Technical Report No. SNU-EE-TR-1999-5

# Timing Constraint Remapping to Avoid Time Discontinuities in Distributed Real-Time Systems

Minsoo Ryu
Jungkeun Park
Seongsoo Hong

March 1999

School of Electrical Engineering
Seoul National University

Haedong Digital Library
Room 312, Building 301, Seoul National University
Shinlim-dong, Kwanak-gu, Seoul 151-742, Korea
Tel: (02)880-7278 Fax: (02)872-1293
http://haedong.snu.ac.kr

# Timing Constraint Remapping to Avoid Time Discontinuities in Distributed Real-Time Systems*

Minsoo Ryu, Jungkeun Park, and Seongsoo Hong†

**Abstract**

In this paper we propose a dynamic constraint transformation technique for ensuring timing requirements in a distributed real-time system possessing periodically synchronized distributed local clocks. Traditional discrete clock synchronization algorithms that adjust local clocks instantaneously yield time discontinuities. Such time discontinuities lead to the loss or the gain of critical time points such as task release times and deadlines, thus raising run-time faults.

While continuous clock synchronization is generally suggested to avoid the time discontinuity problem, it incurs too much run-time overhead to be implemented in software. The proposed *constraint transformation for equi-continuity (CTEC)* technique can solve this problem without modifying discrete clock synchronization algorithms. The CTEC working as an added component of discrete clock synchronization moves timing constraints out of correction intervals. In doing so, it makes use of a mapping derived from continuous clock synchronization in order to exploit the continuity property of continuous clock synchronization.

We formally prove the correctness of CTEC by showing that the CTEC with discrete clock synchronization generates the same task schedule as continuous clock synchronization. In order to show the effectiveness of CTEC, we have implemented it on a distributed platform based on the CAN bus, and performed extensive experiments. The experimental results indicate that time discontinuities present a consistency problem to real-world systems. They also show that CTEC is an effective solution to the problem, while incurring little run-tine overhead.

*The work reported in this paper was supported in part by Engineering Research Center for Advanced Control and Instrumentation (ERC-ACI) under Grant 96K3-0707-02-06-2, by KOSEF under grants 97-0102-05-01-3 and 981-0924-127-2, and by Automation and Systems Research Institute (ASRI).

†School of Electrical Engineering and ERC-ACI, Seoul National University, Seoul 151-742, Korea. Email: sshong@redwood.snu.ac.kr.

# 1 Introduction

The notion of a global clock is fundamental to the correct operation of distributed real-time systems. It provides a distributed real-time system with a common time base which enables the run-time system to monitor the order of event occurrences, measure time durations, and schedule real-time tasks in spatially distributed nodes [4, 7]. However, due to various sources of clock errors including clock drifts and clock reading errors, a perfect global clock would never exist in the real world. This gives rise to a variety of clock synchronization techniques in the literature [5, 8, 16, 12]. They include Interactive Convergence Algorithm (CNV) [8], Interactive Consistency Algorithm (COM, CSM) [8], and Fault Tolerant Average Algorithm (FTA) [5]. These algorithms rely on an assumption that each node in a distributed system possesses a local clock whose drift rate is bounded. Based on this assumption, they can maintain a global clock with known accuracy by periodically synchronizing local clocks in the system.

While these algorithms focus on fault tolerance so as to make the system functional in the presence of the bounded number of clock failures, relatively little attention has been paid to problems that may arise during the application of clock synchronization to distributed real-time systems. These problems have something to do with the implementation of clock synchronization algorithms. In discrete (instantaneous) clock synchronization, at every synchronization period, a node computes an approximate correction and adjusts its clock using this computation. This incurs abrupt changes in local time, thus yielding time discontinuities. Since such a discontinuity leads to the loss or the gain of time amounting to the correction in local clock time, important time points such as release times and deadlines of tasks may disappear or reappear resulting in run-time faults.

The potential problems of time discontinuities were partially addressed in [8, 5, 17]. To avoid negative measurements caused by backward correction [5], Srikanth et. al. [17] proposed to always set a clock to a value greater than the current time. Obviously, this causes the loss of time due to forward correction, thus events scheduled in the lost interval disappear, after all. A sufficient solution to the time discontinuity problem is continuous clock synchronization [8]. In continuous synchronization, a node spreads out a correction value over a finite interval by speeding up or slowing down its clock rather than instantaneously updating it. However, since clock values need to be adjusted every clock tick, this approach is not suitable for software implementation due to an excessive run-time overhead. It should be implemented in hardware using a phase-locked loop [16, 2, 3, 6].

Where continuous clock synchronization hardware is not available, or software clock synchronization is desirable, perhaps, for a reduced cost, we need to use a discrete clock synchronization technique in spite of the time discontinuity problem. In this paper, we propose a *constraint transformation for equi-continuity (CTEC)* that can effectively solve the time discontinuity problem in discrete clock synchronization. The CTEC dynamically modifies timing constraints such as release

times and deadlines of real-time tasks when local clocks need to be adjusted every synchronization period. The CTEC working as an added run-time component of discrete clock synchronization moves timing constraints out of correction intervals. In doing so, it makes use of a mapping derived from continuous clock synchronization in order to exploit the continuity property of continuous clock synchronization.

The CTEC also solves a clock skew problem. Even though a task completes before its deadline in its local clock time, one cannot be sure, due to clock skew, if the task has met its true deadline in reference time. A similar problem was partially addressed in [1] by Baruah et al. They proposed to model clock skew as a special case of arrival jitter and incorporated it into schedulability analysis for periodic tasks. We take a simpler approach than [1]. Our CTEC tightens up task deadlines and release times by the maximum clock skew as determined by a clock synchronization algorithm. This simple transformation enables real-time tasks to tolerate variable but bounded clock skew for timing correctness.

The CTEC possesses a very important merit: even after CTEC, one can safely rely on the result of schedulability analyses performed on the original tasks. We formally prove this property by showing that for a given task set, discrete clock synchronization with CTEC yields the same task schedule as continuous clock synchronization. To the best of our knowledge, this is the first approach that solves the time discontinuity problem of discrete clock synchronization without actually modifying discrete clock synchronization algorithms.

In order to show the effectiveness of CTEC, we have implemented it in the ARX real-time operating system we developed at Seoul National University [15]. We have performed extensive experiments on a CAN-based distributed platform running ARX. The experimental results indicate that time discontinuities pose a consistency problem to real-world systems, and that CTEC is an effective solution to the problem, while incurring little run-tine overhead.

The remainder of this paper is organized as follows. In Section 2, we discuss traditional clock synchronization techniques, introduce the time discontinuity problem, and define a good clock. In Section 3, we describe the CTEC in detail and formally prove its correctness. In Section 4, we present the results of our empirical study on discrete clock synchronization with CTEC. In Section 5, we conclude this paper with a discussion of our contributions.

## 2 Traditional Clock Synchronization

In this section we first give a brief description of the clock model and define several essential properties of good clocks. We then describe traditional discrete clock synchronization algorithms and their time discontinuity problem. For this paper, we focus only on external clock synchronization where clocks are synchronized with respect to an external reference clock. It is fairly straightforward to apply our approach to internal clock synchronization where clocks are synchronized with

respect to one another.

### 2.1 Good Clocks

Every physical clock has a different clock rate from each other due to varying environmental conditions such as temperature and radiations [5]. A local clock $C$ tends to diverge from a reference clock $C_r$. Formally, local clock $C$ is a mapping from reference time $t_r$ to local time $t$. "$C(t_r) = t$" means that the local clock tells time $t$ at reference time $t_r$. We assume that the inverse mapping of $C$ is a single-valued function, and denote it by $c(t) = t_r$. For simplicity, as in the literature [8], we assume that clocks run continuously, although real clocks progress in discrete steps. The discreteness can be modeled as an error in reading a clock (one tick error) when we use the discrete clock model. We state the properties of *good clocks* as follows.

**Definition 1.** *Clock $C(t_r)$ is a* **good clock** *with respect to reference clock $C_r$, if it is continuous, monotonically increasing, differentiable almost everywhere*[1]*, and $|\frac{dC(t_r)}{dt_r} - 1|$ is bounded.*

Continuity and monotonicity are essential properties of a good clock. Note that some events may not be observed by a clock, if it is not continuous. Also, if a clock is not monotonic, the correct event ordering cannot be established by the clock. The last condition states that the maximum drift rate $|\frac{dC(t_r)}{dt_r} - 1|$ should be bounded. We denote the maximum drift rate by $\rho$. If $\rho$ is unbounded, clock synchrony cannot be achieved.

Note that a good clock is defined with respect to a reference clock. In external synchronization, a reference clock is defined to be a unique external clock. It could be any physical clock set aside for an application system or any of existing time standards such as the International Atomic Time (TAI) and the Universal Time Coordinated (UTC) [4]. On the other hand, in internal clock synchronization, it can be an imaginary clock determined by a clock synchronization algorithm. For example, in internal synchronization using the CNV algorithm [8], the reference clock time is the average of local clock times. Throughout the paper, reference time $t_r$ is used in specifying the timing constraints of real-time tasks and enforcing timing correctness.

### 2.2 Discrete Clock Synchronization

Due to a non-zero drift rate, a physical clock deviates from the reference clock. The difference between the two clock readings at $t_r$, denoted by $\Delta(t_r)$, is referred to as *clock skew*.

$$\Delta(t_r) = C_r(t_r) - C(t_r) = t_r - C(t_r) \tag{1}$$

[1] Differentiable except for a negligible set of points in the reference time, e.g., a countable set of points.

To bound clock skew, discrete clock synchronization algorithms periodically synchronize local clocks to the reference clock. Figure 1 illustrates discrete clock synchronization. Let $T_{clk}$ denote the synchronization period in reference time, and $kT_{clk}$ denote the $k^{th}$ synchronization instant. At every $kT_{clk}$ for $k \geq 0$, a discrete clock synchronization algorithm computes correction $\Phi_k$ and applies it to a local clock. Thus, the local clock drifts apart only for a finite period of time until the next synchronization occurs. Since the drift rate is bounded, so is the clock skew. Let $C^d$ and $t^d$ denote a discretely synchronized clock and its clock time, respectively. Note that $C^d$ is constructed from physical clock $C$ by a clock synchronization algorithm. If we ignore the time overhead needed for clock synchronization, the discretely synchronized clock can be represented as follows.

$$C^d(t_r) = C(t_r) + \sum_{k=0}^{\infty} \Phi_k u(t_r - kT_{clk}), \text{ where } u(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{if } t \geq 0. \end{cases} \tag{2}$$

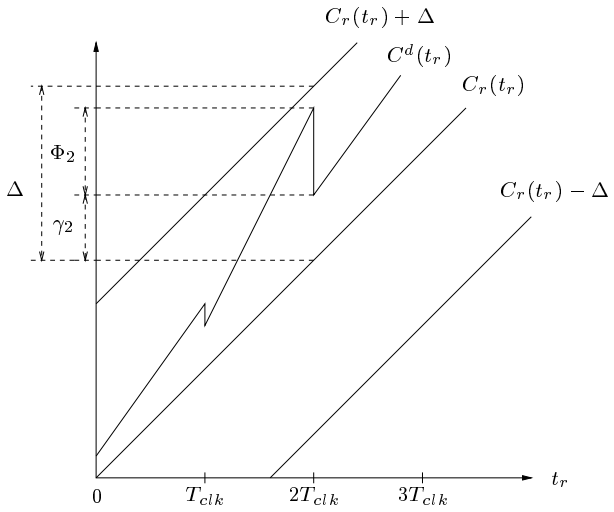


Figure 1: Discrete clock synchronization yielding bounded clock skew.

Let $\gamma_k$ denote the clock skew of $C^d$ immediately after the $k^{th}$ synchronization, and $\gamma$ denote the maximum of $\gamma_k$ for $k \geq 0$. The skew $\gamma_k$ is usually ascribed to an error in reading the value of an external reference clock [5]. Since the clock can drift for at most $T_{clk}$ time units in reference time, the maximum skew $\Delta$ is

$$\Delta = \gamma + \rho T_{clk} \tag{3}$$

### 2.3 Problems with Discrete Clock Synchronization

Discrete clock synchronization is frequently used in practice since it does not require special hardware devices. However, discrete clock synchronization causes the time discontinuity problem which

may confuse the run-time system into making a wrong judgment on real-time tasks. For example, consider a situation depicted in Figure 2. Suppose that a task has a deadline at time 19. At time 18, the current synchronization period began. In the figure, the task was running at that time, and the local clock was corrected by +2 time units since it lagged behind the reference clock by two time units. This made the local clock advance to time 20 abruptly. As a result, the running task missed its deadline.

Apparently, time discontinuities occur since discretely synchronized clock $C^d$ is not a good clock. The discrete clock synchronization algorithm makes an instantaneous update to a local clock using correction $\Phi_k$ computed every synchronization period. We call such a discontinuous interval seen by the local clock a *correction interval*. In Figure 1, the interval labeled as $\Phi_2$ is the correction interval of the second synchronization period. If $\Phi_k$ is positive, there is a loss of time amounting to $\Phi_k$ in local time. As in Figure 2, deadlines in a positive correction interval will disappear and may result in incorrect task scheduling. Similarly, if the release time of a task disappears, it may lose a chance of getting dispatched. We refer to this phenomenon as *constraint disappearance*.

On the other hand, if $\Phi_k$ is negative, constraints in a correction interval will appear again. This is called *constraint reappearance*. If the release time of a task lies in a negative correction interval, the run-time system will think that the current task instance violates the release time constraints.

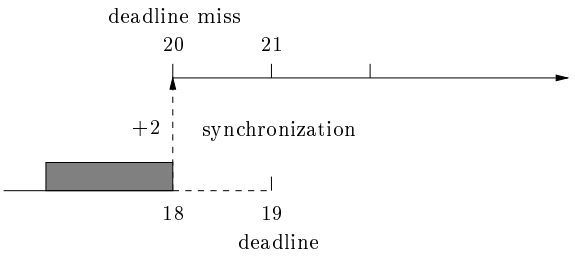


Figure 2: A deadline miss due to constraint disappearance.

## 2.4 Continuous Clock Synchronization

To avoid the problem of time discontinuity, continuous clock synchronization was suggested in [5, 8]. A continuously synchronized clock can be realized by spreading out a correction over a synchronization period. Since a continuous clock uses the same correction information as a discrete clock, a continuous clock function can be easily derived from a discrete clock. To distinguish continuous clocks from discrete ones, we denote a continuous clock and its local time by $C^c(t_r)$ and $t^c$, respectively. For a given discrete clock, there exist many mappings that generate continuous clock functions. We choose the following mapping as a clock function $C^c(t_r)$ for the $k^{th}$ synchronization. This mapping is obtained by subtracting correction $\Phi_k$ from the discrete clock $C^d$ in Eq. (2), and

spreading it uniformly over the synchronization interval $[kT_{clk}, (k+1)T_{clk})$ for $k \geq 0$.

$$\begin{aligned} t^c &= C^c(t_r) \\ &= C^d(t_r) - \Phi_k + \frac{C^d(t_r) - C^d(kT_{clk})}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})}\Phi_k \\ &= (1 + \frac{\Phi_k}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})})C^d(t^r) - (1 + \frac{C^d(kT_{clk})}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})}) \quad (4) \end{aligned}$$

where $C^d((k+1)T_{clk}-)$ denotes its left hand limit, $\lim_{t_r \to ((k+1)T_{clk}-)} C^d(t_r)$.

Though $C^c(t_r)$ is a continuous function, a clock using $C^c(t_r)$ is not a good clock when $1 + \frac{\Phi_k}{C^d((k+1)T_{clk}-)-C^d(kT_{clk})} < 0$. In such a case, $C^c(t_r)$ fails to meet the monotonicity requirement. However, in almost all practical cases, we have $(1 + \frac{\Phi_k}{C^d((k+1)T_{clk}-)-C^d(kT_{clk})}) > 0$, since $C^d((k+1)T_{clk}-) - C^d(kT_{clk}) \geq T_{clk} - 2\Delta$ and $T_{clk} - 2\Delta \gg |\Phi_k|$ for $k \geq 0$. To just give an idea, in our experiment we report later on in this paper, $T_{clk}$ and $\Delta$ are $10s$ and $213\mu s$, respectively, and $|\Phi_k|$ is less than $200\mu s$.

One can easily show that the clock skew between $C^c$ and $C_r$ is bounded. Due to the space limitation, we do not include a formal proof of this property. Instead, we show that $C^c(t_r)$ satisfies the properties of good clocks.

**Theorem 1.** *The continuously synchronized clock $C^c$ is a good clock if the underlying physical clock $C$ is a good clock and $T_{clk} > |\Phi_k| + 2\Delta$ for $k \geq 0$; that is, $C^c$ satisfies the following conditions.*

*(P1) $C^c$ is continuous.*

*(P2) The drift rate of $C^c$ is bounded.*

*(P3) $C^c$ is monotonically increasing.*

**Proof.** See Appendix A.

Note that the continuous clock function in Eq. (4) includes a future term $C^d((k+1)T_{clk}-)$ which cannot be determined at the $k^{th}$ synchronization instant. Fortunately, we can easily eliminate $C^d((k+1)T_{clk}-)$ from Eq. (4) by a simple approximation. Since $\Delta \ll T_{clk}$, we have $\frac{C^d((k+1)T_{clk}-)-C^d(kT_{clk})}{T_{clk}} \leq 1 + \frac{2\Delta}{T_{clk}} \approx 1$. Replacing $C^d((k+1)T_{clk}-) - C^d(kT_{clk})$ by $T_{clk}$ in Eq.(4) yields

$$C^c(t_r) = (1 + \frac{\Phi_k}{T_{clk}})C^d(t_r) - (1 + \frac{C^d(kT_{clk})}{T_{clk}})\Phi_k. \quad (5)$$

This mapping will be used for CTEC in the remainder of the paper.

Continuous clock synchronization using Eq. (5) can be implemented in software by changing the clock value every clock tick. Obviously, this incurs too much run-time overhead to be used in

practice. A plausible solution is to use a dedicated hardware device. To this end, we can use a frequency synthesizer based on the phase-locked loop (PLL) technique. We do not delve into this issue since it is beyond the scope of this paper. Interested readers are referred to [14, 2, 6, 3].

# 3 Constraint Transformations for Discrete Clocks

We have shown that a continuously synchronized clock using the mapping given in Eq. (5) is a good clock, thus it can eliminate the time discontinuity problem. We have also shown that continuous clock synchronization cannot be effectively implemented in software. In this section we present an alternative approach to the time discontinuity problem. The proposed solution uses a run-time constraint transformation technique we name a *constraint transformation for equi-continuity (CTEC)*.

## 3.1 Task Model

Before describing the CTEC, we first define our task model and its associated timing parameters. A periodic task $\tau_i$ is subject to three timing constraints, a *periodicity constraint*, a *release time constraint*, and a *deadline constraint*. They are represented by $\langle T_i, r_i, d_i \rangle$, respectively. The $j^{th}$ instance of $\tau_i$ is denoted $\tau_{i,j}$, and its absolute release time and deadline are denoted $R_{i,j}$ and $D_{i,j}$, respectively.

$$R_{i,j} = jT_i + r_i, \quad D_{i,j} = jT_i + d_i. \tag{6}$$

We also define additional notations for timing values which are determined at run-time. The actual computation time of task instance $\tau_{i,j}$ is denoted $e_{i,j}$. We assume that $e_{i,j}$ is bounded for all $j$ and denote its maximum by $e_i$. The actual start time and the finish time of $\tau_{i,j}$ are denoted $S_{i,j}$ and $F_{i,j}$, respectively. All of these timing parameters are specified in reference time. When they are observed by continuous local clock $C^c$, they are respectively denoted by $e^c_{i,j}$, $e^c_i$, $F^c_{i,j}$, and $S^c_{i,j}$. Similarly, when they are observed by discrete local clock $C^d$, they are respectively represented by $e^d_{i,j}$, $e^c_i$, $F^d_{i,j}$, and $S^d_{i,j}$.

Tasks on the same node are scheduled by local scheduler $\Pi(C^x, y)$ which uses local clock $C^x$ and scheduling policy $y$. For example, if the scheduler uses $C^c$ and the EDF policy [11], it is represented by $\Pi(C^c, EDF)$.

## 3.2 Constraint Transformation for Equi-Continuity

The CTEC consists of two steps. The first step is an off-line transformation that is applied once to a given task set. It tightens up task timing constraints so that tasks can tolerate the maximum amount of clock skew $\Delta$ as specified by a clock synchronization algorithm. Note that a non-zero

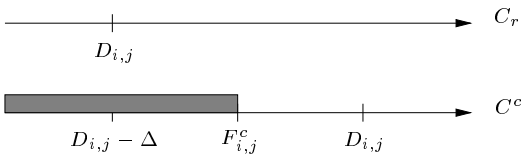


Figure 3: A schedulability problem due to clock skew.

clock skew may prevent the run-time system from ensuring tasks' true timing constraints specified in reference time. As an illustration, consider a scenario depicted in Figure 3. At time $D_{i,j}$ in reference time, task $\tau_{i,j}$ is running and the local clock $C^c$ lags behind the reference clock $C_r$ by $\Delta$. At local time $F^c_{i,j}$, task $\tau_{i,j}$ completes its execution. Since the finish time $F^c_{i,j}$ is earlier than the deadline $D_{i,j}$ in local time, the local scheduler gets confused and believes that $\tau_{i,j}$ meets its deadline, although it has in fact already missed its true deadline.

We can easily avoid such an incorrect timing check by tightening up timing constraints by $\Delta$ before making a schedulability analysis. This simple transformation of timing constraints is the first step of CTEC.

> **Step 1**: Transform $\tau_{i,j}$ into $\widehat{\tau_{i,j}}$:
> $R_{i,j} \Longrightarrow \widehat{R_{i,j}} \stackrel{\triangle}{=} R_{i,j} + \Delta, \quad D_{i,j} \Longrightarrow \widehat{D_{i,j}} \stackrel{\triangle}{=} D_{i,j} - \Delta.$

Note that period constraints need not be changed by CTEC. Since the clock synchronization algorithm keeps the clock skew bounded by a constant value $\Delta$, the frequency of task invocations does not differ in both local time and reference time.

In the second step, the CTEC dynamically modifies the timing constraints of tasks in such a way that no timing constraints lie in a correction interval in local clock time. For instance, if a deadline constraint lies in a correction interval, the proposed approach maps the deadline to a time point outside the correction interval. In doing so, any mapping can be used, as long as it satisfies the following conditions.

(C1) The transformation should not change the given temporal ordering of timing constraints. This condition is required to preserve the result of off-line schedulability analyses.

(C2) It should be possible to analyze the schedulability of a CTEC-transformed task.

One of possible mappings satisfying the above conditions is illustrated in Figure 4. In the figure, the dotted line denotes discrete clock curve $C^d$ and the solid line denotes continuous clock curve $C^c$. For a given deadline constraint $t_z$, **Step 1** derives new deadline $\widehat{t_z} = t_z - \Delta$. With continuous clock $C^c$, the new deadline corresponds to reference time $c^c(\widehat{t_z})$. Similarly, with discrete clock $C^d$, the deadline corresponds to reference time $c^d(\widehat{t_z})$. Both deadlines are globally correct, because (1)

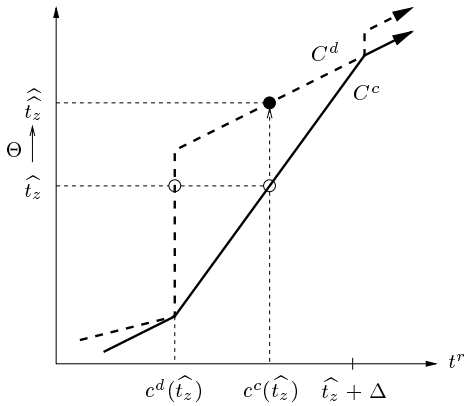


Figure 4: Constraint transformation for equi-continuity.

both clocks have bounded clock skew $\Delta$, and (2) **Step 1** guarantees that both $c^c(\widehat{t_z})$ and $c^d(\widehat{t_z})$ are earlier than the original deadline $t_z$.

The second step simply attempts to emulate continuous clock synchronization for task scheduling. For example, consider discrete clock time $\widehat{\widehat{t_z}}$ that corresponds to reference time $c^c(\widehat{t^z})$. Reference time $c^c(\widehat{t^z})$ is the time when a deadline check is made with continuous clock $C^c$. If we check deadline $\widehat{t_z}$ at $\widehat{\widehat{t_z}}$, with discrete clock $C^d$, we end up with the same deadline check that is made with continuous clock $C^d$. Thus, we are emulating $C^c$ with $C^d$.

As can be seen in Figure 4, the transformation from $\widehat{t_z}$ to $\widehat{\widehat{t_z}}$ is a mapping from continuous time $t^c$ to discrete time $t^d$ via reference time $t^r$ using the property $c^c(\widehat{t^z}) = c^d(\widehat{\widehat{t_z}})$. This mapping is the inverse of the continuous clock function given in Eq. (5). Replacing $C^d(t^r)$ by $t^d$ in Eq. (5), we can establish the following relation between $t^d$ and $t^c$.

$$t^c = (1 + \frac{\Phi_k}{T_{clk}})t^d - (1 + \frac{C^d(kT_{clk})}{T_{clk}})\Phi_k.$$

Solving for $t^d$ yields the following mapping $\Theta : t^c \longrightarrow t^d$.

$$\begin{aligned} t^d &= (\frac{T_{clk}}{T_{clk} + \Phi_k})(t^c + (1 + \frac{C^d(kT_{clk}))}{T_{clk}})\Phi_k) \\ &= \Theta(t^c). \end{aligned} \tag{7}$$

This mapping $\Theta$ is used in **Step 2**, as below.

> **Step 2**: Transform $\widehat{\tau_{i,j}}$ into $\widehat{\widehat{\tau_{i,j}}}$:
> $\widehat{R_{i,j}} \Longrightarrow \widehat{\widehat{R_{i,j}}} \triangleq \Theta(\widehat{R_{i,j}}), \quad \widehat{D_{i,j}} \Longrightarrow \widehat{\widehat{D_{i,j}}} \triangleq \Theta(\widehat{D_{i,j}}).$

Note that **Step 2** can successfully avoid time discontinuities by remapping timing constraints to those outside correction intervals. In Figure 4, $\widehat{t_z}$ is moved to $\widehat{\widehat{t_z}}$ outside the correction interval.

Note also that the mapping $\Theta$ satisfies condition (C1). Since $\Theta$ is monotonic, the ordering of timing constraints is not changed.

We summarize CTEC below by composing its two subsequent transformations.

**CTEC**: $\tau_{i,j} \overset{\text{Step 1}}{\Longrightarrow} \widehat{\tau_{i,j}} \overset{\text{Step 2}}{\Longrightarrow} \widehat{\widehat{\tau_{i,j}}}$:

- $R_{i,j} \overset{\text{Step 1}}{\Longrightarrow} \widehat{R_{i,j}} \overset{\text{Step 2}}{\Longrightarrow} \widehat{\widehat{R_{i,j}}} \triangleq \Theta(R_{i,j} + \Delta)$,
- $D_{i,j} \overset{\text{Step 1}}{\Longrightarrow} \widehat{D_{i,j}} \overset{\text{Step 2}}{\Longrightarrow} \widehat{\widehat{D_{i,j}}} \triangleq \Theta(D_{i,j} - \Delta)$.

### 3.3 Schedulability Analysis and CTEC

It looks extremely difficult to perform a schedulability analysis on a CTEC-transformed task set, since task timing constraints are modified at run-time. However, in reality, it is very straightforward to do so, since we can use the result of a schedulability analysis made with a continuous local clock. Note that CTEC emulates task scheduling performed with a continuous clock. During a schedulability analysis, we only need to take into account the clock skew of local clocks. Clock skew is handled by **Step 1** anyway.

We formally prove this essential property by showing that CTEC with discrete clock synchronization yields the same task schedule as continuous clock synchronization. For a given task set $\mathcal{T}$, let $\widehat{\mathcal{T}} = \{\widehat{\tau_i} | \tau_i \in \mathcal{T}\}$ and $\widehat{\widehat{\mathcal{T}}} = \{\widehat{\widehat{\tau_i}} | \tau_i \in \mathcal{T}\}$. **Lemma 1** shows that for a given task set $\mathcal{T}$, $\widehat{\mathcal{T}}$ and $\widehat{\widehat{\mathcal{T}}}$ generate exactly the same task schedules if they are scheduled by $\Pi(C^c, y)$ and $\Pi(C^d, y)$, respectively. This is proved by comparing the start and finish times of each pair of corresponding tasks in both schedules.

**Lemma 1.** *If $\widehat{\widehat{\mathcal{T}}}$ is scheduled by $\Pi(C^d, y)$ and $\widehat{\mathcal{T}}$ is scheduled by $\Pi(C^c, y)$, then the following conditions hold for every task instance $\tau_{i,j}$ of $\tau_i \in \mathcal{T}$,*

$$c^c(\widehat{S^c_{i,j}}) = c^d(\widehat{\widehat{S^d_{i,j}}}), \tag{8}$$

$$c^c(\widehat{F^c_{i,j}}) = c^d(\widehat{\widehat{F^d_{i,j}}}). \tag{9}$$

**Proof.** See Appendix B.

**Theorem 2.** *Task $\widehat{\widehat{\tau_{i,j}}}$ is schedulable by $\Pi(C^d, y)$ if and only if $\widehat{\tau_{i,j}}$ is schedulable by $\Pi(C^c, y)$.*
**Proof.** See Appendix C.

**Theorem 2** states that the schedulability of a CTEC-transformed task set can be equivalently tested with $\widehat{\tau_{i,j}}$. Note that the constraints of task $\widehat{\tau_{i,j}}$ are obtained off-line by **Step 1** and fixed at run-time, and that $C^c$ has the properties of a good clock. Thus, we can use any of existing

schedulability tests found in the literature [11, 9, 10]. This shows that condition (C2) is satisfied.

### 3.4 Numerical Example

As a walk-through example of CTEC, we consider a periodic task $\tau_i$ whose timing parameters are given below.

| Notation | $T_i$ | $r_i$ | $d_i$ |
|---|---|---|---|
| Quantity | $27,000\mu s$ | $5,230\mu s$ | $19,190\mu s$ |

Suppose that a given discrete clock $C^d$ is periodically synchronized with $T_{clk} = 100,000\mu s$ and its maximum skew is $\Delta = 213\mu s$. This example is pictorially depicted in Figure 5. At local time 99,790 $\mu s$, correction value $\Phi_3$ is $+210\mu s$ since local clock $C^d$ lags behind reference clock $C_r$ by $210\mu s$. Thus, the clock is set forward to local time 100,000 $\mu s$. In the figure, a correction interval is represented as a vertical line. In this walk-through example, we consider only deadline $D_{i,3}$ of $\tau_i$, which is computed below.

$$D_{i,3} = 3T_i + d_i = 81,000 + 19,190 = 100,190 \ (\mu s)$$

**Step I:** With the maximum skew $\Delta = 213\mu s$, $\tau_{i,3}$ is transformed into $\widehat{\tau_{i,3}}$ as below.

$$\widehat{D_{i,3}} = D_{i,3} - \Delta = 100,190 - 213 = 99,977 \ (\mu s)$$

**Step II:** With the correction $\Phi = 210\mu s$, mapping $\Theta$ is derived as follows.

$$\Theta(t^z) = \frac{100000}{100000 + 210}(t^z + (1 + \frac{100000}{100000}) \cdot 210)$$

Using the above mapping, $\widehat{\tau_{i,3}}$ is transformed into $\widehat{\widehat{\tau_{i,3}}}$. The new absolute deadline is

$$\widehat{\widehat{D_{i,3}}} = \Theta(\widehat{D_{i,3}}) = 100,187 \ (\mu s)$$

## 4 Empirical Study

In this section we provide an empirical evidence that time discontinuities frequently incur timing faults in real world environments, and that the proposed CTEC solves this problem with little run-time overhead. Our experiments were performed on a CAN-based distributed platform running the ARX real-time operating system [15]. We extended ARX to support external clock synchronization with CTEC. Results taken from our experiments under various workloads indicate that timing faults

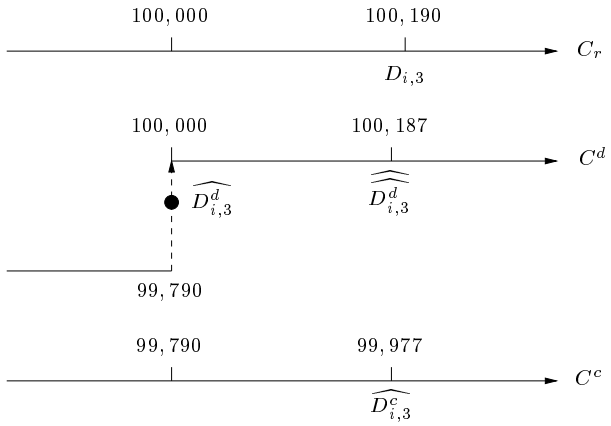


Figure 5: A numerical example of CTEC.

frequently occur as the system utilization moves close to a breakdown utilization, and that CTEC helps eliminate such timing faults.

### 4.1 Experimental Setup

Our distributed platform consisted of three Pentium PC's interconnected with the 1Mbps CAN (controller area network) bus. The CAN protocol is capable of guaranteeing bounded message transfer latencies via predictable, priority-based bus arbitration. Each PC was equipped with a CAN interface board possessing a Philips SJA1000 controller. We have written a CAN board driver and incorporated it into ARX. For more details about the CAN protocol and its chipsets, readers are referred to [18, 19].

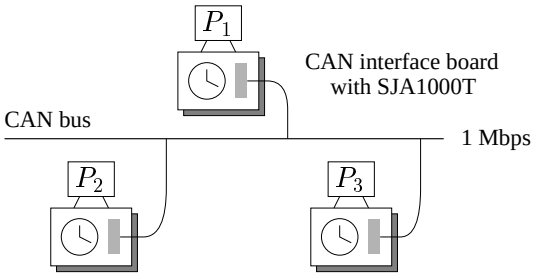


Figure 6: Experimental platform with three Pentium PC's connected via CAN.

We have implemented the *central master synchronization* protocol which is a variant of an external synchronization algorithm [4]. The periodic synchronization activity of the protocol was performed by two kernel-level threads, `initiator` and `synchronizer`, as illustrated in Figure 7. The `initiator` which ran in the master node periodically broadcast the current clock value. Upon

receiving a synchronization message from the `initiator`, the `synchronizer` in the slave node computed a correction and updated its clock. The CTEC was implemented as an independent module that was called by the thread scheduler. When invoked, it reads a correction value and transforms the most imminent timing constraint. The extra overhead by CTEC is merely 38 machine instructions per transformation on a Pentium processor.

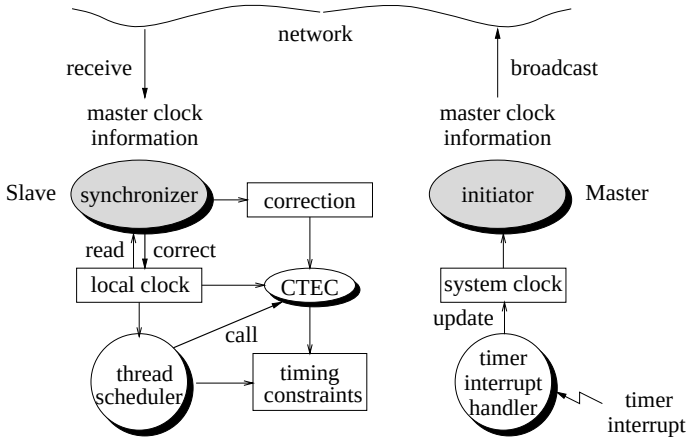


Figure 7: ARX implementation of discrete clock synchronization with CTEC.

The clock synchronization parameters used in our experiments are given in Table 1. The synchronization period $T_{clk}$ was chosen to guarantee a clock skew of $213\mu s$, as shown below.

$$\Delta = 4.4(\mu s) + 20.8(\frac{\mu s}{s}) \cdot 10(s) < 213(\mu s)$$

| Notation | $\rho$ | $\gamma$ | $T_{clk}$ | $\Delta$ |
|---|---|---|---|---|
| Quantity | $2.08 \times 10^{-5}$ | $4.4\mu s$ | $10s$ | $213\mu s$ |

Table 1: Clock synchronization parameters.

## 4.2 Experimental Results

Through our experiments, we intended to show that time discontinuities posed consistency problems to distributed real-time applications. One of such problems arises when a producer/consumer task pair communicates with each other via time-triggered messages [4, 13]. Specifically, consider a periodic producer/consumer task pair where each of the tasks is allocated in a different node and scheduled independently by a local scheduler. They use a receive queue on the consumer's side for communication. A message from the producer is written into the receive queue by a CAN controller,

and the consumer periodically reads from the queue. To enforce timely message transfers between a producer and a consumer, the consumer's release time is chosen to be greater than the producer's deadline by the maximum message transfer delay. This is illustrated in Figure 8 where $w$ denotes the maximum message transfer delay. In this setup, a deadline miss of a producer may prevent a consumer from receiving a timely message and force the consumer to read an old message from the receive queue. Since a belated message may well be overwritten before being used, we call such a message a *lost* message.

For the experiments, we generated periodic task sets consisting of 10 producer/consumer pairs. Task timing parameters were obtained via a uniform distribution with parameters shown in Table 2. We ran each task set for 60 synchronization periods ($600s$) and measured the number of message losses. Also, to observe the varying impact of CTEC under different workloads, we generated task sets with distinct utilizations $U$ ($= \sum_{\tau_i} \frac{e_i}{T_i}$) by varying task execution times.

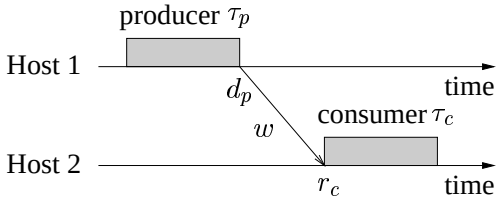


Figure 8: Producer/consumer communication via time-triggered messages.

| Parameter | $T_i$ | $r_i$ | $d_i$ |
|---|---|---|---|
| Min. | $20ms$ | $0ms$ | $10ms$ |
| Max. | $40ms$ | $40ms$ | $40ms$ |

Table 2: Distributions of task timing parameters ($r_i < d_i < T_i$).

We draw a graph in Figure 9 to compare the number of message losses with and without CTEC under varying utilizations. We categorized the generated task sets into three classes depending on their producer tasks' utilization and analyzed the impact of CTEC under these classified workloads. Note that we considered the utilization of only producer tasks since deadline misses of producers were a main source of message losses. The first class included those task sets whose utilization was less than 0.4653, and the third class included those whose utilization was greater than or equal to 0.4746. The second class included task sets in between. Without CTEC, we were able to observe message losses when the workload exceeded $U = 0.4653$. Such message losses were resulted from clock skew and time discontinuities. Without CTEC, and even with CTEC, we were able to observe the increased number of message losses when the workload exceeded $U = 0.4746$. Such message

losses were in large part due to genuine deadline misses. In our experiments, $U = 0.4746$ was a breakdown utilization. As defined in [9], a breakdown utilization is a schedulable utilization bound for a task set. The breakdown utilization in our experiments was much less than the rate monotonic bound since tasks used in the experiments had positive release time constraints and tight deadlines.

- **Light load case** ($U < 0.4635$): No message losses were observed in this case, regardless of the use of CTEC. This is because deadline disappearance does not always lead to a deadline miss. A deadline miss occurs when a task is running at the very moment its deadline disappears. This does not occur when a system is lightly loaded.

- **Medium load case** ($0.4635 \leq U < 0.4746$): Even before the system reached the breakdown utilization, task sets without CTEC yielded message losses, whereas those with CTEC did not lose a single message.

- **Heavy load case** ($0.4746 \leq U$): As the CPU got overutilized, both task sets with and without CTEC yielded many message losses. Still, CTEC significantly helped reduce the number of lost messages. In particular, when $U > 0.54$, CTEC eliminated about a half of message losses. This indicates that CTEC is very beneficial in keeping a system stable while the system experiences a transient overload. Note that CTEC adds very small run-time overhead to the system.

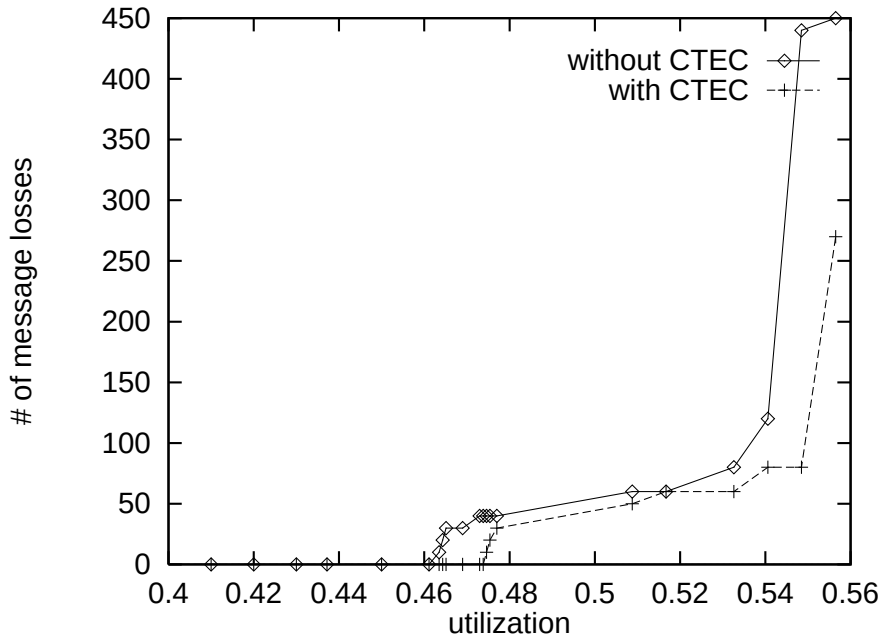


Figure 9: Message losses under varying utilization with and without CTEC.

# 5 Conclusion

We have presented a timing constraint transformation technique CTEC to avoid time discontinuities in discrete clock synchronization. The CTEC working as a run-time component invoked by the thread scheduler remaps timing constraints to those outside correction intervals. In doing so, it makes use of a mapping derived from continuous clock synchronization.

The CTEC proposed in this paper allows for several benefits: (1) it can be effectively implemented in software with little run-time overhead, since it does not change discrete clock synchronization algorithms; and (2) it does not sacrifice the analyzability of a CTEC-transformed task set, although task constraints are dynamically modified. We have formally proved the correctness of CTEC by showing that the CTEC with discrete clock synchronization generates the same task schedule as continuous clock synchronization.

In order to show the effectiveness of CTEC, we have implemented it and performed extensive experiments on a distributed platform based on the CAN bus. In our experiments using producer/consumer tasks, we measured the number of message losses between producers and consumers with and without CTEC. Our experimental results indicate that time discontinuities significantly contribute to message losses when a system gets fully utilized. CTEC will be very beneficial in reducing message losses when a system experiences a transient overload.

There are several future research directions. We are currently extending the proposed approach to deal with synchronization faults. If a node experiences transient faults for several synchronization periods, its local clock may deviate beyond a specified bound. In such a case, a new continuous mapping should be chosen to take into account the missed synchronization periods, and CTEC should be modified accordingly. In addition, we are also seeking to find other applications that may be benefited from CTEC. For example, in applications where obtaining correct time stamps for occurred events is important, a negative correction may lead to the incorrect temporal ordering of events. The equi-continuity and monotonicity properties of CTEC will be useful in those applications.

# References


[1] S. K. Baruah, D. Chen, and A. K. Mok. Jitter concerns in periodic task systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 68–77, December 1997.

[2] Roland E. Best. *Phase-Locked Loops: Theory, Design, and Applications.* McGraw-Hill, Inc., 1993.

[3] J. L. W. Kessels. Two designs of a fault-tolerant clocking system. *IEEE Transactions on Computers*, C-33(10):912–919, October 1984.

[4] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997.

[5] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[6] C. M. Krishna, K. G. Shin, and R. W. Butler. Ensuring fault tolerance of phase-locked clocks. *IEEE Transactions on Computers*, C-34(8):752–756, August 1985.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[9] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[10] J. P. Lehoczy. Fixed prioiry scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 201–209, December 1990.

[11] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[12] P. Ramanathan, D. D. Kandlur, and K. G. Shin. Hardware-assisted software clock synchronization for homogeneous distributed systems. *IEEE Transactions on Computers*, C-39(4):514–524, April 1990.

[13] M. Ryu and S. Hong. End-to-end design of distributed real-time systems. *Control Engineering Practice*, 6(1):93–102, January 1998.

[14] M. Ryu and S. Hong. Revisiting clock synchronization problems: Static and dynamic constraint transformations for correct timing enforcement. Technical Report SNU-EE-TR-1998-3, School of Electrical Engineering, Seoul National University, August 1998.

[15] Y. Seo, J. Park, and S. Hong. Efficient user-level I/O in the ARX real-time operting systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1998.

[16] K. G. Shin and P. Ramanathan. Synchronization of a large clock network in the presence of malicious faults. *IEEE Transactions on Computers*, C-36(1):2–12, January 1987.

[17] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of ACM*, 34(3):626–645, July 1987.

[18] K. Tindell, H. Hansson, and A. Wellings. Analysing real-time communications: Controller area network (CAN). In *Proceedings, IEEE Real-Time Systems Symposium*, December 1994.

[19] K. M. Zuberi and K. G. Shin. Non-preemptive scheduling of messages on controller area network for real-time control applications. In *Real-Time Technology and Applications Symposium*, pages 240–249, May 1995.

## Appendix A – Proof of Theorem 1.

(P1) We show that $C^c(t_r)$ is continuous. By definition, $C(t_r)$ is continuous in interval $[kT_{clk}, (k+1)T_{clk})$, thus $C^c(t_r)$ is also continuous in that interval. We need to show $C^c(t_r)$ is continuous only at each synchronization points $kT_{clk}$ for $k > 0$. Recall that $C^c(t_r)$ is continuous at $kT_{clk}$ if its lefthand limit $C^c(kT_{clk}-)$ is equal to $C^c(kT_{clk})$. As $t_r \to (kT_{clk}-)$ in Eq. (4), we have

$$\begin{aligned}\lim_{t_r \to (kT_{clk}-)} C^c(t_r) &= \lim_{t_r \to (kT_{clk}-)} \{C^d(t_r) - \Phi_{k-1} + \frac{C^d(t_r) - C^d((k-1)T_{clk})}{C^d(kT_{clk}-) - C^d((k-1)T_{clk})}\Phi_{k-1}\} \\ &= C^d(kT_{clk}-) \\ &= C(kT_{clk}) + \sum_{j=0}^{k-1} \Phi_j.\end{aligned}$$

Using Eqs. (4) and (2), we also have

$$C^c(kT_{clk}) = C^d(kT_{clk}) - \Phi_k = C(kT_{clk}) + \sum_{j=0}^{k-1} \Phi_j.$$

This proves the desired result

$$C^c(kT_{clk}-) = C^c(kT_{clk}).$$

(P2) We show that $|\frac{dC^c(t_r)}{dt_r} - 1|$ is bounded from above and derive its upper bound. By definition, $C^c(t_r)$ is differentiable at all points in $(kT_{clk}, (k+1)T_{clk})$. Differentiating $C^c(t_r)$ by $t_r$ in Eq. (2), we have

$$\begin{aligned}\frac{dC^c(t_r)}{dt_r} &= (1 + \frac{\Phi_k}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})})\frac{dC^d(t_r)}{dt_r} \\ &= (1 + \frac{\Phi_k}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})})\frac{dC(t_r)}{dt_r}.\end{aligned}$$

Let $\Phi$ be the maximum of $|\Phi_k|$ for $k > 0$. Since $1 - \rho \leq \frac{dC(t_r)}{dt_r} \leq 1 + \rho$, we have

$$(1-\rho)(1-\frac{\Phi}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})}) \leq \frac{dC^c(t_r)}{dt_r} \leq (1+\rho)(1+\frac{\Phi}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})}).$$

for $t_r$ in interval $(kT_{clk}, (k+1)T_{clk})$. After replacing $C^d((k+1)T_{clk}-) - C^d(kT_{clk})$ by $T_{clk}$, we have the following by a little algebraic manipulation.

$$|\frac{dC^c(t_r)}{dt_r} - 1| \leq max\{|-\frac{\Phi}{T_{clk}} - \rho(1-\frac{\Phi}{T_{clk}})|, |\frac{\Phi}{T_{clk}} + \rho(1+\frac{\Phi}{T_{clk}})|\}.$$

This proves the desired result.

(P3) The monotonicity property directly follows from the fact that $C^c$ is continuous as proved in (P1) and the assumption that $(1 + \frac{\Phi}{C^d((k+1)T_{clk}-) - C^d(kT_{clk})}) > 0$. □

## Appendix B – Proof of Lemma 1.

Let $\mathcal{L}$ be a set of all the periodic task instances in $\mathcal{T}$ such that $\mathcal{L} = \{\tau_{i,j} | \tau_i \in \mathcal{T} \text{ and } j \geq 0\}$. We define priorities among task instances $\tau_{i,j} \in \mathcal{L}$ such that $\tau_{i,j}$ inherits the priority of $\tau_i$ at its $j^{th}$ period. Ties are broken by choosing a task with the earliest absolute release time. For instance, if EDF is used, the priorities are obtained according to the absolute deadlines of task instances. We define $\mathcal{L}_m$ be a priority ordered subset of $\mathcal{L}$ such that $\mathcal{L}_m$ contains $m$ highest priority task instances taken from $\mathcal{L}$. We relabel task instances in $\mathcal{L}_m$ such that $\mathcal{L}_m = \{\tau_1^a, \tau_2^a, \ldots, \tau_m^a\}$ where $\tau_1^a$ has the highest priority. For notational convenience, we denote the execution time of $\tau_i^a$ by $e_i$ which is measured with reference clock $C_r$.

We define availability function $\Psi(t_r, \mathcal{L}_m, \Pi(\cdot))$ as below.

$$\Psi(t_r, \mathcal{L}_m, \Pi(\cdot)) = \begin{cases} 0 & \text{if } \tau_i^a \in \mathcal{L}_m \text{ is running at } t_r \text{ in reference time} \\ 1 & \text{else} \end{cases}$$

The availability function denotes the distribution of idle time in the schedule of task instances in $\mathcal{L}_m$, when they are scheduled by a local scheduler $\Pi(\cdot)$.

We now prove by induction on $m$. When $m = 1$, we have $c^c(\widehat{S_1^c}) = c^d(\widehat{\widehat{S_1^d}})$. Since the execution times of $\tau_1^a$ are the same in both schedules, we have $c^c(\widehat{F_1^c}) = c^d(\widehat{\widehat{F_1^d}})$. Obviously, this implies that $\Psi(t_r, \mathcal{L}_1, \Pi(C^c, y)) = \Psi(t_r, \mathcal{L}_1, \Pi(C^d, y))$. As an induction hypothesis, we assume that the following holds for $m \leq n-1$.

$$c^c(\widehat{S_m^c}) = c^d(\widehat{\widehat{S_m^d}}), \quad c^c(\widehat{F_m^c}) = c^d(\widehat{\widehat{F_m^d}}).$$

It immediately follows that

$$\Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^c, y)) = \Psi(t^r, \mathcal{L}_{n-1}, \Pi(C^d, y)).$$

Let $c^c(\widehat{S_n^c}) = t_{r,1}$ and $c^c(\widehat{F_n^d}) = t_{r,2}$. The start time $t_{r,1}$ of $\widehat{\tau_n^a}$ is the first idle time point no earlier than the release time of $\widehat{\tau_n^a}$. The finish time $t_{r,2}$ is determined by the execution time of $\widehat{\tau_n^a}$ and the availability function. We restate $t_{r,1}$ and $t_{r,2}$ formally as below.

$$t_{r,1} = min\{t_r, \text{ such that } c^c(\widehat{R_n}) \leq t_r \text{ and } \Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^c, y)) = 1\}, \quad (10)$$

$$t_{r,2} = min\{t_r, \text{ such that } \int_{t_{r,1}}^{t_r} \Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^c, y))dt_r = e_n)\}. \quad (11)$$

Let $c^d(\widehat{\widehat{S_n^d}}) = t_{r,3}$ and $c^d(\widehat{\widehat{F_n^d}}) = t_{r,4}$. Similarly,

$$t_{r,3} = min\{t_r, \text{ such that } c^d(\widehat{\widehat{R_n}}) \leq t_r \text{ and } \Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^d, y)) = 1\}, \quad (12)$$

$$t_{r,4} = min\{t_r, \text{ such that } \int_{t_{r,3}}^{t_r} \Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^d, y))dt_r = e_n)\}. \quad (13)$$

We compare Eq. (10) with Eq. (12). Since

$$\begin{aligned} c^c(\widehat{R_n}) &= c^d(\widehat{\widehat{R_n}}), \\ \Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^c, y)) &= \Psi(t_r, \mathcal{L}_{n-1}, \Pi(C^d, y)), \end{aligned}$$

we have $t_{r,1} = t_{r,3}$.

$$c^c(\widehat{S_n^c}) = c^d(\widehat{\widehat{S_n^d}}).$$

Since $t_{r,1} = t_{r,3}$, we also have $t_{r,2} = t_{r,4}$.

$$c^c(\widehat{F_n^c}) = c^d(\widehat{\widehat{F_n^d}}).$$

This completes the proof. □

## Appendix C – Proof of Theorem 2.

We consider the "if" part first. By the assumption that $\widehat{\tau_{i,j}}$ is schedulable by $\Pi(C^c, y)$, we have

$$\widehat{R_{i,j}} \leq \widehat{S_{i,j}^c} \leq \widehat{F_{i,j}^c} \leq \widehat{D_{i,j}}.$$

Since $\Theta(\cdot)$ is monotonically increasing,

$$\Theta(\widehat{R_{i,j}}) \leq \Theta(\widehat{S^c_{i,j}}) \leq \Theta(\widehat{F^c_{i,j}}) \leq \Theta(\widehat{D_{i,j}}).$$

Using the property $c^c(\hat{t}^z) = c^d(\hat{\hat{t}}^z)$, $\Theta(\cdot)$ can be equally represented as $C^d(c^c(\cdot))$. Also from the result of **Lemma 1**, we can write

$$\Theta(\widehat{S^c_{i,j}}) = C^d(c^c(\widehat{S^c_{i,j}})) = \widehat{\widehat{S^d_{i,j}}}, \quad \Theta(\widehat{F^c_{i,j}}) = C^d(c^c(\widehat{F^c_{i,j}})) = \widehat{\widehat{F^d_{i,j}}},$$

proving the following result

$$\widehat{\widehat{R_{i,j}}} \leq \widehat{\widehat{S^d_{i,j}}} \leq \widehat{\widehat{F^d_{i,j}}} \leq \widehat{\widehat{D_{i,j}}}.$$

The converse, or the "only if" part, is proved in the same manner. □