluetooth (specifically BLE) standard. [8] All of these are mesh networks, meaning that nodes can communicate without a direct link, as long as there is a chain of relaying nodes between them.

Comparing Zigbee and Thread, several differences have been pointed to. [9] While Thread builds on the Internet Protocol version 6 protocol, and therefore integrates naturally with other IP networks, Zigbee has its own network layer and addressing system. Further, Zigbee has a wider scope, defining all layers of the system, including the application layer. This makes Zigbee applications somewhat less flexible in behaviour, but yields greater interoperability of similar products from different vendors.

BT Mesh is another mesh network which, like Zigbee, defines all layers up to the application, aiding interoperability. [8] The fact that it is built on top of BLE also means that it can be implemented on any hardware that supports BLE. It also means that smartphones or personal computers (where the operating systems typically include a Bluetooth protocol stack) can connect to the network.

Zigbee and Thread inherit their maximum data rate from the IEEE 802.11.4 standard, which is 250 kbps. The data rate for the physical layer of BLE is 1 Mbps. However, the practical data rates typically used and tested in the mesh networks are far lower. A series of tests performed by Silicon Labs achieves data rates on the order of single-digit kbps in all the mesh networks. [10]

2.2 Details of BT Mesh

2.2.1 Working principles

BLE is normally a connection-based protocol, where a link is established between two nodes before they communicate in an agreed-upon set of channels in the 2.4 GHz band. ([11], volume 6) BT Mesh, however, does not use the connection concept. Rather, there are only *messages* being passed from one node to the another through the network. To achieve this, the advertisement feature of BLE is used. [8] A node sending a message transmits it on the 3 dedicated advertisement channels. When not transmitting, the nodes listen on the advertisement bands (*scanning* state) as much of the time as possible. Transmitting the message on all 3 channels gives some resistance to disturbance; if one of the channels is jammed by another transmission, devices listening on either of the other 2 will receive the message.

To understand the function of the BT Mesh network, two fields in the message are critical: the destination address and the Time To Live (TTL). Each message contains a destination address, which can be unicast or multicast. The behaviour of the network layer of BT Mesh is as follows, in simplified terms (from [8], p. 47): If the packet has a TTL > 1 and the source address is not this

node's unicast address, resend the packet with a decremented TTL. In other words, the message propagates through the network, traversing a maximum of TTL hops.

Security is implemented on both the network and application layer. All messages within a network are encrypted with a common cryptographic network key, and application keys can be used to keep each application secure. ([8], p. 108) For example, if a common mesh network is used for lighting control and door control, a compromised light switch can not compromise the door control system. Furthermore, a separate key, unique to each device, is used for configuration.

2.2.2 Power efficiency

BLE is designed for devices that operate for several years on a coin cell battery. However, this is dependent on the radio not transmitting *nor* receiving for most of the time. As described in Section 2.2.1, when not transmitting, the BT Mesh nodes listen as much of the time as possible. This is currently not reconcilable with ultra low power consumption.

To avoid using power, sensor nodes in a BT Mesh network can turn off its radio most of the time. Another node within radio range can act as a receive buffer. The BT Mesh specification calls this functionality *friendship*. ([8], pp. 74-88) In this case, the low power node can send messages when needed, and poll the friend node as often as is appropriate for the application.

2.2.3 Latency and real-time properties

The BT Mesh specification does not specify hard timing requirements for the propagation of the messages. The protocol can guarantee delivery by using acknowledgements, but not maximum latency.

The approach described in Section 2.2.1 has the advantage of making sure that in practice, small messages reach their target reliably and very fast, compared to routing protocols like Zigbee, but large messages (segmented across multiple transmissions) and large networks with hundreds of relays can cause congestion and delays. Both of these effects are noted in [10].

Simulated results have shown that the increased latencies in a large and dense network (such as an office building with hundreds of nodes) can be drastically improved by letting only a subset of the nodes act as relays. [12]

2.3 Acquiring a common time reference

The tiny-sync algorithm, proposed in [1], can be used to estimate – with tight upper and lower bounds – the state of a timer on another connected device over a network with variable and unknown latency. This is achieved by sending a probe to a time master and receiving a response. The transmission and reception events are time-stamped using a local clock, and the remote time master time-stamps its reception of the probe and transmission of the response. With multiple sets of these time-stamps, upper and lower bounds for the clock offset and drift are calculated. The algorithm is based on the recognition of the fact that these events happen in a certain chronological order, and on the assumption that the clock drift is linear for much of the time.

Rather than keeping all the data and calculating an optimal solution, tiny-sync uses only a few

(2, 3 or 4) of the time-stamp sets in the calculations. In practice, this was shown to be very close to the optimal solution from all data. The metrics by which to select the "best" data to keep is presented in [1], but the process of choosing the points to keep is not presented. Therefore, Chapter 4 in this project covers a C implementation of an algorithm that selects the best data points, and simulated results derived from this implementation are presented in Chapter 5.

2.4 Litterature search summary

In this chapter, we have briefly covered some popular current low-power wireless technologies, placing BT Mesh among other low-power wireless mesh technologies like Zigbee, noting BT Mesh's main distinguishing feature as being compatible with other Bluetooth hardware like smartphones. Focusing on the properties of BT Mesh in particular, we pointed to the advantages and limitations of BT Mesh's way of distributing messages across the network. Finally, the tiny-sync algorithm, which is the main focus of the rest of this report, is presented.

It is worth noting that quantitative sources on the performance of the different mesh networks are sparse, and independent research even more so. The main resource used for discussing performance of the mesh networks in this chapter is a series of tests performed by Silicon Labs [10] using their own implementation of the different networks.

3 The Tiny-sync Time Synchronization Scheme

3.1 Background

The details of the tiny-sync algorithm is presented in [1]. In this section, we summarize the principles and nomenclature necessary to understand the proposed constraint selection algorithm and why it works.

Let us assume that a device, *node 1*, shall use a clock on another device, *node 2* as a time reference for data collection. In other words, node 1 will collect some sensor data, time-stamp the collection using its own clock (t_1). It shall then estimate what state of node 2's clock (t_2) that corresponds to. It is assumed that for considerable periods of time, the relationship between the clocks is linear ([1], Equation 2):

$$t_1(t) = a_{12}t_2(t) + b_{12} \tag{3.1}$$

The tiny-sync algorithm is concerned with estimating a_{12} and b_{12} as precisely as possible, while minimizing resource usage. To achieve this, a series of probes are sent from node 1 to node 2. Node 2 responds to the probe, The transmission (t_o) and reception (t_r) events are time-stamped using a local clock, and the remote device time-stamps its reception of the probe (t_b), and transmits that time-stamp back to node 1. This transaction results in the data point (t_o, t_r, t_b). Since these time-stamps must have been recorded in chronological order, the inequalities

$$t_o < a_{12}t_b + b_{12} < t_r \tag{3.2}$$

must hold. ([1], Equations 3, 4) This constrains the possible values of the unknowns a_{12} and b_{12}. This can be interpreted graphically as upper and lower bounds on the line in the (t_2, t_1) space defined by Equation (3.1) (this representation is used in multiple figures in [1]). The process is repeated to generate multiple such data points. In this project, we will refer to the upper constraint defined by values of t_r and t_b as B_i, where i is some index, and the lower constraint defined by values of t_o and t_b as A_i, where i is some index. This is in keeping with the nomenclature used in [1].

All of the constraints could be kept in memory, and the maximum and minimum values for a_{12} and b_{12} could be computed using linear programming. However, tiny-sync only keeps and uses two upper and two lower constraints, to minimize memory and processor usage. In practice, this was shown to perform very close to the optimal solution. Each time a new data point is received, the best two upper and best two lower are kept. The metrics by which to select the "best" constraints is defined in [1] as those that will result in the tightest bounds on a_{12} and b_{12}. In Section 3.2, a practical algorithm to select the best constraints is presented.

3.2 Algorithm for constraint selection

When receiving the first two data points, there is no selection to be done. They are simply stored. When receiving a new data point, two new constrains are added. We are then left with 6 constraints. We name them as shown in Table 2. We will refer to the lines through constraints X_i and Y_j as X_iY_j.

	The oldest	The previously newest	The brand new
upper constraint	B_1	B_2	B_3
lower constraint	A_1	A_2	A_3

Table 2: Names used for the different constraints used in the algorithm.

Consider the constraints shown in Figure 1. Before the new constraints are known, the line A_1B_2 represents the maximum value of a_{12} and the minimum value of b_{12}, as its slope and intersect, respectively. Given that both clocks have advanced from the previous probe, B_3 can be in one of the four colored regions:

- If B_3 is above the line A_1B_2, then the line A_1B_3 has a both higher slope than A_1B_2 and a lower intersect. [1]In other words, using B_3 instead of B_2 does not provides a tighter estimate, and B_3 should be discarded.
- If B_3 is below the line A_1B_2, then the line A_1B_3 has a both lower slope than A_1B_2 and a higher intersect. In other words, using B_3 instead of B_2 provides a tighter estimate, and B_3 should replace B_2.
- If B_3 is also below the line A_1A_2, then the line A_2B_3 has a both lower slope than A_1B_3 and a higher intersect. In other words B_3 should replace B_2 AND A_2 should replace A_1.
- Finally, if B_3 is below the line B_1A_2, then there exists no line that can pass within all constraints. This means that the linearity assumption does not hold. A_1 and B_1 should be discarded, and the algorithm should restart.

The same arguments can be used based on the placement of A_3, swapping As with Bs, "above" with "below" and "higher" with "lower".

[1]In all these comparisons, two lines compared pass through a common point. Therefore, a lower slope is equivalent to a higher intersect and vice versa. This means that we do not have to choose between a better bound on a_{12} and b_{12}. We get both.

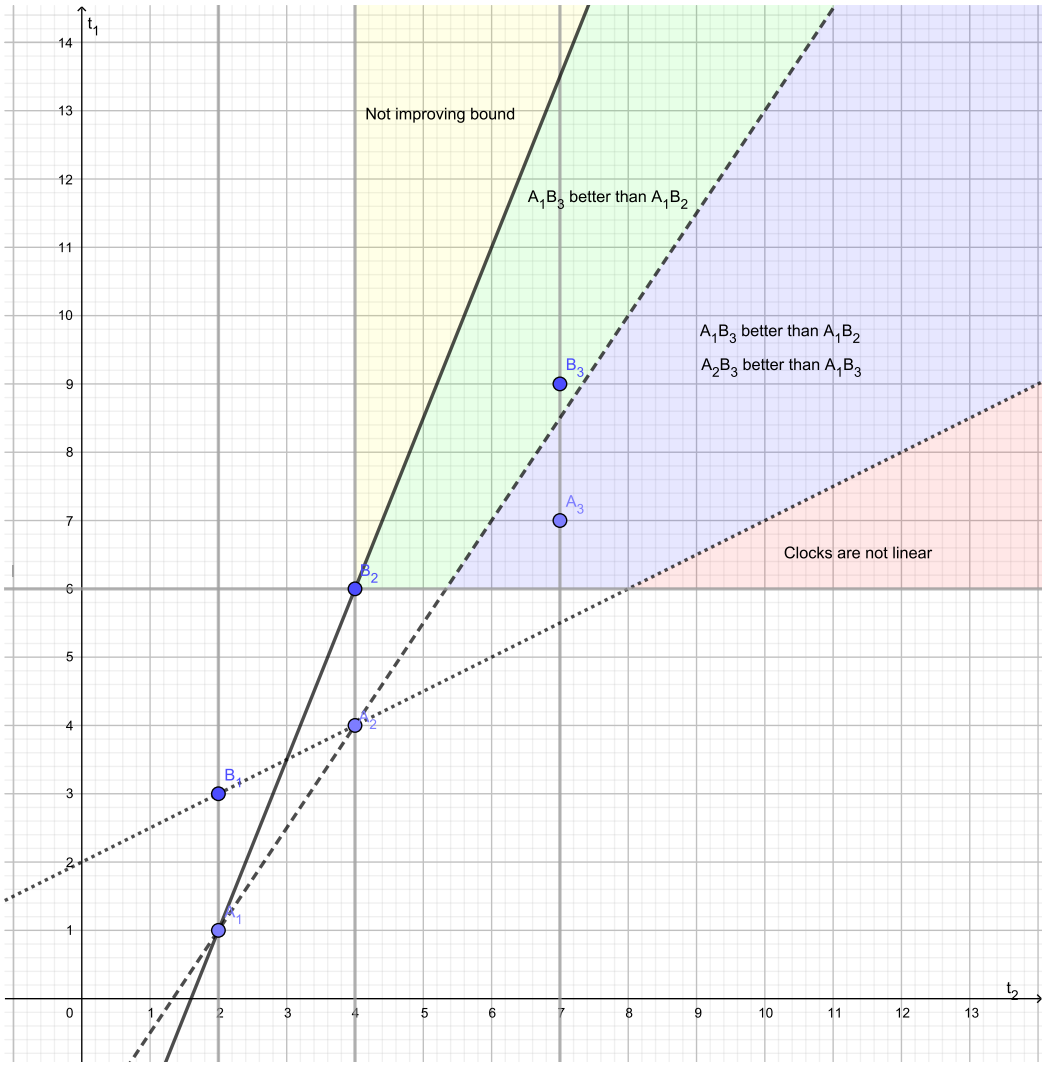

Figure 1: The possible values for a new upper constraint B_3, and the significance of the placement. In the example shown here, B_3 should replace B_2.

4 An Implementation of the Tiny-Sync Constraint Selection Algorithm in C

In this chapter, a C implementation of the tiny-sync algorithm is presented in the form of a single-function library. First, the needed data structures are defined. Then, the details of the initialization is covered. Finally, the implementation of the actual constraint selection is presented.

Code listings are used in this chapter to highlight the critical pieces of code. The complete code for both the tiny-sync implementation and the testing framework is available in Appendix A, and is attached.

4.1 Data structures and state

First, we define data structures for the basic concepts from Chapter 3: the data point consisting of t_o, t_b and t_r, as well as the constraint, consisting of one value for each time dimension t_1 and t_2. We also need a way to store a line, by its slope a and intercept b. The definitions are listed in Listing 4.1. Note that we use 64-bit data types everywhere in this implementation, to minimize integer overflows or rounding errors. When running on a 32-bit system like Nordic's nRF52840, these can be changed to 32-bit types for halved storage requirement and faster calculations.

```
typedef struct tinysync_datapoint_t{
    uint64_t t_o; // t_1 when probe was sent
    uint64_t t_b; // t_2 when response was sent
    uint64_t t_r; // t_1 when response was received
} tinysync_datapoint_t;

typedef struct tinysync_constraint_t{
    uint64_t t_1; // prober clock time
    uint64_t t_2; // responder clock time
} tinysync_constraint_t;

// Line t_1 = a(t_2) + b
typedef struct tinysync_line_t{
    double a; // slope of line
    double b; // offset of line
} tinysync_line_t;
```

Listing 4.1: Definition of the basic data structures used in tiny-sync

More data structures are defined for state stored from iteration to iteration. The state consists of:

- A set of 4 constraints (two upper, two lower)
- A set of 4 lines (those used in the algorithm). As mentioned in Section 3.2, two of these lines (A_1B_2 and B_1A_2) define the maximum and minimum a_{12} and b_{12}.
- A counter used for initialization

The definitions of these structures are listed in Listing 4.2. Note that the lines do not actually have to be stored in the state; exactly the same information is contained in the constraints. By not storing them, the storage requirement is halved. However, because the lines both define the maximum and minimum a_{12} and b_{12} (being the main output) and are used in the next iteration, it is useful to just keep them. In other words, it is a trade-off between storage and computation.

```
// Stored set of constraints
typedef struct tinysync_constraints_t{
    tinysync_constraint_t b_1; // Constraint B_1
    tinysync_constraint_t b_2; // Constraint B_1
    tinysync_constraint_t a_1; // Constraint A_1
    tinysync_constraint_t a_2; // Constraint A_1
} tinysync_constraints_t;

// Stored set of 4 lines between 4 constraints
typedef struct tinysync_lineset_t{
    tinysync_line_t ba; // Line B_1 A_2
    tinysync_line_t ab; // Line A_1 B_2
    tinysync_line_t aa; // Line A_1 A_2
    tinysync_line_t bb; // Line B_1 B_2
} tinysync_lineset_t;

typedef struct tinysync_est_state_t{
    tinysync_constraints_t constraints;
    tinysync_lineset_t lineset;
    uint8_t init_counter;
} tinysync_est_state_t;
```

Listing 4.2: Definition of the data structures held as state between iterations

4.2 Initializing the algorithm

The first two times the algorithm is run, its behaviour is special, as shown in Listing 4.3. The first time, only one data point is available. In other words, no estimate of the clock drift or offset can be made. The first constraints are added to the state, and the algorithm is cancelled.

The second time, the new constraints are added. With 4 constraints, the first estimate can now be made. The constraint selection part is skipped, and the 4 lines are calculated.

```
switch(state->init_counter){
case 0: // First run: 1 data point, cannot estimate: directly to state and early return
    state->constraints.b_1 = b_3;
    state->constraints.a_1 = a_3;
    state->init_counter ++;
    return TINYSYNC_EST_FIRST; // Skip rest of algorithm
```

```
    break;
    case 1: // Second run: 2 data points, can estimate
        b_2 = b_3;
        a_2 = a_3;
        state->init_counter ++;
        ret = TINYSYNC_EST_SECOND;
        break; // Skip constraint selection
    default: // Algorithm initialized, normal behaviour
```

Listing 4.3: Initialization of the algorithm, affecting the first two iterations

4.3 Constraint selection

The constraint selection proceeds with the same conditions that are presented in Section 3.2, once for each of the two new constraints. The same nomenclature for the constraints is also used (Table 2). For easy comparison, Listing 4.4 lists the part that deals with the new upper constraint (B_3). The same algorithm is repeated for handling A_3, with all a's swapped with b's and vice versa, and reversed inequality signs.

The state is not updated directly here, but rather through temporary variables that are transferred to the state at the end of the function. This is to ensure that no constraints are discarded before the algorithm is complete.

At the end of the function, the stored constraints are updated from the temporary variables, and the lines are computed. This allows the minimum and maximum estimates to be read from the state structure, and it prepares the lines for the next iteration. For brevity, The state update and computation of the lines are not shown in the listing. It can be seen in Appendix A.

```
    // Handle new upper constraint (b_3)
    if        (b_3.t_1 > state->lineset.ab.b
                      + state->lineset.ab.a * (double)(b_3.t_2)){
        // Upper constraint too high to be useful: do nothing
    } else if (b_3.t_1 < state->lineset.ba.b
                      + state->lineset.ba.a * (double)(b_3.t_2)){
        // Upper constraint too low to be linear: remove old constraints and add new
        b_1 = state->constraints.b_2;
        a_1 = state->constraints.a_2;
        b_2 = b_3;
        a_2 = a_3;
        ret = TINYSYNC_EST_NONLINEAR;
        break;
    } else {
        // Upper constraint useful: update b_2
        b_2 = b_3;
        if (  (!tinysync_constraint_t_compare(state->constraints.a_1,
                                              state->constraints.a_2)) &&
              (b_3.t_1 < state->lineset.aa.b
                      + state->lineset.aa.a * (double)(b_3.t_2))){
            // a_2 is better than a_1 for this upper constraint: update a_1
            a_1 = state->constraints.a_2;
        }
    }
```

Listing 4.4: Constraint selection algorithm responsible for setting A_1 and B_2

5 Testing the Implemented Algorithm

5.1 Simulation framework

In order to test the tiny-sync implementation, a basic simulation framework was created. The framework simulates the process of node 1 probing node 2. The two clocks advance at different rates, and an initial offset. The framework also introduces configurable random delays between all relevant events, to simulate variations due to scheduling, radio contention, and similar random conditions.

A "non-linearity event" is configured to happen at a certain time. After the non-linearity event, the clock skew a_{12} changes abruptly to a different value. This is used to simulate an event like sudden temperature change.

First, a series of test settings are defined by preprocessor directives. An example is shown in Listing 5.1. In this case, the constants used for testing detection of non-linearity are used. For the purposes of testing, we will assume that the clock on node 2 is a perfect timer, incrementing each microsecond of wall time. In other words, t_2 is interchangeable with the time dimension in the settings and the result graphs.

```
#define S 1000000L // Factor for seconds
#define MS 1000L // Factor for milliseconds
#define TEST_N 10000L // number of iterations to run test
#define TEST_A12 1.4 // Clock skew
#define TEST_B12 5L*S // Clock offset
#define TEST_INTERVAL 1*S // Test interval, t_2 units
#define TEST_INTERVAL_RAND 100*MS // error magnitude of same
#define TEST_DELAY_B 50*MS // delay from t_o to t_b
#define TEST_DELAY_B_RAND 15*MS // error magnitude of same
#define TEST_DELAY_R 50*MS // delay from t_b to t_r
#define TEST_DELAY_R_RAND 15*MS // error magnitude of same
#define TEST_NONLINEAR_T_2_START 500*S // Time of non-linearity event
#define TEST_NONLINEAR_A12 1.6 // new A12 after non-linearity
```

Listing 5.1: Test configuration constants

A data structure is defined as shown in Listing 5.2 for storing the simulated state of each clock during the simulation.

```
// Stored clocks of t_1 and t_2, for testing
typedef struct test_moment_t{
    uint64_t t_1; // Current state of node 1's timer
    uint64_t t_2; // Current state of node 2's timer, and true wall time
} test_moment_t;
```

Listing 5.2: Data structure for storing clock states

A function is then defined for the probe simulation, shown in Listing 5.3. Between each event, t_2 is advanced with a time defined by the settings, with a random component. t_1 is advanced

the same amount, multiplied by a_{12}. The `random_error` function returns a uniformly distributed random number with a maximum magnitude of the argument. a_{12} is either set to `TEST_A12` or to `TEST_NONLINEAR_A12`, based on whether the non-linearity event has happened.

```
void test_advance_timestamp(tinysync_datapoint_t* d, test_moment_t* now){
    double test_a12 = (now->t_2 < TEST_NONLINEAR_T_2_START) ? TEST_A12 :
                                                              TEST_NONLINEAR_A12;

    int64_t error = random_error(TEST_INTERVAL_RAND);
    now->t_1 += (TEST_INTERVAL + error) * test_a12;
    now->t_2 += (TEST_INTERVAL + error);
    d->t_o = now->t_1;

    error = random_error(TEST_DELAY_B_RAND);
    now->t_1 += (TEST_DELAY_B + error) * test_a12;
    now->t_2 += (TEST_DELAY_B + error);
    d->t_b = now->t_2;

    error = random_error(TEST_DELAY_R_RAND);
    now->t_1 += (TEST_DELAY_R + error) * test_a12;
    now->t_2 += (TEST_DELAY_R + error);
    d->t_r = now->t_1;
}
```

Listing 5.3: Function for generating new data points

In the test program, listed in Listing 5.4, the `test_advance_time-stamp` function is run many times. the resulting maximum and minimum time estimates are computed, and various statistics are printed for each iteration. The output was used to generate the graphs in Section 5.2.

5.2 Simulation results

The test program was run 3 times with different parameters, to evaluate different properties of the algorithm. One simulation focuses on the ability to attain a close estimate, one focuses on the consequences of asymmetrical network delays, and one focuses on the consequences of a the clocks not being linearly related.

The settings for the delays in the simulations were inspired by latencies measured for short packets on embedded mesh networks, including BT Mesh, on the order of 10-100 ms ([10], Figures 2.6 and 2.8). The error of the estimated t_2, as well as the error of the minimum and maximum estimates are plotted over time for each of the runs, and is shown in Figure 2 (run 1), Figure 3 (run 2) and Figure 4 (run 3). The settings used for each simulation is summarized in table Table 3.

```c
int main(){
    tinysync_est_state_t state;
    tinysync_est_state_t_initialize(&state); // Set initialization counter to zero

    test_moment_t now = {.t_1 = TEST_B12, // Start off the clocks with an offset
                         .t_2 = 0};
    tinysync_datapoint_t d1;

    for(uint64_t i=0; i<TEST_N; i++){
        test_advance_timestamp(&d1, &now);
        tinysync_est_ret_t ret = tinysync_est_etimate(&state, &d1);
        double b_12_exp = (state.lineset.ba.a + state.lineset.ab.a) / 2.0;
        double a_12_exp = (state.lineset.ba.b + state.lineset.ab.b) / 2.0;
        uint64_t estimated_t_2 = (uint64_t)( (((double)now.t_1) - a_12_exp) / b_12_exp );
        uint64_t max_t_2 = (uint64_t)( (((double)now.t_1) - state.lineset.ba.b) /
                                                state.lineset.ba.a );
        uint64_t min_t_2 = (uint64_t)( (((double)now.t_1) - state.lineset.ab.b) /
                                                state.lineset.ab.a );
        printf("test %u %.9g %.9g %.9g %.9g %u %.9g %.9g %.9g %.9g %u %d %f %d %d %d\n",
               ret, // Return code
               state.lineset.ba.a, // drift lower limit
               state.lineset.ab.a, // drift upper limit
               state.lineset.ba.b, // offset upper limit
               state.lineset.ab.b, // offset lower limit
               now.t_2,
               b_12_exp, // Expected offset
               fabs(b_12_exp - TEST_B12), // B12 absolute error
               a_12_exp, // Expected skew
               fabs(a_12_exp - TEST_A12), // A12 absolute error
               estimated_t_2,
               estimated_t_2 - now.t_2, // Error of t_2 estimate
               fabs((int64_t)estimated_t_2 - (int64_t)now.t_2), // Absolute same
               max_t_2 - now.t_2, //Error of max; always positive
               min_t_2 - now.t_2, //Error of min; always negative
               (max_t_2 - min_t_2) / 2  //Maximum possible absolute error
               );
    }
    return 0;
}
```

Listing 5.4: Main program used to test the implementation

Setting	Description	Value, run 1 (basic test)	Value, run 2 (asymmetrical delays)	Value, run 3 (non-linear clocks)
TEST_A12	simulated a_{12}	1.4		
TEST_B12	simulated b_{12}	5 000 000 cycles of t_1		
TEST_INTERVAL	Test interval	1 s		
TEST_INTERVAL_RAND	variation magnitude of the above	100 ms		
TEST_DELAY_B	delay from t_o to t_b	50 ms	35 ms	50 ms
TEST_DELAY_B_RAND	variation magnitude of the above	15 ms	10 ms	15 ms
TEST_DELAY_R	delay from t_b to t_r	50 ms	10 ms	50 ms
TEST_DELAY_R_RAND	variation magnitude of the above	15 ms	5 ms	15 ms
TEST_NONLINEAR_T_2_START	Time of non-linearity event	(outside of captured data)		500 s
TEST_NONLINEAR_A12	new a_{12} after non-linearity	(irrelevant)		1.6

Table 3: Settings used for each simulation.

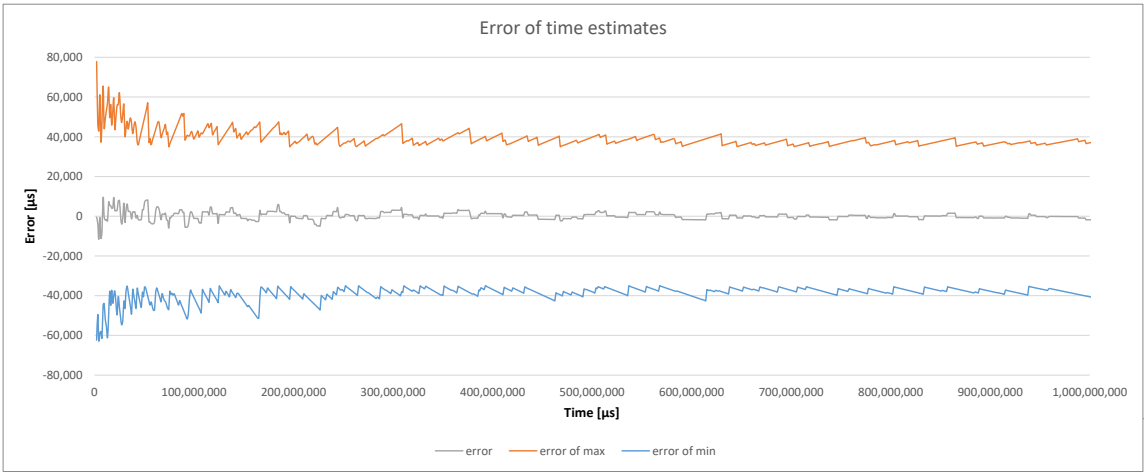

Figure 2: Simulated synchronization errors of tiny-sync when using the algorithm implemented in Chapter 4, in run 1. Errors of the minimum and maximum estimate is shown, as well as the average estimate.

Figure 3: Simulated synchronization errors of tiny-sync when using the algorithm implemented in Chapter 4, in run 2. Errors of the minimum and maximum estimate is shown, as well as the average estimate. Note the constant error of the average estimate due to the difference in probe and reply delay.

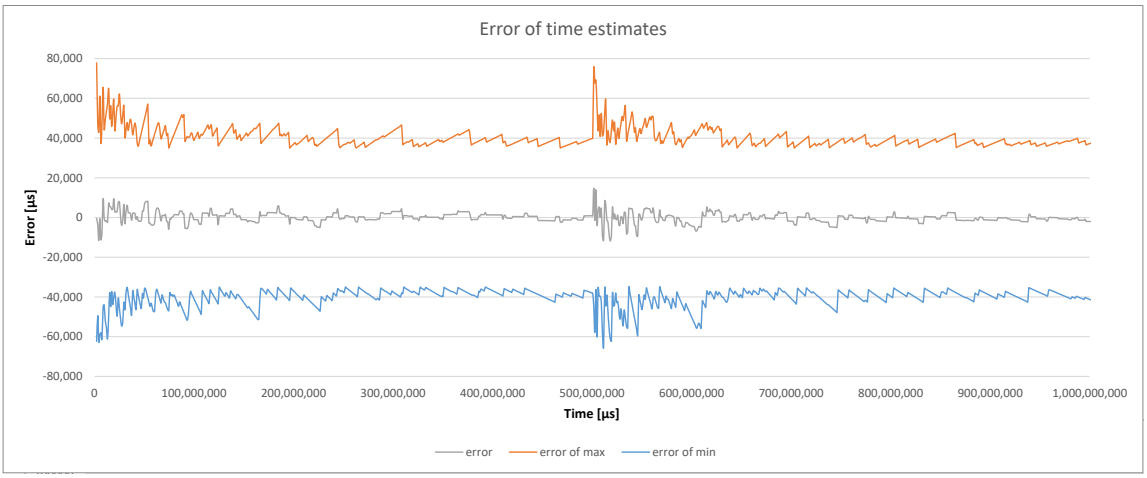

Figure 4: Simulated synchronization errors of tiny-sync when using the algorithm implemented in Chapter 4, in run 3. Errors of the minimum and maximum estimate is shown, as well as the average estimate. Note the sudden jump after the non-linearity event at 500 seconds.

6 Discussion

6.1 Precision of the estimate

We can see from Figure 2 that when the clocks are linear, and the delays are symmetric, our tiny-sync implementation generates an estimate that grows increasingly accurate and more consistent over time. With the particular settings used in this case, stable millisecond-level precision is achieved after a few hundred seconds. This suggests that if this implementation is used to synchronize nodes in even a fairly large BT Mesh network in the most naive way – that is, each nodes using the network to synchronize with some time master at the software level – this level of precision can be reached. This assumes that the minimum delays in communication is symmetric.

6.2 Constant error given asymmetric delays

If the minimum delays are asymmetric, the estimate has a constant component in its error. This is seen in Figure 3. This asymmetry could have many reasons. For example, the network stack may have different delays for incoming and outgoing packets. A modification to the data point, to account for an extra delay between node 2's reception of the probe and its transmission of the response, is presented in ([1] section 3.1.1). In essence, the known delay is added to the first time-stamp. In principle, the same idea could be used to account for any minimum known delay between the 3 time-stamps.

6.3 Guaranteed upper and lower bounds

Regardless of whether the minimum delays are asymmetric, two things remains true in all the results: the error of the maximum estimate is always positive, and the error of the minimum estimate is always negative. In other words, node 1 has a guarantee that the t_2 lies between the maximum and minimum estimate, and can know the uncertainty of its synchronization. Knowing this uncertainty could be useful in a number of ways. One example would be location systems using time difference of arrival, where the range of possible times for reception of a signal could yield a shape within which the target is located with certainty, rather than just an estimate. A more topical example would be synchronization in a BT Mesh network with low power nodes, where the synchronization exchange could take place only as often as is necessary to maintain a certain accuracy. Since radio usage time is a very important factor for power draw in low-power systems, this could make a substantial difference, compared to just synchronizing at a fixed period.

6.4 Extending the synchronization from a pair to a network

To synchronize an entire BT Mesh network with this algorithm could be done in several ways. The very simplest would be to send the probes to a time master and receive the responses as messages across the network. This would, however, result in sub-optimal accuracy if the probe and response pass through multiple nodes and are forwarded. With hardware-level time-stamping of the radio transmissions and receptions, the delays between two nodes could be measured on the order of nanoseconds, while a transmission through multiple nodes takes milliseconds. The network could be synchronized using pairwise synchronization, which is mentioned in [1]. However, this would require the nodes to build a tree, manage pairwise relations with each other, and exchange more information. What approach to implement will depend on the accuracy required for the use case.

6.5 Response to non-linearity

Looking at Figure 4, we can see a sharp increase in the uncertainty at the 500 second mark. This is the point at which a_{12} suddenly changes from 1.4 to 1.6. As the old constraints can no longer be used to form estimates, they are discarded at this point. The path of the estimates after that looks much like it does right at the beginning. This makes sense, as the algorithm has in effect been restarted after detecting the non-linearity.

7 Conclusion

In Chapter 2, we surveyed different low-power wireless technologies, and placed BT Mesh among other the other low-power mesh technologies. We also investigated the particular details of BT Mesh. Of particular note is BT Mesh's ability to include smartphones and other Bluetooth-enabled devices to the network, which could be useful if a user wishes to interact with the network. Also of note is that a BT Mesh network can be configured heterogeneously: to improve performance of large networks, only some nodes can be configured as relays. To improve power consumption, some nodes can be configured as low-power nodes. The nature of the BT Mesh network prevents it from giving guarantees for latencies, but in practice, latencies have been achieved on the order of tens of milliseconds in large networks with 8-byte messages.

The tiny-sync algorithm was implemented in C as a method of synchronizing timers distributed across a low-power wireless network. This algorithm has many desirable properties, most notably that it generates bounds on its accuracy. The implementation was tested in a simulation framework. The simulated results yielded some insights into how this this might be used. First, it may be possible to achieve millisecond-level accuracy over a BT Mesh network (or any other network with similar latencies), even with an application-level implementation that does not care about the underlying network. If the underlying network is considered, and known delays are accounted for, much higher accuracy is possible. What approach to take will depend on the accuracy required for the use case.

7.1 Future work

Quantitative sources on the performance of the different mesh networks are sparse, and independent research even more so. Reproducing the results of [10] using various hardware and software would go a long way to providing a larger data set for evaluating the properties of the different networks. Accurately measuring the power consumption of these networks during these tests would also be valuable, as that would provide insights into the trade-offs and compromises made in each technology.

For the purposes of connecting multiple devices at a site, a more holistic comparison of BLE and/or BT Mesh with cellular solutions like NB-IoT, considering power efficiency, scalability, bandwidth and cost would be valuable.

The tiny-sync implementation presented could probably be optimized in various ways. For example, the effect of the data types (fixed point/floating point/integer) of the different values on the precision could be investigated. Further, the current implementation stores redundant data, in the form of the `lineset` structure.

Testing of the tiny-sync algorithm over BLE and BT Mesh on embedded hardware like the Nordic nRF52840 System on Chip is an obvious next step. Investigating the effects of integrating more tightly with the network stack – like using time-stamps from the radio hardware itself – would be valuable.

Sources

[1] Yoon, S., Veerarittiphan, C., & Sichitiu, M. L. June 2007. Tiny-sync: Tight time synchronization for wireless sensor networks. *ACM Trans. Sen. Netw.*, 3(2). URL: http://doi.acm.org/10.1145/1240226.1240228, doi:10.1145/1240226.1240228.

[2] 3GPP. 2016. Standards for the iot. http://www.3gpp.org/images/presentations/2016_11_3gpp_Standards_for_IoT.pdf. Accessed: 2018-12-02.

[3] GSM Association. 2018. Mobile iot lpwa - lte-m & nb-iot commercial launches. https://www.gsma.com/iot/mobile-iot-commercial-launches/. Accessed: 2018-12-02.

[4] Semtech. 2018. What is lora? https://www.semtech.com/lora/what-is-lora. Accessed: 2018-12-02.

[5] Zigbee Alliance. 2018. What is zigbee. https://www.zigbee.org/what-is-zigbee/. Accessed: 2018-11-14.

[6] Thread Group. 2018. Thread overview page. https://www.threadgroup.org/What-is-Thread/Overview. Accessed: 2018-11-14.

[7] IEEE. April 2016. Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, 1–709. doi:10.1109/IEEESTD.2016.7460875.

[8] Bluetooth SIG. 2017. Mesh profile specification 1.0. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=429633. Accessed: 2018-11-14.

[9] Texas Instruments. 2018. Thread vs zigbee – what's the difference? https://e2e.ti.com/blogs_/b/connecting_wirelessly/archive/2018/05/16/thread-vs-zigbee-what-s-the-difference. Accessed: 2018-11-15.

[10] Silicon Labs. 2018. Mesh network performance comparison. https://www.silabs.com/documents/login/application-notes/an1142-mesh-network-performance-comparison.pdf. Accessed: 2018-11-15.

[11] Bluetooth SIG. 2016. Bluetooth core specification, verison 5. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043. Accessed: 2018-11-14.

[12] Ericsson. 2017. Bluetooth mesh networking. https://www.ericsson.com/assets/local/publications/white-papers/wp-bluetooth-mesh_ver2_171115-c2.pdf. Accessed: 2018-11-15.

A Tiny-sync Implementation Source Files

A.1 Interface: estimator.h

```
#ifndef __TINYSYNC_ESTIMATOR_H__
#define __TINYSYNC_ESTIMATOR_H__

#include <stdint.h>

typedef enum tinysync_est_ret_t{
    TINYSYNC_EST_OK = 0,
    TINYSYNC_EST_FIRST = 1,
    TINYSYNC_EST_SECOND = 2,
    TINYSYNC_EST_NONLINEAR = 3,
    TINYSYNC_EST_INVALID = 4 // Should never occur
} tinysync_est_ret_t;

typedef struct tinysync_datapoint_t{
    uint64_t t_o; // t_1 when probe was sent (+ minimum known delay until response transmission)
    uint64_t t_b; // t_2 when response was sent
    uint64_t t_r; // t_1 when response was received
} tinysync_datapoint_t;

typedef struct tinysync_constraint_t{
    uint64_t t_1; // prober clock time
    uint64_t t_2; // responder clock time
} tinysync_constraint_t;

// Line t_1 = a(t_2) + b
typedef struct tinysync_line_t{
    double a; // slope of line
    double b; // offset of line
} tinysync_line_t;

// Stored set of constraints
typedef struct tinysync_constraints_t{
    tinysync_constraint_t b_1; // Constraint B_1
    tinysync_constraint_t b_2; // Constraint B_1
    tinysync_constraint_t a_1; // Constraint A_1
    tinysync_constraint_t a_2; // Constraint A_1
} tinysync_constraints_t;

// Stored set of 4 lines between 4 constraints
typedef struct tinysync_lineset_t{
    tinysync_line_t ba; // Line B_1 A_2
    tinysync_line_t ab; // Line A_1 B_2
    tinysync_line_t aa; // Line A_1 A_2
    tinysync_line_t bb; // Line B_1 B_2
} tinysync_lineset_t;

typedef struct tinysync_est_state_t{
```

```
48      tinysync_constraints_t constraints;
49      tinysync_lineset_t lineset;
50      uint8_t init_counter;
51  } tinysync_est_state_t;
52
53  void tinysync_est_state_t_initialize(tinysync_est_state_t* state);
54  tinysync_est_ret_t tinysync_est_etimate(tinysync_est_state_t* state,
55                                          const tinysync_datapoint_t* new_datapoint);
56
57  #endif //__TINYSYNC_ESTIMATOR_H__
```

Listing A.1: Tiny-sync implementation interface

A.2 Implementation: estimator.c

```c
#include "estimator.h"

uint8_t tinysync_constraint_t_compare(tinysync_constraint_t p_1, tinysync_constraint_t p_2){
    return (p_1.t_1 == p_2.t_1 &&
            p_1.t_2 == p_2.t_2) ? 1 : 0;
}

void tinysync_est_state_t_initialize(tinysync_est_state_t * state){
    state->init_counter = 0;
}

void calculate_line(tinysync_constraint_t p_1, tinysync_constraint_t p_2, tinysync_line_t * line){
    line->a = (double)(p_2.t_1 - p_1.t_1) / (double)(p_2.t_2 - p_1.t_2);
    line->b = (double)(p_1.t_1) - line->a * (double)(p_1.t_2);
}

//estimator of linear clock drift and offset
tinysync_est_ret_t tinysync_est_etimate(tinysync_est_state_t * state, const tinysync_datapoint_t *
     new_datapoint){
    tinysync_est_ret_t ret = TINYSYNC_EST_OK;

    // default to keeping old points.
    // These new variables should only be written, not read, until after constraint selection
    tinysync_constraint_t b_1 = state->constraints.b_1;
    tinysync_constraint_t a_1 = state->constraints.a_1;
    tinysync_constraint_t b_2 = state->constraints.b_2;
    tinysync_constraint_t a_2 = state->constraints.a_2;

    // Convert probing datapoint to 2 points in the t_1-t_2 plane.
    tinysync_constraint_t b_3;
    tinysync_constraint_t a_3;
    b_3.t_1 = new_datapoint->t_r;
    b_3.t_2 = new_datapoint->t_b;
    a_3.t_1 = new_datapoint->t_o;
    a_3.t_2 = new_datapoint->t_b;

    // Handle constraint selection
    switch(state->init_counter){
    case 0: // First run: 1 data point, cannot estimate: directly to state and early return
        state->constraints.b_1 = b_3;
        state->constraints.a_1 = a_3;
        state->init_counter ++;
        return TINYSYNC_EST_FIRST;
    break;
    case 1: // Second run: 2 data points, can estimate
        b_2 = b_3;
        a_2 = a_3;
        state->init_counter ++;
        ret = TINYSYNC_EST_SECOND;
        break;
    default: // Algorithm initialized, normal behaviour

        // Handle new lower constraint (a_3)
        if        (a_3.t_1 < state->lineset.ba.b + state->lineset.ba.a * (double)(a_3.t_2)){
            // Lower constraint too low to be useful: do nothing
        } else if (a_3.t_1 > state->lineset.ab.b + state->lineset.ab.a * (double)(a_3.t_2)){
```

```
            // Lower constraint too high to maintain linear model: remove old constraints, add new
            b_1 = state->constraints.b_2;
            a_1 = state->constraints.a_2;
            b_2 = b_3;
            a_2 = a_3;
            ret = TINYSYNC_EST_NONLINEAR;
            break;
        } else {
            // Lower constraint useful: update a_2
            a_2 = a_3;
            if ( (!tinysync_constraint_t_compare(state->constraints.b_1,state->constraints.b_2)) && (
    a_3.t_1 > state->lineset.bb.b + state->lineset.bb.a * (double)(a_3.t_2)) ){
                // b_2 is better than b_1 for this lower constraint: update b_1
                b_1 = state->constraints.b_2;
            }
        }

        // Handle new upper constraint (b_3)
        if        (b_3.t_1 > state->lineset.ab.b + state->lineset.ab.a * (double)(b_3.t_2)){
            // Upper constraint too high to be useful: do nothing
        } else if (b_3.t_1 < state->lineset.ba.b + state->lineset.ba.a * (double)(b_3.t_2)){
            // Upper constraint too low to maintain linear model: remove old constraints, add new
            b_1 = state->constraints.b_2;
            a_1 = state->constraints.a_2;
            b_2 = b_3;
            a_2 = a_3;
            ret = TINYSYNC_EST_NONLINEAR;
            break;
        } else {
            // Upper constraint useful: update b_2
            b_2 = b_3;
            if ( (!tinysync_constraint_t_compare(state->constraints.a_1,state->constraints.a_2)) && (
    b_3.t_1 < state->lineset.aa.b + state->lineset.aa.a * (double)(b_3.t_2)) ){
                // a_2 is better than a_1 for this upper constraint: update a_1
                a_1 = state->constraints.a_2;
            }
        }
    }

    // Update stored points
    state->constraints.b_1 = b_1;
    state->constraints.a_1 = a_1;
    state->constraints.b_2 = b_2;
    state->constraints.a_2 = a_2;

    // Update lines
    calculate_line(b_1, b_2, &(state->lineset.bb));
    calculate_line(b_1, a_2, &(state->lineset.ba));
    calculate_line(a_1, b_2, &(state->lineset.ab));
    calculate_line(a_1, a_2, &(state->lineset.aa));

    return ret;
};
```

Listing A.2: Tiny-sync implementation

A.3 Test framework: main.c

```c
// main.c for testing library
#include <stdint.h>
#include "estimator.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define S 1000000L
#define MS 1000L

#define TEST_N 10000L // number of iterations to run test
#define TEST_A12 1.4 // Clock skew
#define TEST_B12 5L*S // Clock offset
#define TEST_INTERVAL 1*S // Test interval, t_2 units
#define TEST_INTERVAL_RAND 100*MS // error magnitude of same
#define TEST_DELAY_B 50*MS // delay from t_o to t_b
#define TEST_DELAY_B_RAND 15*MS // error magnitude of same
#define TEST_DELAY_R 50*MS // delay from t_b to t_r
#define TEST_DELAY_R_RAND 15*MS // error magnitude of same
#define TEST_NONLINEAR_T_2_START 500*S // Time of nonlinearity event
#define TEST_NONLINEAR_A12         1.6 // new A12 after nonlinearity

// Generate a random (uniform) number with a maximum magnitude of 'range'
int64_t random_error(uint64_t range){
    if(range == 0){
        return 0;
    }else{
        uint64_t magnitude = rand() % range;
        int8_t sign = (rand() % 2 ? 1 : -1);
        return sign * magnitude;
    }
}

// Stored clocks of t_1 and t_2, for testing
typedef struct test_moment_t{
    uint64_t t_1; // Current state of node 1's timer
    uint64_t t_2; // Current state of node 2's timer, and true wall time
} test_moment_t;

void test_advance_timestamp(tinysync_datapoint_t* d, test_moment_t* now){
    double test_a12 = (now->t_2 < TEST_NONLINEAR_T_2_START) ? TEST_A12 :
                                                              TEST_NONLINEAR_A12;

    int64_t error = random_error(TEST_INTERVAL_RAND);
    now->t_1 += (TEST_INTERVAL + error) * test_a12;
    now->t_2 += (TEST_INTERVAL + error);
    d->t_o = now->t_1;

    error = random_error(TEST_DELAY_B_RAND);
    now->t_1 += (TEST_DELAY_B + error) * test_a12;
    now->t_2 += (TEST_DELAY_B + error);
    d->t_b = now->t_2;

    error = random_error(TEST_DELAY_R_RAND);
    now->t_1 += (TEST_DELAY_R + error) * test_a12;
    now->t_2 += (TEST_DELAY_R + error);
```

```
    d->t_r = now->t_1;
}

int main(){
    tinysync_est_state_t state;
    tinysync_est_state_t_initialize(&state); // Set initialization counter to zero
    test_moment_t now = {.t_1 = TEST_B12, // Start off the clocks with an offset
                         .t_2 = 0};
    tinysync_datapoint_t d1;

    for(uint64_t i=0; i<TEST_N; i++){
        test_advance_timestamp(&d1, &now);
        tinysync_est_ret_t ret = tinysync_est_etimate(&state, &d1);
        double b_12_exp = (state.lineset.ba.a + state.lineset.ab.a) / 2.0;
        double a_12_exp = (state.lineset.ba.b + state.lineset.ab.b) / 2.0;
        uint64_t estimated_t_2 = (uint64_t)((((double)now.t_1) - a_12_exp) / b_12_exp);
        uint64_t max_t_2 = (uint64_t)((((double)now.t_1) - state.lineset.ba.b) / state.lineset.ba.a);
        uint64_t min_t_2 = (uint64_t)((((double)now.t_1) - state.lineset.ab.b) / state.lineset.ab.a);
        printf("test %u %.9g %.9g %.9g %.9g %u %.9g %.9g %.9g %.9g %u %d %f %d %d %d\n",
                ret, // Return code
                state.lineset.ba.a, // drift lower limit
                state.lineset.ab.a, // drift upper limit
                state.lineset.ba.b, // offset upper limit
                state.lineset.ab.b, // offset lower limit
                now.t_2,
                b_12_exp, // Expected offset
                fabs(b_12_exp - TEST_B12), // B12 absolute error
                a_12_exp, // Expected skew
                fabs(a_12_exp - TEST_A12), // A12 absolute error
                estimated_t_2,
                estimated_t_2 - now.t_2, // Error of t_2 estimate
                fabs((int64_t)estimated_t_2 - (int64_t)now.t_2), // Absolute same
                max_t_2 - now.t_2, //Error of max; always positive if done right
                min_t_2 - now.t_2, //Error of min; always negative if done right
                (max_t_2 - min_t_2) / 2  //Maximum possible absolute error
                );
    }
    return 0;
}
```

Listing A.3: Tiny-sync implementation test framework