Clock Synchronization with Deterministic Accuracy Guarantee

Ryo Sugihara Rajesh K. Gupta

Computer Science and Engineering Department, University of California, San Diego
{ryo,rgupta}@ucsd.edu

January 13, 2011

Abstract

Accuracy is one of the most important performance metrics in clock synchronization. While state-of-the-art synchronization protocols achieve μsec-order average accuracy, they usually do not focus on the worst case accuracy and do not have any deterministic guarantees. This lack of accuracy guarantee makes it hard for sensor networks to be incorporated into larger systems that require more reliability than e.g., typical environmental monitoring applications do. In this paper, we present a clock synchronization algorithm with deterministic accuracy guarantee. A key observation is that the variability of oscillation frequency is much smaller in a single crystal than between different crystals. Our algorithm leverages this to achieve much tighter accuracy guarantee compared to the interval-based synchronization methods mostly proposed in the literature of distributed systems. We designed an algorithm to solve a geometric problem involving tangents to convex polygons, and implemented that in TinyOS. Experimental results show the deterministic error bound less than 9.2 clock ticks (280 μsec) on average at the first hop, which is close to the simulation results. Further, by a combination with previously proposed synchronization algorithms, it achieves the estimation error of 1.54 ticks at 10 hop distance, which is more than 40% better than FTSP, while giving deterministic error bounds.

1 Introduction

Clock synchronization is a fundamental service that is required in many applications in sensor networks as well as in distributed systems in general. It is of more importance when sensor networks extend beyond research-oriented systems and get incorporated into industrial systems often referred to as cyber-physical systems (CPS). These systems are often mission-critical, and thus providing a performance guarantee is as important as yielding a good average performance. In the context of clock synchronization, performance guarantee corresponds to guarantees on accuracy.

Accuracy guarantee in clock synchronization, especially deterministic one, is important in sensor networks in several situations including sensor fusion, coordinated actions, and hard-realtime applications, as discussed in [9]. We add security applications to the list. For example in secure localization (e.g., [10]), in which multiple verifiers verify the location of a prover in a non-spoofable way, time difference of arrival (TDOA) of a signal from the prover is one of the key techniques for the verifiers to estimate the location. In this case, the accuracy of clock synchronization directly affects the reliability of the location estimation and thus the security level that the application can provide. However, the most important point is, without a guarantee on accuracy of clock synchronization, we cannot even quantify the reliability of such secure localization system.

Recent research on clock synchronization in wireless sensor networks has been mostly focused on improving the average case accuracy rather than the worst case. Although state-of-the-art techniques enable impressive average performance of few microseconds error [19, 20, 29], they either do not have any guarantees or only have probabilistic guarantees on the worst case accuracy.

In this paper, we propose a novel clock synchronization algorithm that gives a deterministic accuracy guarantee. In this algorithm each node gathers constraints based on the causality relation in message transmissions, and then calculates the upper and lower limits on the current time. such that all of these constraints are satisfied. Although the basic idea is analogous to classical interval-based clock synchronization [8, 9, 11, 18, 21, 24, 28], there are several key differences that enable our algorithm to achieve more correct and tighter bounds. One of the differences is the bounded drift fluctuation, which is given in the data sheet of crystal oscillators and we also experimentally verified by ourselves. The algorithm does not require or construct a fixed network topology and works completely in a distributed manner. It is also efficient by packing the information for multiple receivers into a single packet thus leveraging the broadcasting nature of wireless communication.

Our contribution is a novel clock synchronization algorithm that

- Gives deterministic accuracy guarantee without simplifying assumptions, such as constant drift,
- Is fully distributed and does not require any a priori knowledge on the network topology or the topology being stationary,
- Accommodates various settings including multiple root nodes, and
- Achieves good clock estimation when combined with other algorithms.

We implement the algorithm in TinyOS for testbed experiments and compare the results with simulation and FTSP.

The rest of the paper is organized as follows. In Section 2, we present the system model as well as the experimental results on temperature vs. clock frequency. We present main ideas of the synchronization algorithm in Section 3 and the details in Section 4. Section 5 discuss several design choices about message exchanges. In Section 6, we discuss some of the issues that arise in implementing the algorithm in TinyOS. Section 7 presents the evaluation results from both simulation and testbed experiments. We overview related work in Section 8 and Section 9 concludes the paper.

2 System Model

We first define the clock model and describe the assumptions. Then we validate these assumptions through preliminary experiments on the frequency vs. temperature characteristics of crystals.

2.1 Clocks

We refer to the clock reading at node v as *localtime* at v and denote as s^v. There are one or more nodes that have access to accurate time e.g., through GPS. These nodes are called *roots* and their time is called *globaltime* t, which is also called “wall-clock time” or “physical time” in the literature. All other nodes are just referred to as *nodes.* Node v's clock has a *clock drift* $h_v(s^v)$, which is defined as the amount of increase in globaltime when the localtime is increased by unit amount. For each node, we define *clock function* f_v to give corresponding globaltime for each localtime as follows:

$$f_v(s^v) = \int_0^{s^v} h_v(\tau)d\tau + \delta_v,$$

where δ_v is a constant called *clock offset.* As a notational convention, we assume globaltime t_i corresponds to localtime s_i^v; i.e., $t_i = f_v(s_i^v)$.

The drift consists of *drift offset* $\overline{h_v}$ and *drift fluctuation* $\alpha_v(s^v)$, where the former is a constant and the latter is a time-varying function:

$$h_v(s^v) = \overline{h_v} + \alpha_v(s^v).$$

While these are not known in advance, we assume they satisfy the following two properties:

- Bounded drift offset: $1 - \eta \leq \overline{h_v} \leq 1 + \eta$,
- Bounded drift fluctuation: $|\alpha_v(s^v)| \leq \xi$,

where η and ξ are given constants. The values of η and ξ depend on the type of crystal oscillator and usually specified in or can be calculated from the data sheet. In normal temperature range, ξ is much smaller than η. Note that we do not assume any statistical properties for $\alpha_v(s^v)$ except that it is bounded by ξ. Also note that η and ξ are different from "drift bound" (usually denoted by ρ) that often appears in the literature. Since $1 - \rho \leq h_v(s^v) \leq 1 + \rho$ by definition, we can consider that $\rho = \eta + \xi$. This is one of the key differences between the proposed algorithm and so-called interval-based methods, which we will discuss later in Section 3.4.

The objective of a synchronization algorithm is for each node to obtain a mapping from its localtime to the globaltime. We specifically focus on giving a deterministic guarantee on accuracy: for any localtime, each node must be able to tell the interval that the globaltime is contained within.

2.2 Crystal Oscillator

The assumptions of bounded drift offset and bounded drift fluctuation are not common and also very important for our algorithm. For these reasons, we have measured these values in the actual nodes to assess how reasonable these assumptions are. Since temperature is the primary cause that affects the clock frequency [30], we measure the frequency under different temperatures.

Crossbow Telos (Rev. B) nodes use Citizen CMR200T for 32.768kHz crystal oscillator [5]. This is a tuning fork crystal unit [2] and is known to have a quadratic relation between frequency and temperature. The relation is expressed by $f = f_0(1 + \beta(T - T_0)^2)$, where f_0 is the nominal frequency, T_0 is the reference temperature, and β is a constant called "temperature coefficient." In practice the actual frequency f_0' at T_0 is different from f_0 by a small amount and the bound on frequency tolerance $(f_0' - f_0)/f_0$ is usually specified in the data sheet. For CMR200T, $f_0 = 32768$ Hz, $\beta = -0.034 \pm 0.006$ ppm/℃^{2} (ppm: parts-per-million), and the frequency tolerance is ± 20 ppm [2]. The characteristic differs by types and also by parameters of the cut.

Figure 1 shows the relation of temperature and frequency measured at 19 Telos B nodes. Clock frequency is measured by counting the number of ticks precisely in 10 seconds, which is obtained from PPS (Pulse-per-Second) output of a GPS module (Garmin 18x LVC). This GPS module guarantees that the PPS signal is aligned to the start of each GPS second within 1 μsec [3]. We gradually change the environment temperature and associate the measured clock frequency with the temperature measured simultaneously by the onboard sensor.

All nodes exhibit curves peaked between 20 to 25 ℃ and their shapes are close to the one specified in the data sheet (not shown). As for the offset, the peaks are above the nominal frequency (32768 Hz) in 18 out of 19 nodes. This is likely to be due to the mismatch between ideal and actual load capacitance. Specifically, CMR200T requires load capacitance of 12.5 pF, whereas the MPU (MSP430F1611) has internal 12 pF fixed capacitance per pin [7] and they are added serially for two pins, resulting in 6 pF capacitance in total. Smaller load capacitance leads to higher oscillation frequency [6] and apparently, it is in accordance with the results. Unfortunately, since there are other sources of parasitic capacitance e.g., from PCB traces, we do not have a guarantee as strong as that for the frequency tolerance in the

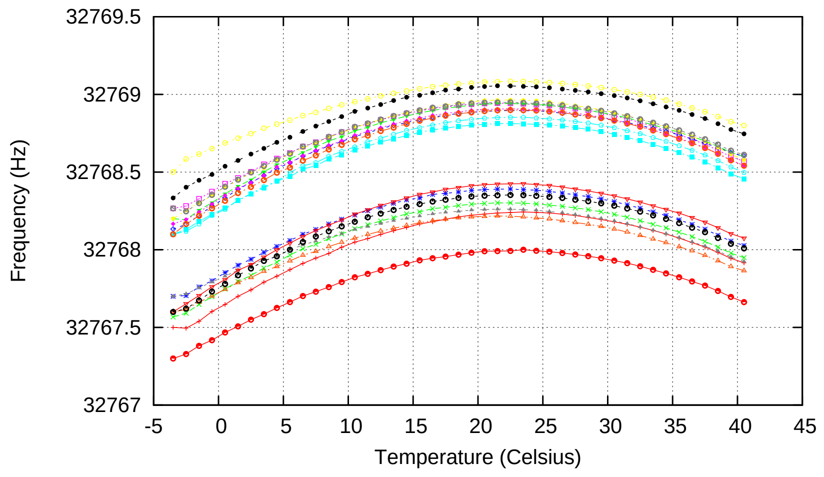


Figure 1: Temperature vs. frequency of 32.768 kHz crystal oscillator: For 19 nodes

crystal's datasheet. However, based on the observation, the curve still satisfies the frequency tolerance specification and we assume the frequency is only shifted by unknown small constant amount.

In summary, as we expected, we have observed larger variability among different crystals than in a single crystal at different temperatures. To accommodate the shift of peak frequency, we set the nominal frequency $f_0 = 32768.5$ [Hz]. In the later experiments, we assume the temperature range is 10 to 35 ℃. Then we can guarantee the frequency range for a single crystal is within 10 ppm (0.33 Hz in 32 kHz crystal) from both the specification and the measurement results. Since the peak frequency is within 20 ppm from the nominal frequency, we set the drift offset bound η to 25 ppm and the fluctuation bound ξ to 5 ppm to cover the whole range. If the temperature range is broader or unknown, these parameters must be chosen accordingly and conservatively to assure the correctness of the accuracy guarantee.

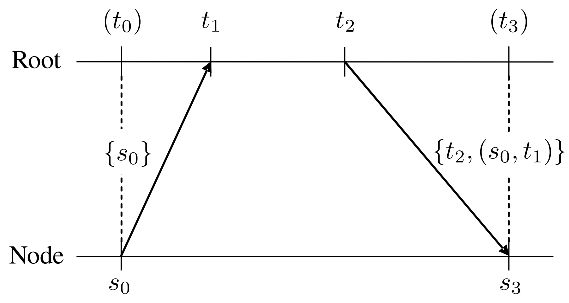
3 Synchronization with Accuracy Guarantee

In this section we describe the overall idea of our synchronization algorithm. For clarity we first assume constant drift (i.e., no drift fluctuation: $\xi = 0$) and explain the algorithm for synchronization between a root and a node. Then we extend it for synchronization between two nodes to enable network-wide synchronization, and also describe how we can take into account the drift fluctuation. Finally we discuss the differences between our algorithm and interval-based methods.

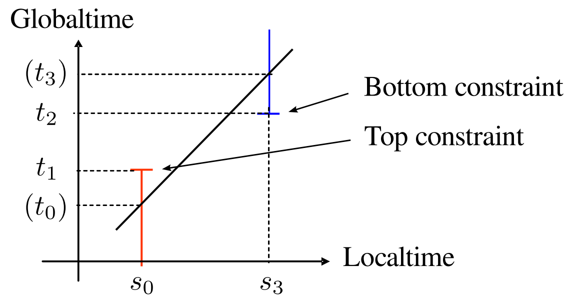
3.1 Main Idea

Our synchronization algorithm is based on a simple causality principle that a message is received only after it is sent. From sender and receiver timestamps, each node obtains a set of constraints that its clock function must satisfy.

Figure 2(a) shows the message exchange between a root and a node. The node sends a message at localtime s_0 and the root receives it at globaltime t_1. Then the root sends back a message at t_2 with the information on previously received message (s_0, t_1) as well as with the timestamp t_2. Upon receiving the message, the node can do the following calculations. For the first message, since the receiving time is later than the sending time, $t_0 \leq t_1$, though we cannot know t_0. Using $f(s_0) = t_0$,



(a) Message exchange: t_0, t_3 are not known.



(b) Constraints on clock function

Figure 2: Basic idea for synchronization between a root and a node

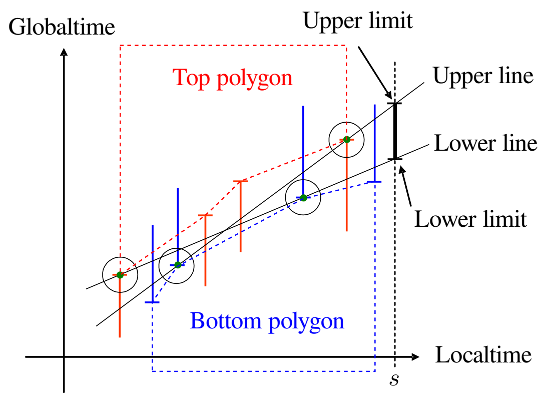


Figure 3: Upper/lower limit given by upper-/lowermost line satisfying all constraints.

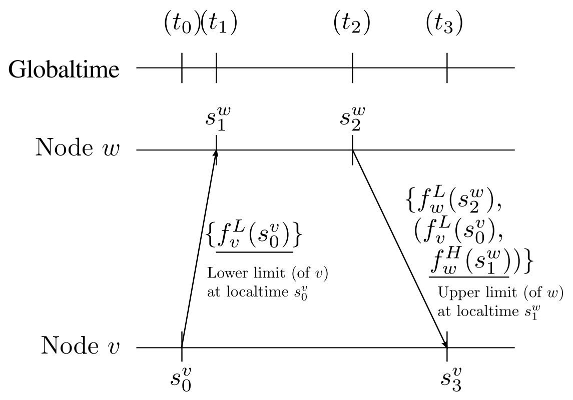

$(t_0)(t_1)$
(t_2)
(t_3)
Globaltime
Node w
s_1^w
s_2^w
$\{f_v^L(s_0^v)\}$
Lower limit (of v) at localtime s_0^v
$\{f_w^L(s_2^w), (f_v^L(s_0^v), f_w^H(s_1^w))\}$
Upper limit (of w) at localtime s_1^w
Node v
s_0^v
s_3^v

Figure 4: Node-node message exchange: $t_0, \ldots, t_3$ are not known.

where f is the clock function for the node, we have $f(s_0) \leq t_1$. Similarly for the second message, we obtain $f(s_3) \geq t_2$.

The constraints that give upper or lower bounds for a clock function are called *top constraints* and *bottom constraints*, respectively (Figure 2(b)). A constraint C_i is expressed by $(s_i, l_i, type_i)$, where s_i is localtime, l_i is called the *value* of C_i, and $type_i$ is either “top” or “bottom.”

After several messages are exchanged, each node has a set of top constraints and bottom constraints (Figure 3). When we ignore the drift fluctuation (this is relaxed later in the section), f is a linear function that satisfies all the constraints. Of all such linear functions, we can determine the ones that give the maximum and minimum globaltime at given localtime s. We call these *upper line* and *lower line*, which are collectively called *limiting lines*. The maximum and minimum globaltime are called *upper limit* and *lower limit* at localtime s, respectively. Further, we call a constraint a *support* or a *support constraint* when it determines the upper or lower line.

The problem of finding limiting lines can be viewed in a more geometric way by considering two polygonal objects for each of the sets of top and bottom constraints. The clock function is a long bar and the upper and lower limits correspond to the range of motion when it is inserted between two polygons.

3.2 Network-wide Synchronization

The algorithm for synchronization between a root and a node can be extended for network-wide synchronization. We describe the case for two nodes where neither of them is a root. We use f_v^H

and f_v^L to denote the upper and lower limits at node v. For any localtime s^v, they satisfy $f_v^L(s^v) \leq f_v(s^v) \leq f_v^H(s^v)$.

Figure 4 shows the message exchange for synchronization between two nodes. Suppose node v initiates the message exchange. Different from the root vs. node case, at localtime s_0^v, v sends the lower limit $f_v^L(s_0^v)$ instead of s_0^v itself, which node v remembers for later use. Node w records the localtime s_1^w when it receives the message, but it remembers the upper limit $f_w^H(s_1^w)$ instead. Then in the response message, node w puts the pair $(f_v^L(s_0^v), f_w^H(s_1^w))$ as well as the lower limit $f_w^L(s_2^w)$, just as node v did. This pair of (lower limit at sent time, upper limit at received time) is called *SyncInfo* for the original sender.

After this message exchange, node v obtains the following two constraints:

$$\text{top: } f_v(s_0^v) = t_0 < t_1 = f_w(s_1^w) \leq f_w^H(s_1^w) \tag{1}$$

$$\text{bottom: } f_v(s_3^v) = t_3 > t_2 = f_w(s_2^w) \geq f_w^L(s_2^w) \tag{2}$$

Note that $t_0, ..., t_3, f_w(s_1^w), f_w(s_2^w)$ are all unknown.

As a side-effect, node w also obtains a bottom constraint $f_w(s_1^w) > f_v^L(s_0^v)$ from the first message. This is a preferable property especially for wireless environment where every communication is essentially a broadcast. For network-wide synchronization, multiple receivers within the communication range of a sender can obtain a bottom constraint. We can also embed multiple SyncInfo in one message so that multiple receivers can obtain top constraints simultaneously.

3.3 Compensation for Drift Fluctuation

So far we have assumed that clock drift is constant. However, in practice, clock drift fluctuates over time with external causes, mostly due to temperature changes. Here we extend the algorithm for the case with drift fluctuations.

The idea is to compensate each of the constraints for the effect of drift fluctuation after the constraint is obtained. For constraint $C_i = (s_i, l_i, type_i)$, we define *compensated constraint* $\tilde{C}_i(s) = (s_i, \tilde{l}_i(s), type_i)$ at localtime s, where compensated value $\tilde{l}_i(s)$ is defined as follows:

$$\tilde{l}_i(s) = \begin{cases} l_i + \xi(s - s_i) & \text{if } type_i = \text{“top”} \\ l_i - \xi(s - s_i) & \text{if } type_i = \text{“bottom”} \end{cases}$$

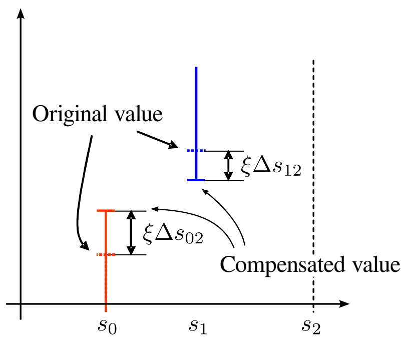
Then we have the following lemma:

Lemma 1. *Given $f(s)$ that satisfies all the constraints and $f(s_1) = t_1$, linear function $g(s) = \overline{h}s + \delta$ with $g(s_1) = t_1$ satisfies all the compensated constraints, where $\overline{h}, \delta$ is the drift offset and clock offset, respectively.*

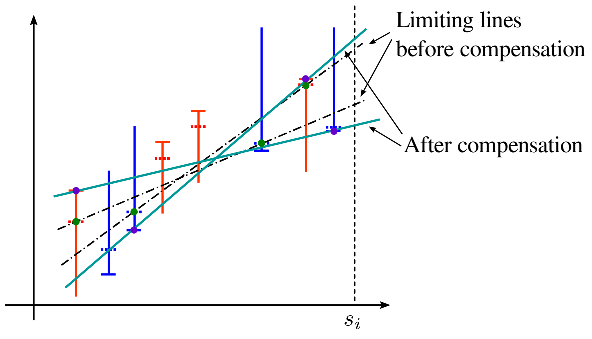
Proof is in the Appendix. Then we have the following theorem:

Theorem 1. *$\forall s. g^L(s) \leq f^L(s) \leq f^H(s) \leq g^H(s)$, where $g^L(s), g^H(s)$ are the lower and upper limits at localtime s calculated for linear clock function $g(s)$ with the slope in $[1-\eta, 1+\eta]$ that satisfies all the compensated constraints.*

Figure 5 explains compensated constraints. As Figure 5(a) shows, at localtime s_2, the value of the top constraint at s_0 is increased by $\xi \Delta s_{02}$, and that of the bottom constraint at s_1 is decreased by $\xi \Delta s_{12}$. Then, as shown in Figure 5(b), we change the value for all constraints in the same way and the limiting lines are determined for these compensated constraints. Since the compensated constraints are “looser” than the original ones, according to Theorem 1, the new interval of upper and lower limits contains the one calculated without compensation. In this way, after compensation, we only need to solve the same problem of finding limiting lines.



(a) Compensation at localtime s_2



(b) Limiting lines at localtime s_i

Figure 5: Compensated constraints

3.4 Comparison with Interval-based Methods

Our algorithm shares the idea with the clock synchronization algorithms that provide an interval that the globaltime is contained. These algorithms, often called "interval-based algorithms," are proposed mostly in the literature of distributed systems before sensor networks emerged, but there are some papers in the contexts of mobile ad-hoc networks [24] and sensor networks [9] as well. Here we compare the proposed algorithm with these works.

While it is often claimed that an algorithm guarantees the accuracy, the correctness may not hold in the context of sensor networks. For example, some of the interval-based algorithms just ignore transmission delay (e.g., [9, 21]). This is not an appropriate assumption when the transmission delay is dominant in the resulting accuracy. More often clock drift is assumed to be constant [8, 11, 31]. This is not appropriate as well for sensor networks that are often deployed outside and exposed to drastic temperature changes. Our algorithm does not make these assumptions and only uses the bounded drift offset and bounded drift fluctuation assumptions, both of which are guaranteed to be correct and derived from the data sheet of crystal oscillator.

Meanwhile, the common underlying idea for interval-based synchronization methods is to bound the actual length of any time interval by using most pessimistic value of clock drift. For example, when clock drift bound is ρ, we can guarantee that the actual length of localtime interval Δs is bounded by $[(1-\rho)\Delta s, (1+\rho)\Delta s]$. In our settings, we can emulate this by setting drift offset bound $\eta = 0$ and drift fluctuation bound $\xi = \rho$. This is because their assumption is precisely equivalent to assuming that each crystal can fluctuate in the whole range of $[1-\rho, 1+\rho]$. In the experiments, we will use this to compare our algorithm with the interval-based methods.

4 Algorithm Details

In this section we describe the algorithm in more detail. Receive and Send are the procedures called upon message reception and transmission, respectively. Upon receiving a new message, constraints are added using AddConstraints and Convexify, and limiting lines are calculated using UpdateUpperLine/UpdateLowerLine. As we see later, all these procedures can be done in the time logarithmic to the number of constraints stored in a node.

Algorithm 1 Procedures for communication between nodes

```
procedure RECEIVE($f_w^L, \{S\}$)    ▷ at localtime $s_{recv}^v$; $\{S\}$: Set of SyncInfo in received message
    UPDATEUPPERLINE
    UPDATELOWERLINE
    ADDCONSTRAINT($s_{recv}^v, f_w^L, bottom$)
    $f_v^H \leftarrow$ CALCUPPERLIMIT
    if $f_v^H \neq \infty$ then SAVESYNCINFO($src = w, f_w^L, f_v^H$)
    for $S_i \in \{S\}$ do
        if $S_i$ is for this node then
            $s_{send}^v \leftarrow$ RETRIEVESENDTIME($S_i$)
            ADDCONSTRAINT($s_{send}^v, value(S_i), top$)
    if new constraint became a support then SEND

procedure SEND    ▷ at localtime $s_{send}^v$
    UPDATELOWERLINE
    $f_v^L \leftarrow$ CALCLOWERLIMIT($s_{send}^v$)
    STORESENDTIME($s_{send}^v, f_v^L$)
    SENDMESSAGE($f_v^L, \{S\}$)    ▷ $\{S\}$: Set of SyncInfo
```

4.1 Communications

There are several design choices on communication between nodes, which we discuss in depth in the next section. Here we describe the method we have implemented. Pseudocode for the procedures is shown in Algorithm 1.

A root broadcasts a message periodically. The period is to be determined based on application requirements and energy availability. In the experiments we use uniformly random period in $18 - 22$ seconds.

Upon receiving a new message, a node updates upper and lower lines. This is necessary because, due to the compensation of drift fluctuation, the limiting lines will change even without any changes in the sets of constraints.

After that, the node adds a bottom constraint based on the received time and the lower limit that the message carries. The same information is saved as SyncInfo, which will be sent back to the sender. Then the node scans through all SyncInfo that the message carries. If there is one for this node, it adds a top constraint based on that.

Shortly after receiving, a node sends a message when one of the new constraints has become a support. We use this "pulse-like" propagation pattern based on the analysis in [19]. Upon sending a message, the node first updates the lower line to calculate the current lower limit.

4.2 Computations

4.2.1 Add Constraint

After adding a new constraint, we call CONVEXIFY to eliminate any non-convex constraints, since they will never become supports, as seen from Figure 3. Then, if the new constraint violates the current limiting line, we update the limiting line.

Algorithm 2 Procedures for computation at each node

```
procedure AddConstraint($C_{new}$)
    if $C_{new}$ is a top constraint then
        $\mathcal{C}_{TOP} \leftarrow \mathcal{C}_{TOP} \cup C_{new}$    ▷ $\mathcal{C}_{TOP}$: set of top constraints, sorted by localtime
        Convexify($C_{new}$)
        if $C_{new} \in \mathcal{C}_{TOP} \wedge C_{new}$ violates upper line then UpdateUpperLine
    else    ▷ $C_{new}$ is Bottom
        $\mathcal{C}_{BOT} \leftarrow \mathcal{C}_{BOT} \cup C_{new}$    ▷ $\mathcal{C}_{BOT}$: set of bottom constraints, sorted by localtime
        Convexify($C_{new}$)
        if $C_{new} \in \mathcal{C}_{BOT} \wedge C_{new}$ violates lower line then UpdateLowerLine

procedure Convexify($C_{new}$)
    Find leftmost $C_i$ s.t. $\{C_i, ..., C_{new}\}$ is non-convex
    Remove non-convex constraints between $C_i, C_{new}$
    Find rightmost $C_j$ s.t. $\{C_{new}, ..., C_j\}$ is non-convex
    Remove non-convex constraints between $C_{new}, C_j$

procedure UpdateUpperLine
    $l \leftarrow$ FindTangent
    if Slope($l$)$> 1 + \eta$ then $l \leftarrow$ FindTopSupport($1 + \eta$)

procedure UpdateLowerLine
    $l \leftarrow$ FindTangent
    if Slope($l$)$< 1 - \eta$ then $l \leftarrow$ FindBottomSupport($1 - \eta$)
```

4.2.2 Convexify

Convexify is the procedure for making the set of constraints form a convex polygon by eliminating non-convex constraints. Finding non-convex constraints is easily done by calculating the turning direction of two vectors (typically by using the external product) made by three points. This is similar to what Graham scan [14] does for calculating the convex hull of points. However, since in our case the points are already sorted and we know the set is convex without the new point, we can use binary search to find a convex point instead of linearly scanning the points. This makes the running time $O(\log n)$, when n is the number of top/bottom constraints stored in the node.

One thing to note is that, in case of top constraints, a newly added constraint may not be the latest constraint in terms of localtime. Therefore, we may need to run the above binary search for both directions from the inserted point, though it does not affect the order of the computation time.

4.2.3 Update Limiting Lines

The core of the procedures for updating upper and lower lines is FindTangent. In FindTangent we find a tangent of two convex polygons. Note that polygons are convex whenever FindTangent is called. A naive method mentioned in [8] is to try all possible pairs of vertices and check if the line intersects with the polygons. This takes $O(n^2)$ time in the worst case, but can be improved to $O(\log^2 n)$ by using nested binary search. Further, by using more sophisticated algorithms, it is improved to $O(\log n)$ time [15, 23].

When the slope of the calculated limiting line is out of range of $[1 - \eta, 1 + \eta]$, we can replace the line with the slope $(1 + \eta)$ (for upper line) or $(1 - \eta)$ (for lower line). Figure 6 explains this for a case

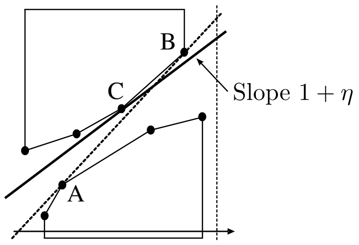


Figure 6: When the slope of a tangent is out of range $[1 - \eta, 1 + \eta]$, we use a limiting line with one support.

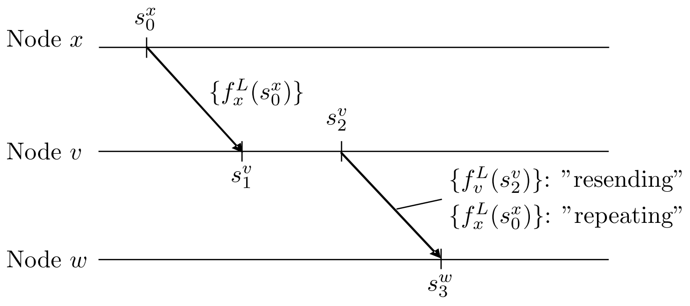


Figure 7: Two strategies for three-node communication

of upper line. The tangent for the top and bottom polygons is line AB (dotted line), but the slope is more than $(1 + \eta)$. In this case, the upper limit is determined not by this line but by the line with slope $(1 + \eta)$ that touches the top polygon (solid line). As in the figure, since the support for this line (C) may be different from the support for the tangent (B), we need to find it again. In this case the line is supported by one point instead of two. Searching the support is done efficiently by calculating the slope of each edge of the polygon, and since it is convex, the slope is monotonically increasing (for top polygon) or decreasing (for bottom polygon). By using binary search, it takes $O(\log n)$ time.

5 Message Exchange

Message exchange is the core of the synchronization algorithm and we have several design choices to make. These choices have a large impact on the efficiency of the algorithm as well as the performance. Here we discuss two major aspects about it: what and when to exchange messages.

5.1 What to Transmit

In Section 3 we have described what to send in each message for synchronization between root and node, and also between two nodes. In case of three or more nodes, however, there are multiple choices on what to transmit in a message. Namely, a node can calculate its own lower limit and send it, or alternatively, it can just repeat the message it received. So far we have implicitly considered only the former "resending" strategy, but now we assess the benefit of the latter "repeating" strategy.

Figure 7 shows two strategies for the case of bottom constraint. After receiving the lower limit $f_x^L(s_0^x)$, node v either calculates the lower limit and sends $f_v^L(s_2^v)$ or just repeats $f_x^L(s_0^x)$. About this case, we can show the following:

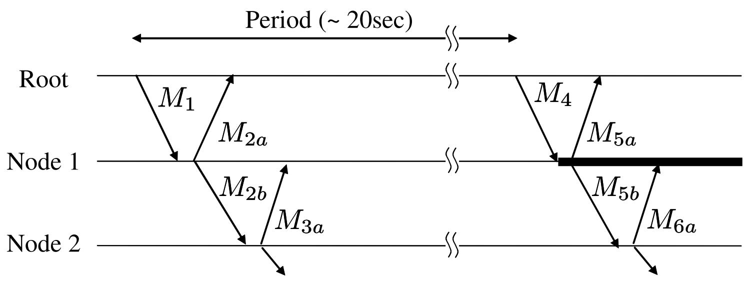


Figure 8: Message transmissions in normal mode (in line topology): Upon receiving a message that produces a support constraint, a node forwards a message.

Claim 1. $f_v^L(s_2^v) \geq f_x^L(s_0^x)$.

The proof is in the Appendix. This implies that, in order to get a tighter bottom constraint, we should always take the resending strategy.

As for top constraints, similar argument does not hold and there is a case that repeating produces a better top constraint than resending. However, repeating top constraints makes the algorithm more complicated because we need to do routing of each SyncInfo, whose recipient may not be in the direct communication range. From these considerations, we just take the resending strategy and do message exchanges among single-hop neighbors.

5.2 When to Transmit

As for the timing of transmission, we have three modes: normal mode, fast start mode, and request mode.

5.2.1 Normal mode

A root sends a message periodically. The period is to be determined based on application requirements and energy availability. In the experiments we use uniformly random period in $18 - 22$ seconds.

Upon receiving a message from a root, a node needs to forward it so that the information propagates throughout the network. There are two design questions here: whether to send it and when to send it. For the first question, a simple strategy is to send whenever a node receives a new message. However, this strategy will cause a situation similar to broadcast storm [22]. Instead, we use the fact that a node cannot provide essentially no new information unless it added a new constraint that became a support. Therefore, it sends a message after receiving a message that has provided a support constraint. In this way each node does not use any information on network topology.

For the second question about the timing, we make the nodes send a message as soon as possible. This is based on the results of Lenzen et al. [19], where they showed that it is more beneficial to forward information as quickly as possible than each node forwards on its own timing as in FTSP [20]. This applies to our algorithm as well and we follow the same principle.

Figure 8 summarizes the behavior in normal mode in case of a line topology. The root sends one message in every period. A message from the root most likely provides a support constraint to the receiver (node 1). Thus node 1 forwards the message and the message propagates toward the edge of the network in the same way.

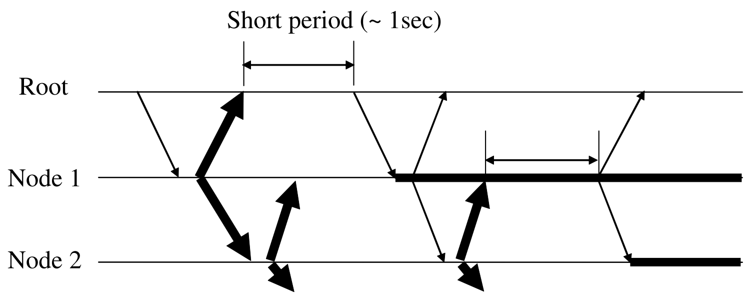


Figure 9: Fast start: Bold arrows represent messages with REQ flag. Horizontal bold line means the node has at least one top constraint (i.e., has bounded upper limit).

5.2.2 Fast start mode

A downside of the normal mode is that the propagation can be slow at first. This becomes a problem especially when all nodes are started around same time, for example in an emergency response application after the nodes are deployed at the site.

We explain this situation using Figure 8. Let's assume all nodes do not have any constraints yet. Now, the root sends a message (M_1). Since node 1 gets a bottom constraint from this and that becomes a support for lower line[1], it sends a message (M_{2a}, M_{2b}). Node 2 does the same and sends M_{3a}. When node 1 received M_{3a}, however, it does not have any top constraints yet. Thus it cannot calculate the upper limit at the time of receiving M_{3a} and does not store the SyncInfo for it[2]. In the next period, the root sends M_4 that contains SyncInfo for node 1 based on M_{2a}. Now, it can calculate the upper limit and thus sends a message (M_{5a}, M_{5b}). Node 2 does the same and send M_{6a} and finally, the SyncInfo for node 2 is saved upon receiving it. In this way, it takes one period until the nodes in the next hop obtains its first top constraint.

To resolve this problem, we use fast start (Figure 9). When a node receives a message, it sends a message with a special flag if it has no top constraint and the upper limit is unbounded. This flag, which we call "REQ" flag, is a request for neighbors to respond so that this node can get a top constraint.

Upon receiving a message with "REQ" flag, all nodes having a top constraint start a timer for a short period ($\sim$ 1 sec). When the timer fires, it sends a message that contains a SyncInfo for the requester. The reason for this short delay is to aggregate responses to multiple requests. This rule overrides the forwarding decision in the normal mode and a node will send a message even if there is no change on its supports. Also, to avoid a broadcasting storm, nodes do not send another request upon receiving a request from other nodes.

In this way, initial propagation becomes much faster. This comes in exchange for more message transmissions. In the line topology, a node at distance d will send and receive d more messages than in the case without fast start. Instead, it will get its first top constraint roughly in d seconds after initialization, instead of $20d$ seconds.

[1] We can determine a lower line without a top constraint using bounded drift offset assumption. See Section 4.2.3 for details.

[2] It is possible for node 1 to keep the information until it gets a top constraint and then calculate the upper limit retrospectively, but we do not do this in the current implementation.

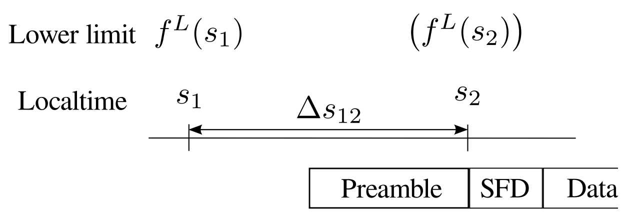


Figure 10: Delay compensation: Lower limit $f^L(s_1)$ is calculated at localtime s_1, but actual send time is s_2. Calculate the lower limit $f^L(s_2)$ and put it in the packet is not feasible because of timing. Instead, we put $f^L(s_1)$ and $\Delta s_{12} = s_2 - s_1$ so that a receiver can calculate the delay-compensated lower bound $\tilde{f}^L(s_2)$.

5.2.3 Request mode

When a node has not received a message for a long time, it sends a message to request the neighbors to provide information. This is done using the same REQ flag as in the fast start. In the experiments we set the period to 30 seconds.

6 Implementation

We have implemented the proposed synchronization algorithm in TinyOS 2.1.1. Besides dealing with limitations in TinyOS programming, we needed to address several fundamental issues specific to our synchronization algorithm.

6.1 Delay Compensation

Reducing the transmission delay between sender and receiver is the key to achieving precise synchronization. This also applies to our algorithm, as the minimum possible gap between upper and lower limits is at least twice as big as the minimum transmission delay (proof omitted).

To minimize the transmission delay, we use MAC-layer timestamping, which is to put a timestamp when the packet is actually transmitted over radio. In case of CC2420 radio chip, the timestamp for a packet is obtained when SFD (start frame delimiter) is captured. Since this is same for both sender and receiver, two timestamps are expected to be obtained at very close time points.

A problem in MAC-layer timestamping is the delay after a node calculates the lower limit until the packet is transmitted. A receiver can get a tighter bottom constraint if the lower limit is calculated at the time when the packet is transmitted. However, it is not feasible to recalculate it at MAC layer.

A solution for this problem is to reconstruct the lower limit at the receiver side. Figure 10 explains this. At localtime s_1, the sender calculates the lower limit $f^L(s_1)$, puts it in the packet, and issues the send command. In the timestamp field, which will be rewritten at the MAC-layer, the sender put s_1, the localtime when it calculated the lower limit. At the MAC-layer this field is rewritten with the difference with value in the field and the current timestamp s_2. When the packet is received, the receiver carries $f^L(s_1)$ and $\Delta s_{12} = s_2 - s_1$. Then it calculates the delay-compensated lower bound $\tilde{f}^L(s_2)$ as follows:

$$\tilde{f}^L(s_2) = f^L(s_1) + (1 - 3\eta - \xi)\Delta s_{12}.$$

This is based on the following theorem. Proof is in the Appendix.

Theorem 2. *$f^L(s_2) \geq f^L(s_1) + (1 - 3\eta - \xi)\Delta s_{12}$, where $f^L(s_2)$ is actual lower limit at localtime s_2.*

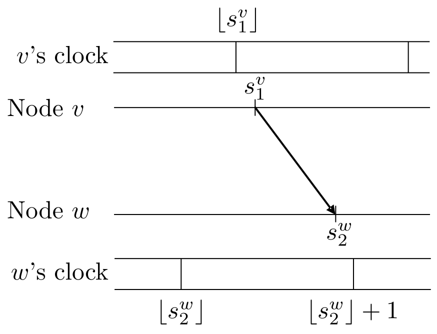


Figure 11: Order-preserved timestamping: Sender timestamp s_1^v is taken before receiver timestamp s_2^w, but the causality may be violated after quantization.

6.2 Quantization Error

In MSP430 microprocessor used in Telos B nodes, unsigned 32 bit is the largest size of integer. As floating point computation is emulated on software and has low precision, we primarily use integer for accurate computation. Related to this quantization, there are several issues that require a special attention for the algorithm to work correctly.

One issue is about the order of timestamping. We strictly require that the timestamp at the sender is taken before that at the receiver. However, when we consider integer timestamps, this is not always satisfied. Figure 11 shows an example. This order violation will not happen if the receiver timestamp is always taken more than one clock tick after the sender's one, but unfortunately in our case, the difference is much smaller than one tick. To avoid this situation, we add one clock tick to the receiver's timestamp.

The other is about compensated constraints. Even though the original value of each constraint is an integer, it will become a noninteger when we calculate the compensated value. For not to violate the compensated constraints, we need to be conservative on quantization. Specifically, we round up the value for a top constraint and round down the value for a bottom constraint. Because of this, there are some cases that message transmission loops in two nodes without receiving any new messages. To avoid this, a node does not send a message if it has sent the previous one within a short ($\sim$ 1 sec) period.

6.3 Parameters

6.3.1 Number of Constraints

Although we eliminate non-convex constraints that do not contribute to limiting lines, the number of constraints is still unlimited. Since this is not preferable, we limit the number to a constant. In the experiment, we set the maximum to 5 for each of top and bottom constraints.

When the set of constraints is full, we evict one of the constraints before adding a new one. Note that this eviction does not affect the correctness of the limits in any ways; it only results in looser accuracy guarantees if we evict a "tight" constraint. To minimize the impact, we evict the newest constraint that is not a support.

Table 1: Message structure: Each message has 23 Bytes of data. Sender ID (2 Byte) is in the header.

Name		Size (B)	Comments
seqnum		1	Sender's sequence number
lowerlimit		4	Lower limit
delta		4	Delay until transmission
SyncInfo		7×2	
	nodeID	2	Recipient node
	upperlimit	4	Upper limit at received time
	seqnum	1	Recipient's sequence number

6.3.2 Message Size

By default, the maximum data size in each packet is 28 bytes in Telos B. With this limited size, it requires a good way to include SyncInfo for multiple nodes in a packet as well as sender's timestamp.

Table 1 shows the message structure. A key idea is to use 1-byte sequence number to reduce the size of SyncInfo, instead of sending back 4-byte lower limit. Sender keeps track of the association between the sequence number and the timestamp. We use a ring buffer of size 256 for storing timestamp.

6.3.3 Number of SyncInfo

In each message up to two SyncInfo are included. In the current implementation, each node stores up to 10 SyncInfo and choose two randomly upon sending.

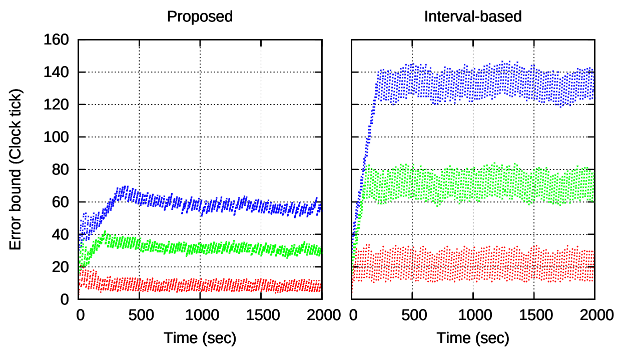
7 Evaluations

In this section we evaluate the proposed synchronization algorithm by both simulation experiment and testbed experiment. For the performance metric, we use the error bound, which is calculated from the upper and lower limits by $(f^H(s) - f^L(s))/2$. This is the minimum error bound, which is achieved by using the average of upper and lower limits as the estimate.

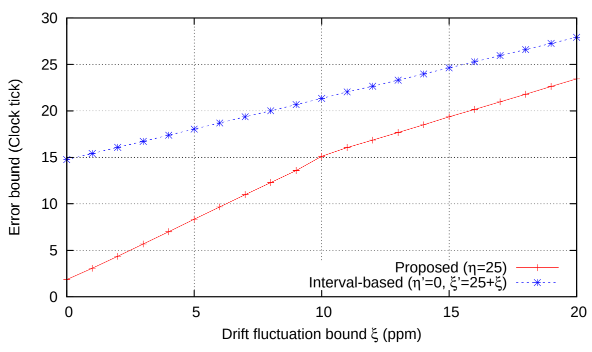
7.1 Simulation Experiments

To see the performance in various networks with different sizes and different topologies, we first implemented the synchronization algorithm in Java and built a discrete event simulator. On initialization, each node is assigned a constant drift offset randomly chosen in the range $[1-\eta, 1+\eta]$, where $\eta = 25$ (ppm). The clock drift actually does not fluctuates, but we set the drift fluctuation bound $\xi = 5$ (ppm) for taking into account the compensation. Connectivity graph among nodes is given. Packet reception rate is set to 95% and there is a random transmission delay determined from given range.

For simulation, we estimate the transmission delay as follows. The delay from sender's SFD to receiver's SFD in CC2420 is typically 3 μsec [1], and more precise measurement shows it is 3.16 μsec on average with the standard deviation around 41 ns [27]. Larger delay comes from quantization: we add one tick (30.52 μsec) to the receiver's timestamp to guarantee the order (see Section 6.2). Assuming that the phase is random, the average delay is expected to be a half tick (15.26 μsec) with a half tick range. Adding these together, we estimate the delay to be in $[3.16, 33.68]$ μsec.



(a) Proposed vs. Interval-based method



(b) Average error bound at first hop

Figure 12: Comparison between proposed and interval-based methods

7.1.1 Comparison with interval-based method

First we compare the proposed algorithm and interval-based methods. We use a topology of one root at the end and 10 nodes connected in line. Figure 12(a) compares the error bounds of three nodes (at 1st, 5th, and 10th hop from the root) for both algorithms. For all three nodes, the proposed algorithm achieves much smaller error bounds than the interval-based method. This suggests the information on drift fluctuation bound can help improving the accuracy in clock synchronization.

To see this effect in more quantitative way, we change the drift fluctuation bound ξ and see the error bound. For the proposed method, the drift offset bound η is kept constant at 25ppm, and we emulate an interval-based method by setting $\eta' = 0$ and $\xi' = \eta + \xi$ based on the discussion in Section 3.4. Figure 12(b) shows the results for the first hop. The results show that the proposed method can achieve smaller error bound for all cases and the improvement is bigger for the smaller ξ.

7.1.2 Grid topology

Figure 13 shows the error bounds for 25 nodes in a grid topology of 5×5. The root is on one of the corners. An interesting finding is that the error bound is not very proportional to the hop distance (Manhattan distance in this case) from the root, especially for the nodes far from the root.

This is likely to be related to the number of paths from the root. For example, node A and B are both at two hop distance from the root, but node B has smaller error bound. This is partly because node B has two separate shortest paths from the root, while node A has only one. Since the delay is randomly distributed over a large range, having multiple paths improves the chance of getting good constraints that are close to the actual globaltime. This effect is not clear among the distant nodes, all of which have multiple paths.

7.1.3 Dynamic topology

Figure 14 shows the error bound when the network topology changes dynamically. Specifically we tested the case when a node sleeps (i.e., becomes inactive) for some time interval.

When there is one root in a line topology of 10 nodes (Figure 14(a)), the error bound increases linearly while node 2 is sleeping. This is reasonable because, without node 2, there is no path from the root and thus no "essentially new" information comes to node 3. Once node 2 becomes active again, the error bound quickly approaches the original range.

Figure 14(b) shows the case for two roots at both ends of a line topology of 10 nodes. In this case, while node 2 is unavailable, node 3 loses the path to the root on the left, but is still connected to the

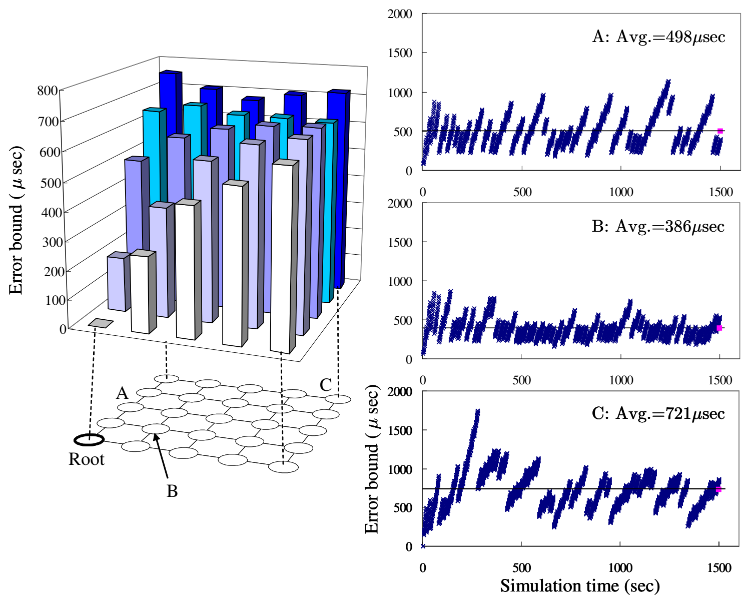


Figure 13: Error bound in grid topology: 5 × 5 nodes

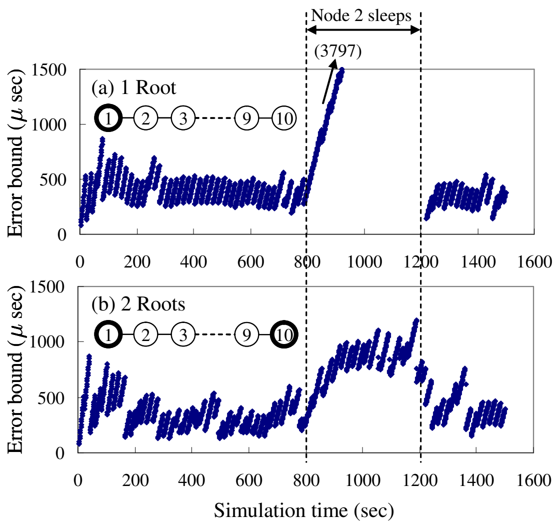


Figure 14: Error bound (at Node 3) in case of dynamic topology change: Node 2 sleeps in [800, 1200] sec. (a) One root case; (b) Two roots case.

root on the right. Therefore, it can receive new information. However, as the hop distance is larger then before, the error bound distributes in the higher range. Similarly as the one root case, it comes back to the original range after node 2 is back.

7.2 Testbed Experiments

We programmed 11 Telos B nodes; one for root and other 10 for nodes. In the software we made a line topology with a root at one end. The UserINT ports of the root and all nodes are connected to a

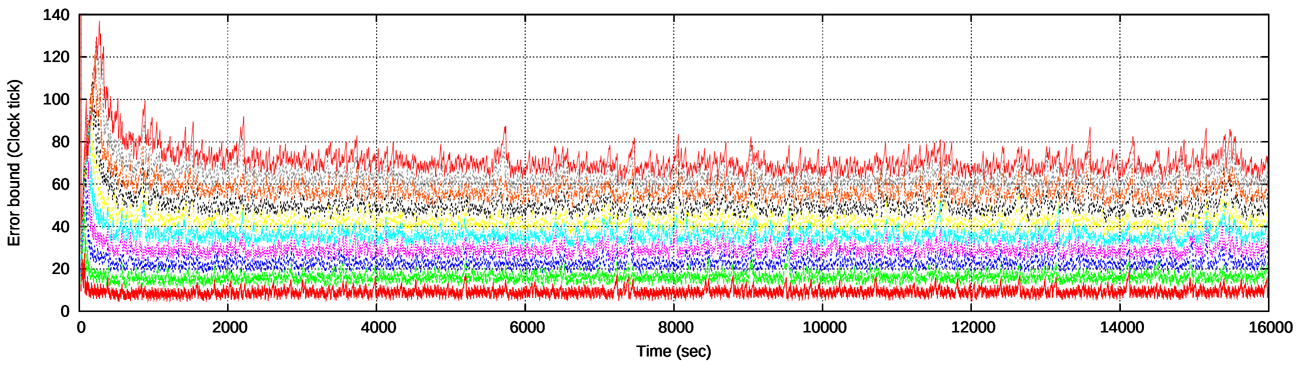
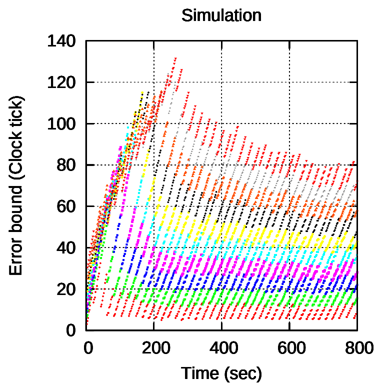
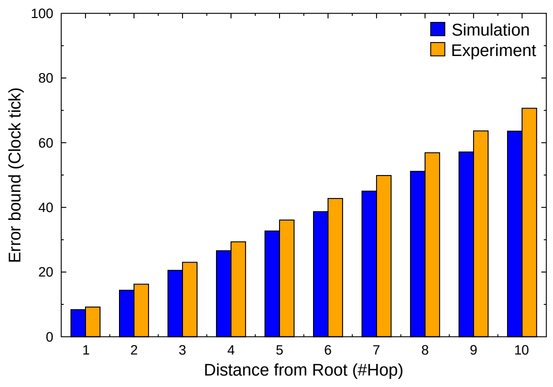


Figure 15: Testbed experiment: Error bounds of 10 nodes in a line topology



(a) First 800 sec



(b) Average after 800 sec

Figure 16: Comparison between simulation and testbed experiment

single trigger source that emits a trigger every two seconds. Upon receiving a trigger, the root node sends the current time and all other nodes send the upper and lower limits at that time to the PC. We use serial communication via USB port for the communication with the PC to avoid congesting the radio channel.

7.2.1 Comparison with Simulation Results

Figure 15 shows the error bound of all 10 nodes (except the root). Figure 16 shows the comparison between measured and simulated results. The change of error bounds closely matches the simulation result both in the transient state at first (Figure 16(a)) and in the steady state after long time (Figure 16(b)), though the average error bound in the experiment is a little (9-13%) higher than in the simulation. The error bound at the first hop is 9.2 ticks, which corresponds to 280.0 μsecs at 32768.5Hz clock, and it is roughly proportional to the hop distance from the root.

7.2.2 Improved Clock Estimation and Comparison with FTSP

As an extension we have incorporated the ideas from previous work to improve the clock estimation performance. We slightly change the proposed algorithm and add the current estimate of globaltime in each message, in addition to the lower bound of it. For estimation, we use a common method used in e.g.,[17, 19]: Each node considers the time lapse between receiving and sending, makes a table of (localtime, globaltime) pair, and uses linear regression to obtain an estimate.

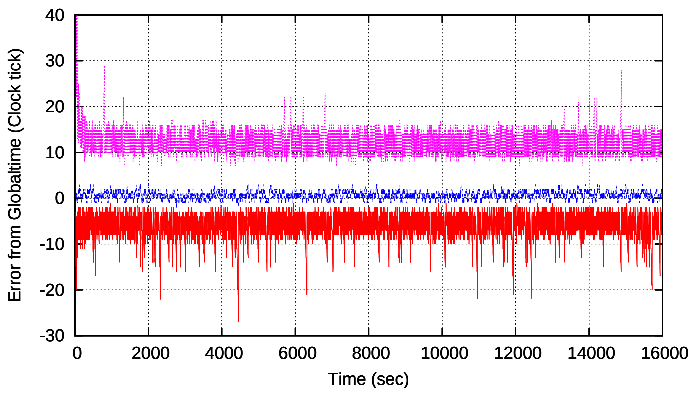


Figure 17: Upper/lower limits and estimate at the first hop

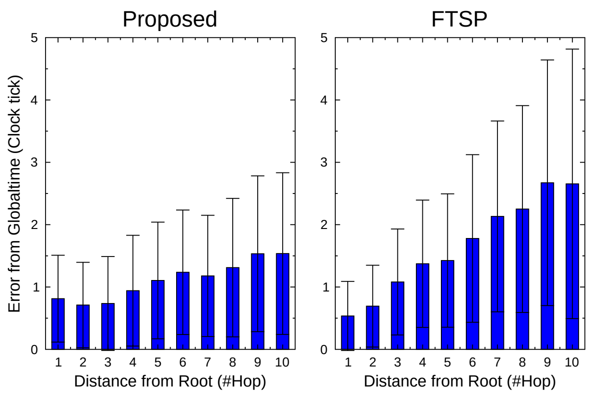


Figure 18: Comparison with FTSP: Average estimation error

Figure 17 shows upper/lower limits and estimate at the first hop in the 10-node line topology. The vertical axis is the error from globaltime. Since the upper and lower limits are deterministic bounds, the error for upper limit is always positive and that for lower limit is negative. The estimate distributes around zero. Figure 18 shows the average estimation error for each of 10 hops line topology compared between the proposed algorithm and FTSP. It is observed that the average error for the proposed algorithm is comparable to that for FTSP in the near distance and up to 40% better in the far (At 10th hop, 1.54 vs. 2.65 ticks; 42% better).

8 Related Work

Our synchronization algorithm is most closely related to interval-based methods, as we have discussed in Section 3.4. Marzullo and Owicki [21] studied the synchronization problem where each time server returns an interval that contains the true time and the objective is to minimize the length of interval by exchanging messages among multiple time servers. Blum et al. [9] modified their algorithm for sensor networks and improved the average case performance. Römer proposed another interval-based algorithm tailored for ad hoc networks settings [24]. Schmid and Schossmaier [28] provided bounds under the assumption that several performance parameters such as transmission delay bounds as well as drift bounds are given. In our work, we have made our first attempt to give bounds only from the information readily available in the specifications of the hardware components. Potentially we can improve the results when we have more information about the hardware and the environment.

As we have discussed in Section 3.4, we have improved the model by taking a closer look on the characteristics of crystal oscillators and separating drift offset bound and drift fluctuation bound. As the one similar to drift fluctuation bound, the use of bounded drift change $d\rho(t)/dt$ has been suggested in [28]. We did not use this for the following two reasons. First, the bound for drift change is hard to determine, since it is not directly derived from the data sheet provided by hardware venders. The other reason is, though temperature is the primary cause of drift fluctuation, there are certainly other effects that cause abrupt changes in clock frequency [30], such as acceleration. This implies we can only give a conservative bound on the drift change.

The use of convex polygons and estimation of clock functions as tangents to them was first proposed by Duda et al. [11] and studied in further detail by Berthaud [8] including the issues related to numerical errors. In [31], the authors designed optimal and approximate algorithms to keep only the constraints that become supports. All these works assume constant drift either indefinitely or for a short time. Our algorithm share the same idea for determining the limiting lines, but without constant drift assumptions. We also designed the algorithm in such a way that the communication pattern is more suitable for broadcast environment.

The notion of delay compensation (Section 6.1) has been discussed in [18, 28]. We have extended these results under the bounded drift fluctuation assumption and also devised a method suitable for the case of MAC-layer timestamping.

Compared to general distributed systems, sensor networks usually require much more precise synchronization. To realize that, eliminating the indeterminism from communication delay, which is mostly at the sender side, was a major problem in early times. Elson et al. dealt with this by using a reference broadcast [12], where a node broadcasts a packet and multiple receivers compare their reception time. Then TPSN [13] introduced MAC-layer timestamping as a more efficient alternative and this has been used by most of the recent studies on clock synchronization protocols in sensor networks.

Since MAC-layer timestamping reduces the transmission delay down to few microseconds, clock drift is the main issue for synchronization, since many sensor nodes use normal crystal oscillators with large drift rather than temperature compensated crystal oscillators (TCXOs) or oven-controlled crystal oscillators (OCXOs) due to energy and cost limitations. FTSP [20] estimates the clock drift and offset with high precision by using linear regression, assuming constant drift for a short period. GTSP [29] focuses on minimizing the local error among the neighboring nodes by adjusting the logical clock rate based on the estimation of clock drift of neighborhood nodes. PulseSync [19] improves the performance at distant hops by introducing a pulse-like propagation pattern. The latter two papers include convergence and optimality results on the performance including accuracy, but they are either asymptotic or probabilistic under the assumption of constant drift and bounded transmission delay. In this paper we focused on the deterministic accuracy guarantee instead, though we have demonstrated that the proposed algorithm can be combined with them to obtain good estimation.

Recent works propose synchronization protocols for specialized settings. Harmonia [16] is a synchronization protocol focused on quickly synchronizing many nodes within their very short duty cycle. Since MAC-layer timestamping is not available in their work, they use two packets to tell receivers the actual send time. This is a similar approach as IEEE 1588 (Precision Time Protocol) [4]. Rowe et al. propose the use of electromagnetic energy from AC power line as a shared trigger for clock synchronization [25].

Schmid et al. [26] proposed a method for compensating for the drift change due to temperature change. This is essentially a software implementation of what TCXOs do. Using the onboard temperature sensor, each node makes a table storing the pair of temperature and relative drift. After getting enough entries in the table, a node can estimate the drift with high accuracy and thus can compensate for that without communicating with other nodes. Our algorithm is orthogonal to this. As we have seen in Figure 12(b), we can achieve tighter error bound for smaller drift fluctuation bound. Therefore, with this or any other techniques that reduce drift fluctuation and provide deterministic guarantee for that, we can improve the accuracy guarantee with the proposed algorithm.

9 Conclusions

We have presented a clock synchronization algorithm that gives deterministic accuracy guarantees. The main idea is to find the upper and lower limits of clock function that satisfies all the constraints obtained using causality relations in communication. While the idea is similar to classical interval-based synchronization methods, we do not make simplifying assumptions such as constant drift or negligible transmission delay to obtain strict guarantees. Still, as demonstrated by the experiments, we achieve tighter guarantees owing to the bounded drift fluctuation assumption, which is confirmed by the hardware specification as well as by the preliminary experiments. We have implemented the algorithm in TinyOS and demonstrated that the accuracy guarantees are close to the simulation results. Furthermore, we extended the algorithm to obtain good estimation and achieved the estimation error up to 40% better than FTSP.

10 Acknowledgments

This work was supported in part by UCSD/LANL Engineering Institute and by NSF under grant numbers CNS-0932360, CCF-0702792, and SRS-0820034. The first author thanks Eric Yip for his help on the TinyOS implementation. The first author also thanks Muhammad Abdullah Adnan and Thomas Schmid for fruitful discussions.

References

[1] 2.4 GHz IEEE 802.15.4 / ZigBee-Ready RF Transceiver (Rev. B). `http://focus.ti.com/lit/ds/symlink/cc2420.pdf`.

[2] CMR200T / CMR250T data sheet. `http://www.citizencrystal.com/images/pdf/k-cmr.pdf`.

[3] GPS 18x Technical Specifications. `http://www8.garmin.com/manuals/GPS18x_TechnicalSpecifications.pdf`.

[4] IEEE 1588 standard for precision clock synchronization protocol for networked measurement and control systems. `http://ieee1588.nist.gov/`.

[5] TinyOS Hardware Designs: Telos (Rev B) BOM. `http://webs.cs.berkeley.edu/tos/hardware/telos/telos-revb-bom-2004-09-21.xls`.

[6] MSP430 32-kHz Crystal Oscillators (Rev. B). `http://focus.ti.com/lit/an/slaa322b/slaa322b.pdf`, 2006.

[7] MSP430x1xx Family User's Guide (Rev. F). `http://focus.ti.com/lit/ug/slau049f/slau049f.pdf`, 2006.

[8] J.-M. Berthaud. Time synchronization over networks using convex closures. *IEEE/ACM Transactions on Networking*, 8(2):265–277, 2000.

[9] P. Blum, L. Meier, and L. Thiele. Improved interval-based clock synchronization in sensor networks. In *IPSN '04: Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 349–358, 2004.

[10] S. Capkun, M. Cagalj, and M. Srivastava. Secure localization with hidden and mobile base stations. In *INFOCOM*, pages 1 –10, 2006.

[11] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating global time in distributed systems. In *ICDCS '87: Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 299–306, 1987.

[12] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.

[13] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149, 2003.

[14] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information processing Letters*, 1:132–133, 1972.

[15] D. G. Kirkpatrick and J. Snoeyink. Computing common tangents without a separating line. In *WADS '95: 4th International Workshop on Algorithms and Data Structures*, pages 183–193, 1995.

[16] J. Koo, R. K. Panta, S. Bagchi, and L. Montestruque. A tale of two synchronizing clocks. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 239–252, 2009.

[17] B. Kusy, P. Dutta, P. Levis, M. Maroti, A. Ledeczi, and D. Culler. Elapsed time on arrival: a simple and versatile primitive for canonical time synchronisation services. *Int. J. Ad Hoc Ubiquitous Comput.*, 1(4):239–251, 2006.

[18] L. Lamport. Synchronizing time servers. SRC Research Report 18, 1987.

[19] C. Lenzen, P. Sommer, and R. Wattenhofer. Optimal clock synchronization in networks. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 225–238, 2009.

[20] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, 2004.

[21] K. Marzullo and S. Owicki. Maintaining the time in a distributed system. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, 1983.

[22] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162, 1999.

[23] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

[24] K. Römer. Time synchronization in ad hoc networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182, 2001.

[25] A. Rowe, V. Gupta, and R. Rajkumar. Low-power clock synchronization using electromagnetic energy radiating from AC power lines. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 211–224, 2009.

[26] T. Schmid, Z. Charbiwala, R. Shea, and M. B. Srivastava. Temperature compensated time synchronization. *IEEE Embedded Systems Letters*, 1(2):37–41, 2009.

[27] T. Schmid, P. Dutta, and M. B. Srivastava. High-resolution, low-power time synchronization an oxymoron no more. In *IPSN '10: Proceedings of the 9th international symposium on Information processing in sensor networks*, 2010.

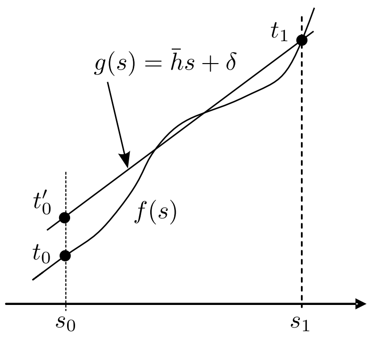


Figure 19: Proof of Lemma 1

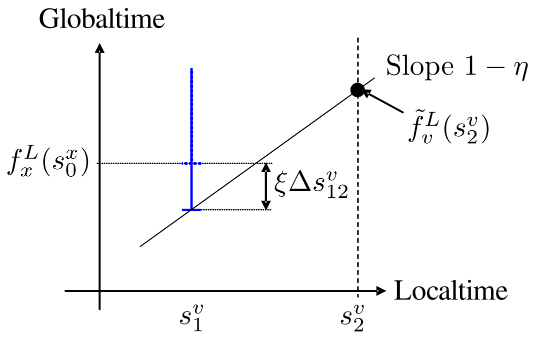


Figure 20: Proof of Claim 1

[28] U. Schmid and K. Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, 1997.

[29] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 37–48, 2009.

[30] J. R. Vig. Introduction to quartz frequency standards. Technical Report SLCET–TR–92–1, Army Research Laboratory, Electronics and Power Sources Directorate, 1992.

[31] S. Yoon, C. Veerarittiphan, and M. L. Sichitiu. Tiny-sync: Tight time synchronization for wireless sensor networks. *ACM Transactions on Sensor Networks*, 3(2):8, 2007.

A Proofs

A.1 Proof of Lemma 1

For localtime $s_0 < s_1$, assume $f(s_0) = t_0$ and $g(s_0) = t'_0$. Then we have $t_1 = t_0 + \int_{s_1}^{s_0} h(\tau)d\tau$ and $t'_0 = t_1 - \overline{h}\Delta s_{01}$, where $\Delta s_{01} = s_1 - s_0$. Since $\overline{h} - \xi \leq h(\tau) \leq \overline{h} + \xi$, we have

$$t_1 - (\overline{h} + \xi)\Delta s \leq \quad t_0 \quad \leq t_1 - (\overline{h} - \xi)\Delta s.$$

Combining these, we obtain

$$-\xi\Delta s \leq \quad t_0 - t'_0 \quad \leq \xi\Delta s.$$

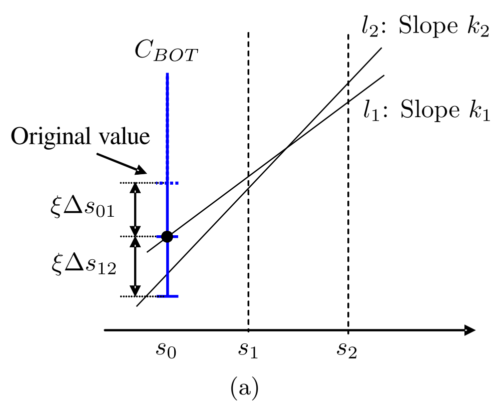
Thus at localtime s, if $f(s)$ satisfies a top constraint $f(s_i) \leq l_i$ $(s_i \leq s)$, then $g(s)$ satisfies $g(s_i) \leq l_i + \xi\Delta s_i$, where $\Delta s_i = s - s_i$. Similarly for a bottom constraint $f(s_i) \geq l_i$, $g(s)$ satisfies $g(s_i) \geq l_i - \xi\Delta s_i$. □

A.2 Proof of Claim 1

Upon receiving the message from node x, node v adds a bottom constraint at localtime s_1^v with value $f_x^L(s_0^x)$. At localtime s_2^v, as Figure 20 shows, the value of this constraint is decreased by $\xi\Delta s_{12}^v$ as a result of compensation, where $\Delta s_{12}^v \equiv s_2^v - s_1^v$.

Now let's consider a line with the slope $1-\eta$ and passes this point. About the value of this line at localtime s_2^v (denoted $\tilde{f}_v^L(s_2^v)$) and the actual lower limit $f_v^L(s_2^v)$, we have

$$f_v^L(s_2^v) \geq \tilde{f}_v^L(s_2^v), \tag{3}$$



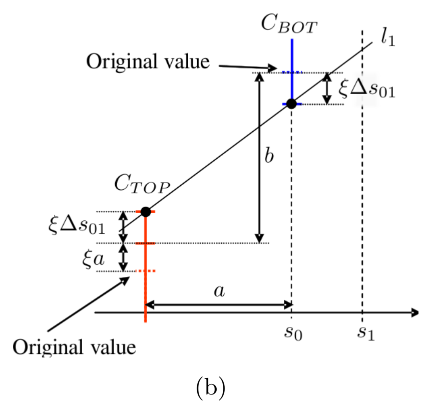


Figure 21: Proof of Theorem 2

since $1-\eta$ is the minimum possible slope of a lower line and the lower line satisfies all the constraints. Since

$$\begin{aligned}\tilde{f}_v^L(s_2^v) &= f_x^L(s_0^x) - \xi\Delta s_{12}^v + (1-\eta)\Delta s_{12}^v \\ &= f_x^L(s_0^x) + (1-\eta-\xi)\Delta s_{12}^v \qquad (4)\end{aligned}$$

and $1-\eta-\xi > 0$ from $\eta, \xi \ll 1$, the claim follows. □

A.3 Proof of Theorem 2

Let l_1, l_2 denote the lower lines calculated at localtime s_1, s_2, respectively. Let k_1, k_2 denote the slope of l_1, l_2. Also let C_{BOT} denote the bottom support of l_1 and s_0 denote the localtime of it.

If l_1 is only supported by C_{BOT}, $k_1 = 1-\eta$ (Figure 21(a)). Since $k_2 \geq 1-\eta$ and at localtime s_2 and l_2 satisfies (compensated) C_{BOT},

$$\begin{aligned}f^L(s_2) &\geq f^L(s_1) - \Delta s_{01}k_1 - \xi\Delta s_{12} + (\Delta s_{01} + \Delta s_{12})k_2 \\ &= f^L(s_1) + (k_2-\xi)\Delta s_{12} + (k_2-k_1)\Delta s_{01} \\ &\geq f^L(s_1) + (1-\eta-\xi)\Delta s_{12}. \qquad (5)\end{aligned}$$

This satisfies the theorem.

When l_1 has two supports C_{TOP} and C_{BOT} (Figure 21(b)), let a denote the difference of localtime of these. Let b denote the difference of compensated values at localtime s_0. Since there is a line that satisfies these constraints, $b/a \leq 1+\eta$. At localtime s_1, since these are the supports of l_1,

$$k_1 = \frac{b - 2\Delta s_{01}}{a} \geq 1-\eta. \qquad (6)$$

Thus we have

$$\frac{2\Delta s_{01}\xi}{a} \leq 2\eta. \qquad (7)$$

Now, at localtime s_2, since k_2 is equal to or larger than the slope of the line that connects these two (compensated) constraints,

$$k_2 \geq \frac{b - 2\Delta s_{02}\xi}{a} = k_1 - \frac{2\Delta s_{12}\xi}{a}. \qquad (8)$$

Using (7), (8), and $k_2 \geq 1 - \eta$, we have

$$\begin{aligned} f^L(s_2) &\geq f^L(s_1) - \Delta s_{01} k_1 - \Delta s_{12} \xi + (\Delta s_{01} + \Delta s_{12}) k_2 \\ &= f^L(s_1) + \Delta s_{12}(k_2 - \xi) + \Delta s_{01}(k_2 - k_1) \\ &\geq f^L(s_1) + \Delta s_{12}(1 - \eta - \xi) - \Delta s_{01} \cdot \frac{2 \Delta s_{12} \xi}{a} \\ &= f^L(s_1) + \Delta s_{12} \left(1 - \eta - \xi - \frac{2 \Delta s_{01} \xi}{a}\right) \\ &\geq f^L(s_1) + (1 - 3\eta - \xi) \Delta s_{12}. \qquad (9) \end{aligned}$$

The theorem follows. □