Accurate Clock Models for Simulating Wireless Sensor Networks

Federico Ferrari Andreas Meier Lothar Thiele

Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
{ferrari, a.meier, thiele}@tik.ee.ethz.ch

ABSTRACT

Time-critical parts of Wireless Sensor Network (WSN) applications, like medium access (MAC) and synchronization protocols, require an accurate timing analysis of their behavior. Meaningful simulation results are only achieved when the simulator provides a realistic model of the node's hardware (HW) clock. This paper provides three main contributions: (1) a realistic clock-drift model that allows to simulate HW clocks with an accuracy error of less than 1 μsec, (2) a clear interface to schedule timers/events that abstracts the artificial simulation time from the user and ensures that the HW time is used when implementing applications, and (3) a clock translator that converts the HW time of a node to the simulation time when scheduling events (hidden from the user). We implement and validate these extensions in Castalia, a WSNs simulator based on the OMNeT++ platform. We show that they have only minimal effects on the memory and processing demands of the simulation.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Algorithms, Documentation, Experimentation

Keywords

Sensor Networks, Simulation, Clock Drift, Time Synchronization, MAC, Scheduling, Timers, OMNeT++, Castalia

1. INTRODUCTION

An important step for meaningful simulations of Wireless Sensor Networks (WSNs) is to model the nodes hardware (HW) clock. Relying on an accurate model of a real HW clock, as opposed to scheduling events in simulation time (*i.e.*, real time), is of utmost importance for time-critical parts of a protocol stack. For instance, a MAC protocol that requires global synchronization (*e.g.*, TDMA) cannot be properly evaluated if all clocks run at the same speed. Several simulators like Castalia, a simulator based on the OMNeT++ platform specifically designed for Wireless Sensor Networks (WSNs), suggest to manually account for clock drift when scheduling events, assuming a random but constant clock drift. This approach implies two major disadvantages for the application designer. Firstly, the constant-drift model for the HW clock has shown to be too simplistic [4]. In particular, applications that try to compensate for clock drift (*e.g.*, synchronization protocols) cannot be properly evaluated. More complex models should thus be used to simulate a realistic behavior, where the drift of a clock from its nominal frequency changes over time. Secondly, a dedicated interface to the HW clock is missing and the application designer has only access to the real-time (simulation time). When scheduling events, inaccuracies of the HW clock have to manually be taken into account. This approach is error-prone, since missing the time translation causes an inconsistent timing behavior.

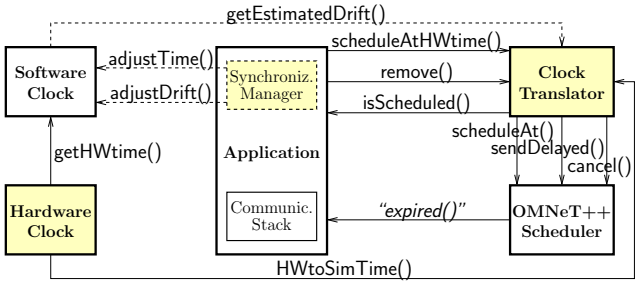

Figure 1: Clock modules and interfaces.

This paper makes three contributions for dealing with HW clocks in a simulator. We propose (1) an accurate model for the HW clock that assumes a bounded drift and a bounded drift variation, (2) a clear interface to schedule events that abstracts the notion of the simulation time from the application, and (3) a clock translator that converts the HW time to the simulation time.

As shown in Figure 1 for our Castalia case study, we implemented the clock and the scheduling modules separately from the main application module. The Hardware Clock module provides the current time as read from the HW clock, by using the model discussed in Section 2. This HW time is translated to the simulation time by the Clock Translator module. The application can schedule an event directly in

OMNeT++ 2010 March 15–19, Torremolinos, Malaga, Spain.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8693
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8693

HW time by invoking the method scheduleAtHWtime(): the time translation is completely hidden from it. The simulation time generated by the Clock Translator is used to schedule the desired event by invoking the standard scheduling method of the simulator (*e.g.*, scheduleAt() in OMNeT++). Following the approach of OpenCastalia [5], we also enable an application to be simulated concurrently to a synchronization manager. The (optional) synchronization manager module provides a more accurate software (SW) time to the application by estimating the drift of the HW clock.

The remaining of the paper is organized as follows. Section 2 discusses HW clocks, drift models, and possible approximations for simulating them. Section 3 discusses our extensions for enabling the simulation of more accurate clock models and the impact on memory consumption and execution time. Section 4 concludes this paper.

2. HARDWARE CLOCK

The local time provided by the HW clock of a sensor node can be expressed as a function $h(t)$ of the real-time t [4]. The HW time is usually the value of a counter that counts intervals of an ideally fixed length. The clock drift $\rho(t)$ is defined as the deviation of the rate with which the counter is actually incremented by the oscillator and the ideal rate 1: $\rho(t) = (\mathrm{d}h(t)/\mathrm{d}t) - 1$. An ideal clock would therefore have drift $\rho \equiv 0$. We assume $\rho(t) > -1$: a clock never stops and never runs backwards. Due to changes in supply voltage, temperature, and aging, the drift of a real clock fluctuates over time. Different models try to represent this behavior.

The *bounded-drift* model assumes that the drift $\rho(t)$ is bounded: $|\rho(t)| < \hat{\rho},\ \forall t$. A special case of this model is the *constant-drift* model, where the clock drift $\rho(t) = \rho$ is constant yet still bounded: $|\rho| \leq \hat{\rho}$. Reasonable values for standard sensor nodes equipped with inexpensive oscillators are $\hat{\rho} \in [10, 100]$ ppm[1] [4].

The *bounded-drift-variation* model assumes that the variation of the drift $\vartheta(t) = \mathrm{d}\rho(t)/\mathrm{d}t = \mathrm{d}^2h(t)/\mathrm{d}t^2$ over time is bounded: $|\vartheta(t)| < \hat{\vartheta},\ \forall t$. Assuming a bounded drift variation is very reasonable for most settings, since environmental conditions (*i.e.*, temperature, battery voltage, age of oscillator) change gradually. Furthermore it should be noted that drift estimation assumes a bounded drift variation.

The bounded-drift and the bounded-drift-variation models can be unified into one *combined* model that covers them both. For instance, setting $\hat{\vartheta} = \infty$ in the combined model results in a bounded-drift model. If not otherwise stated, we assume a maximal drift of $\hat{\rho} = 100$ ppm and a maximum drift-variation of $\hat{\vartheta} = 10^{-8}$ sec^{-1} in the remaining of the paper, which are common values for (inexpensive) clocks equipping sensor nodes.

Other models for the drift exist, such as the *tuning-fork* clock model that directly links the clock drift $\rho(T)$ of an oscillator to the temperature T. The drift is given by $\rho(T) = -A(T - T_0)^2$ ppm, where T is the temperature measured in degree Celsius, $T_0 = 25$ °C is the temperature at which the drift is zero, and $A \in [0.03, 0.042]$ ppm/°C^2 for common 32KHz sensor node clocks. It should be mentioned that the tuning-fork model is covered by the combined model. The drift is bounded since the temperature is within a limited range and also the drift variation is bounded since the temperature varies with a limited rate (*i.e.*, it cannot jump).

[1]ppm: parts per million, *i.e.*, 10^{-6}.

2.1 Approximation of Hardware Clock

It is impossible to use the exact $h(t)$ in simulation, since the analytic expression of the HW clock is usually unknown and thus an infinite number of points would be necessary to exactly represent the HW clock. It is possible to use a continuous piecewise linear approximation $h^*(t)$, where samples of $h(t)$ values are taken at regular intervals t_{int} of the real-time t (see Figure 2), starting at time t_0 when the HW clock is started: $h(t_0) \equiv 0$. Every k-th part of the piecewise approximation corresponds to the k-th time interval given by $t \in [t_0 + kt_{\mathrm{int}}, t_0 + (k+1)t_{\mathrm{int}}]$. The HW clock $h(t)$ and its approximation $h^*(t)$ during these intervals are denoted as $h_k(\tilde{t})$ and $h_k^*(\tilde{t})$, respectively, where $\tilde{t}$ is the real-time displacement within a window: $h_k(\tilde{t}) \equiv h(t_0 + kt_{\mathrm{int}} + \tilde{t})$ and $h_k^*(\tilde{t}) \equiv h^*(t_0 + kt_{\mathrm{int}} + \tilde{t})$. Due to the piecewise linear approximation, $h(t)$ and $h^*(t)$ are equal at the beginning and at the end of each interval, *i.e.*, $h_k(0) = h_k^*(0)$ $(\equiv h_k)$ and $h_k(t_{\mathrm{int}}) \equiv h_k^*(t_{\mathrm{int}}),\ \forall k \geq 0$.

The linear approximation implies a constant clock drift ρ_k^* for every interval k: $\rho_k^* = (h_k(t_{\mathrm{int}}) - h_k)/t_{\mathrm{int}} - 1$. The piecewise clock approximation $h_k^*(\tilde{t})$ is then given by

$$h_k^*(\tilde{t}) = h_k + \tilde{t} \cdot (1 + \rho_k^*). \tag{1}$$

For simulating the clock, drift values ρ_k are randomly generated within the bounds provided by the combined model. Alternatively, a temperature curve can be used to calculate the drift by using the tuning-fork model with the average temperature T_k of the k-th interval. In both cases, the linear approximation of the clock introduces a maximum error $\varepsilon = \max(|h(t) - h^*(t)|)$, which we determine analytically in the following.

In the combined model, we have bounded drift ($|\rho(t)| < \hat{\rho}$) and drift-variation ($|\vartheta(t)| < \hat{\vartheta}$). For a sufficiently small t_{int} we further have

$$\vartheta(t) \approx (\rho(t + t_{\mathrm{int}}) - \rho(t))/t_{\mathrm{int}}. \tag{2}$$

From $|\vartheta(t)| < \hat{\vartheta}$, it follows that we have a second constraint for the clock drift for any interval $k > 0$:

$$\rho_{k-1} - \hat{\vartheta} \cdot t_{\mathrm{int}} < \rho_k < \rho_{k-1} + \hat{\vartheta} \cdot t_{\mathrm{int}}. \tag{3}$$

The approximation of $\vartheta(t)$ in (2) holds if $\max|\rho(t + t_{\mathrm{int}}) - \rho(t)| \approx \hat{\vartheta}t_{\mathrm{int}} \ll \hat{\rho}$. This limits t_{int} for the piecewise linear approximation:

$$t_{\mathrm{int}} \ll \hat{\rho}/\hat{\vartheta} = 10^{-4}/10^{-8}\ \text{sec} = 10,000\ \text{sec}\,. \tag{4}$$

At the beginning and at the end of each interval of length t_{int}, $h(t) = h^*(t)$. The maximum error occurs when $h(t)$ has a constant drift variation $\vartheta(t) = \hat{\vartheta}$, since for $t_{\mathrm{int}} \ll \hat{\rho}/\hat{\vartheta}$ the drift is limited by the maximum allowed variation $\hat{\vartheta}$. In this case: $\Delta(t) = |h(t) - h^*(t)| = \frac{\hat{\vartheta}}{2}t(t_{\mathrm{int}} - t),\ 0 \leq t \leq t_{\mathrm{int}}$. The maximum error occurs thus at $t = t_{\mathrm{int}}/2$:

$$\varepsilon(t_{\mathrm{int}}) = \max_{0 \leq t \leq t_{\mathrm{int}}} \Delta(t) = \Delta(t_{\mathrm{int}}/2) = \hat{\vartheta} \cdot t_{\mathrm{int}}^2/8 \tag{5}$$

2.2 Clock Translation

In order to perform translations between HW time and real-time, let us suppose that the values for t_0, t_{int}, $\{\rho_k^*\}$, and $\{h_k\}$ are available $\forall k \geq 0$ (see Section 3.2). It is straightforward to compute the HW time $h^*(t)$ that corresponds to the real-time t, by first computing the corresponding interval $k = \lfloor (t - t_0)/t_{\mathrm{int}} \rfloor$ and then applying (1).

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8693
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8693

In simulation, however, it is usually required to compute the real-time t as a function of the HW time $h^*(t)$. Every time a node schedules a timer, this conversion is necessary in order to determine the real-time (simulation time) at which this timer is supposed to expire. The main problem is that computing t is not straightforward, since the inverse function of $h^*(t)$ is unknown. In particular, we have to find the interval k to which the HW time $h^*(t)$ belongs, *i.e.*, to find k such that $h_k \leq h^*(t) \leq h_{k+1}$.

Since we cannot schedule events in the past, we can determine a lower bound $\bar{k}$ for k, which corresponds to the current interval: $k \geq \bar{k} = \lfloor (t_{\text{sim}} - t_0)/t_{\text{int}} \rfloor$, where t_{sim} is the current simulation time. Since WSN applications usually schedule timers that are in the very near future, we usually have $k = \bar{k}$. If instead $h^*(t) > h_{\bar{k}+1}$, we check intervals $\bar{k}+1$, $\bar{k}+2$, ... in order to find the interval k to which t belongs. Knowing the interval k, we can determine the real-time t as follows:

$$t = t_0 + k \cdot t_{\text{int}} + (h^*(t) - h_k)/(1 + \rho_k^*). \qquad (6)$$

3. IMPLEMENTATION

The approximated model of the HW clock and the related translation functions (1, 6) have been implemented as Castalia modules. Castalia [1, 2] is a simulator based on the OMNeT++ platform specifically designed for Wireless Sensor Networks (WSNs) and networked embedded systems in general. It provides an advanced model of the wireless channel, which includes temporal variation of path loss. Packet collisions are calculated based on the signal to interference ratio (SIR). The radio model features multiple transmission power levels and multiple operational states, with different power consumptions and transition times between the radio states. In the following, we discuss implementation details and we also introduce a new interface to schedule events.

3.1 New Event Scheduling Interface

Our implementations provides the new interface scheduleAtHWtime() that should be used by the application and the protocol stack for scheduling an event (*e.g.*, a timer). This interface uses the HW time as a reference and hides the translation between HW time and simulation time to an application programmer by implicitly using the Clock Translator module (see Figure 1). Only this interface method should be used to schedule events, since the HW time is the only time reference available on a real sensor node. It is still possible to directly use the standard scheduleAt() method of OMNeT++, for debugging or statistical purposes.

3.2 Clock Translator

With a non-constant drift model (such as the bounded-drift and bounded-drift-variation model), multiple drift values have to be computed and stored for the piecewise linear approximation of the HW clock for every sensor node. We assumed in Section 2.2 that $\{\rho_k^*\}$ and $\{h_k\}$ are known for *all* k during the simulated time, a prerequisite for scheduling events with the OMNeT++ interface during the whole simulation time. The simplest approach would be to precompute these values (depending on the chosen drift model) during the initialization phase and store them in a lookup table. In several simulation scenarios it is however required to simulate the long-term behavior of an application, *i.e.*, the end of simulation (sim-time-limit in OMNeT++) is set to high

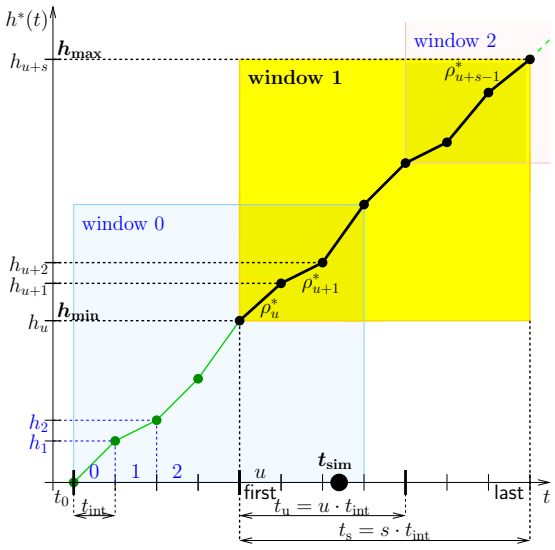

Figure 2: Sliding time windows used for translations between HW time $h^*(t)$ and real-time t ($u = 4$, $s = 7$).

values, in the order of days or months. Storing the values for all time intervals during the initialization phase implies a prohibitively large memory overhead for the lookup table.

We propose to use a sliding storage window (see Figure 2), which is kept independently for every node in the simulation. At initialization (*i.e.*, at real-time t_0), the clock module precomputes and stores only the values $\{\rho_k^*\}$ and $\{h_k\}$ for the first s intervals $k \in [0, s-1]$ (of real-time length t_{int}), where the integer parameter $s > 0$ represents the window size. Furthermore every node schedules an event at $u \cdot t_{\text{int}}$ time units (*i.e.*, at real-time $t_0 + u \cdot t_{\text{int}}$), where u ($0 < u \leq s$) is an integer parameter that represents the window update interval. When this event is received, the first u values for $\{\rho_k^*\}$ and $\{h_k\}$ are discarded (they belong to past intervals and they are not going to be used anymore), and u new values are computed for intervals $[s, s+u-1]$. A new event is then scheduled for real-time $t_0 + 2u \cdot t_{\text{int}}$, where the oldest u values will again be discarded and u newest values will be computed. This process of scheduling events to update the window is repeated until the end of the simulation.

During a simulation, the HW time of a node is only available for the current time window, up to h_{max} (see Figure 2). If an event has to be scheduled at a HW time corresponding to a real-time within the current window, it is directly scheduled after translating the HW time to the real-time by applying (6). If an application schedules an event outside the current window (*i.e.*, at a time larger than h_{max}), the clock translator cannot yet schedule the event. The event is therefore kept in a local queue and is scheduled with OMNeT++ as soon as the time is covered by the updated time window. As an example, at t_{sim} in Figure 2 only values for window 1 are stored. Therefore, it is not possible to schedule a timer at a HW time corresponding to a real-time outside window 1. It should be noted that the majority of events that are scheduled by WSN applications are very short term in nature (*e.g.*, setting timeouts at the radio, MAC, and network layers), and are thus covered by the current storage window; only long term (*e.g.*, watchdog) timers are thus stalled in the local queue.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8693
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8693

3.3 Synchronization Manager

The *synchronization manager* is an optional block within the application (*cf.* Figure 1) that corrects the time provided by the HW clock by providing a more accurate software (SW) time. This is achieved by using a synchronization protocol that corrects the SW time and estimates the current clock drift by retrieving information from neighboring nodes. The clock translator provides the interface for scheduling events in both HW and SW time. If a SW time interface is used, the clock translator requests the current time and drift estimate from the synchronization manager in order to schedule the event using the current drift estimation.

If synchronization is not required by the application, a "no synchronization" module is also available. In this case, the SW time is equal to the HW time and the estimated drift is always zero. It is thus suggested to always schedule events in SW time, except for debugging purposes. Our simple design provides the opportunity to easily test and implement new synchronization protocols: it is sufficient to provide an implementation of the synchronization manager that fulfills the given interface.

We integrated the available OpenCastalia implementation [6] for the popular Flooding Time Synchronization Protocol [3] in our framework as a protocol for the synchronization manager. With the old HW clock model it was not possible to evaluate such a synchronization protocol, since the constant drift leads to an unrealistic perfect synchronization.

3.4 Overhead Evaluation

In the previous sections, we discussed the benefits of using an accurate HW clock model and the new scheduling interface with a clock translator. If the following, we evaluate the overhead introduced in terms of execution time and memory consumption, by simulating four built-in Castalia applications ("connectivityMap", "simpleAggregation", "valuePropagation", and "valueReporting") both in the original version and with our extensions. We also added the constant drift model in our framework (similar to the one already available in Castalia), but with the difference that we use the new scheduling interface. We evaluate two different parameter sets of the combined model with different lengths of the approximation intervals t_{int} of 10 sec and 100 sec, which result in a model accuracy ε of 0.125 μsec and 12.5 μsec, respectively. It should be noticed that a maximum error $\varepsilon = 12.5$ μsec is sufficiently low for simulating a wide range of wireless sensor networks applications and MAC protocols.

Figure 3 shows the execution time of each application normalized with respect to the average execution time of the original simulator. The bars represent the average execution time whereas the error bars indicate the minimum and maximum of different runs. Comparing the first two bars clearly shows that the computation overhead introduced by the new interface is negligible. The average overhead introduced by the accurate HW clock model (bars 3 and 4) is about 11%, which is explained by the more complex translation of the HW time. It can be noticed that the overhead is almost independent of the window size, which indicates that the most expensive part for the new HW clock is the translation and not the update of the sliding window.

The accurate HW clock model requires additional memory for the storage window. Every node stores s entries, *i.e.*, one entry for each interval of the current window. Each entry contains the two double values ρ_k and h_k (8 Bytes each).

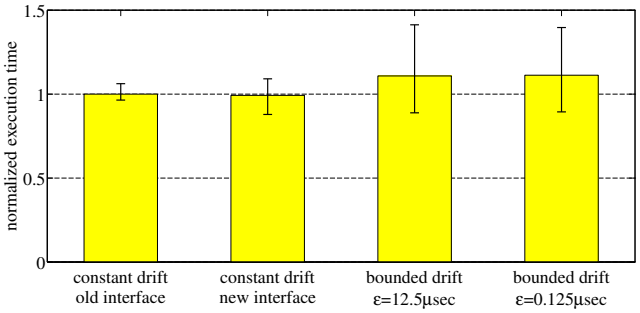

Figure 3: Execution time overhead.

If N is the number of nodes being simulated, the memory overhead is thus equal to $16Ns$ Bytes. As an example, a simulation with $N = 150$ nodes and a window size $s = 1000$ results in 2400 KBytes of additional memory, which is negligible under most simulation settings.

4. CONCLUSIONS

In this paper we introduced several extensions for WSNs simulators that allow an accurate timing analysis of time-critical parts of WSN applications, such as time synchronization and MAC protocols. In particular, we proposed a framework that provides realistic clock models, a well defined interface that abstracts the notion of simulation time from the user, and an optional synchronization manager that can be used for drift estimation and compensation. Within this framework, we implemented a realistic clock model that assumes bounded drift and bounded drift variation. This clock model has proven to be suitable to evaluate time-critical protocols like FTSP and MAC. It introduces minimal overhead in terms of memory and execution time in all tested Castalia applications. Our framework is very flexible and can readily be extended to any other clock model that can be approximated using a piecewise linear function, and to other network simulators.

5. ACKNOWLEDGMENT

The work presented in this paper was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

6. REFERENCES

[1] A. Boulis. Castalia. `http://castalia.npc.nicta.com.au/index.php`.

[2] A. Boulis. Demo Abstract: Castalia: Revealing Pitfalls in Designing Distributed Algorithms in WSN. In *SenSys'07*, New York, USA, 2007.

[3] M. Maróti et al. The flooding time synchronization protocol. In *SenSys'04*, New York, USA, 2004.

[4] K. Römer, P. Blum, and L. Meier. *Sensor Networks*, chapter Time Synchronization and Calibration in WSNs. John Wiley & Sons, New York, jul 2005.

[5] T. Schmid. OpenCastalia. `http://projects.nesl.ucla.edu/~thomas/opencastalia.html`.

[6] T. Schmid et al. Temperature compensated time synchronization. *IEEE Embedded Systems Letters*, 2009.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8693
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8693