FLOPSYNC-2: efficient monotonic clock synchronisation

Federico Terraneo*, Luigi Rinaldi*, Martina Maggio†, Alessandro Vittorio Papadopoulos†, Alberto Leva*
*Politecnico di Milano, Milano, Italy
†Lund University, Sweden

***Abstract*—Time synchronisation is crucial for distributed systems, and particularly for Wireless Sensor Networks (WSNs), where each node is executing concurrent operations to achieve a real-time objective. However, synchronisation is quite difficult to achieve in WSNs, due to the unpredictable deployment conditions and to physical effects like thermal stress, that cause drifts in the local node clocks. As a result, state-of-the-art synchronisation schemes do not guarantee monotonicity of the nodes clock, or are relying on external hardware assistance. In this paper we present FLOPSYNC-2, a scheme to synchronise the clocks of multiple nodes in a WSN, requiring no additional hardware, and based on the application of control-theoretical principles. The scheme guarantees low overhead, low power consumption and synchronisation with clock monotonicity.**

We propose an implementation of FLOPSYNC-2 on top of the microcontroller operating system Miosix, and prove the validity of our claims with several-days-long experiments on an eight-hop network. The experimental results show that the average clock difference among nodes is limited to a hundred of ns, with a sub-μs standard deviation. By introducing a suitable power model, we also prove that synchronisation is achieved with a sub-μA consumption overhead.

I. Introduction

Synchronising the clocks of different nodes is a problem that dates back to the advent of distributed systems [7, 8]. A network of nodes needs synchronisation whenever many events should be ordered and processed depending on their arrival times or, more in general, whenever the nodes should coordinate towards a common objective. In Wireless Sensor Networks (WSNs) nodes coordination is crucial, especially when real-time operations must be executed.

However, it is especially difficult to obtain and maintain synchronisation in WSNs. This may be due to jitters and uncertainty associated with the arrival of external timing information. Protocols may repeat the synchronisation packets transmission to obtain better accuracy, however at the expense of a higher energy consumption, which is undesirable for battery powered devices. Further difficulties come from the deployment conditions. Nodes may be deployed in extreme environments, like an active volcano [17], where thermal stress could cause significant clock drifts, and the number of synchronisation messages transmitted should be minimised. Even in not so extreme cases, ageing effects and manufacturing limitations can harm synchronisation.

This work was partially supported by the Swedish Research Council (VR) for the projects "Cloud Control" and "Power and temperature control for large-scale computing infrastructures", and through the LCCC Linnaeus and ELLIIT Excellence Centers.

The mentioned difficulties caused synchronisation schemes to neglect guarantees on clock monotonicity [2, 4, 6, 10, 12, 16, 18] or to rely on support, either from the power lines [14] or from external hardware [1].

We build on FLOPSYNC [9], our first attempt to introduce control theory for synchronisation. FLOPSYNC is based on a proportional-integral (PI) controller and can only compensate for a constant or slowly varying clock skew. In this paper we overcome this limitation by introducing FLOPSYNC-2, a message passing synchronisation scheme that on the contrary relies on a tailored controller structure, and requires no purpose-specific hardware. Different from the majority of the literature contributions, the sent packets do not contain any data and synchronisation is achieved by checking their actual arrival times against those expected by the recipient nodes. To guarantee clock monotonicity, nodes never reset the values of their internal clocks, as this could lead to potential backward or forward jumps in time. The tailored control scheme is used to select a clock correction that is introduced *along* the interval between the reception of two subsequent synchronisation packets. We theoretically prove that we achieve error convergence and clock monotonicity, without external support.

To verify our claims, we implemented FLOPSYNC-2 using the Glossy flooding scheme [5] on top of the microcontroller operating system Miosix, which is released as open source software and available for download. We performed many experiments to certify that the nodes' clocks are synchronised, including a test with an eight-hop network deployed in an office building, and also outdoor ones. In all cases nodes had to keep their clock synchronised for several days. Our results indicate that the average difference between the WSN nodes' clocks is of the order of 100ns, with a sub-μs standard deviation, even when the nodes are deployed in an environment where the temperature is time varying and environmental conditions can influence the nodes' behaviours. Also, our synchronisation scheme has a very low energy overhead. To support this claim we used a suitable power model to compute the amount of power consumed for the synchronisation phase, both for the radio module of each node and for the CPU. The current consumed for the purpose of synchronisation is less than 1μA.

II. Methodology

Our target WSN is composed by a master node and a certain number of slaves. The aim of the synchronisation

scheme is to align the clocks of the slave nodes to the master one. The master node periodically sends synchronisation packets by means of a flooding scheme like Glossy [5], with period T. Therefore, we can assume that medium access contention does not introduce transmission time uncertainty.

The key innovative idea of the presented research is that the master node does not send timestamps and the slave synchronisation is based on the expected and actual arrival time of synchronisation packets. If the expected arrival time matches the effective one, the clock of the slave is synchronised with the master. Otherwise, the slave node applies a correction that depends on the difference between the two mentioned times.

The initial offset between the slave and the master node should be eliminated. Indeed, initialisation is the only moment when the master node sends a timestamp. When a slave joins the network, it asks the master for the reference time t^0 corresponding to the transmission of the *next* synchronisation packet, and for the value of T. Upon reception, the slave waits until the next synchronisation packet arrives and initialises its local clock to t^0. This zeroes the offset and synchronises the slave with the master.

To maintain the clock synchronised, the slave node behaves as follows. At the reception of each k-th synchronisation packet, the slave measures the synchronisation error as the difference between its expected and actual arrival time, both counted in the local clock t_{loc}. Based on this error, the slave computes an additive correction $u(k)$, attempting to match a span of $T+u(k)$ in the local time to a span of T in the reference one. As a last operation, the slave sets the expected local time for the next packet, i.e., it waits in its local time for $T+u(k)$ instead of T. The slave iterates this process upon receipt of each synchronisation packet. Note that all the slave computations are in the local time, and no timestamp transmission or local clock reset is required after initialisation, leaving the slave hardware clock uncorrected.

Finally, at any instant between the k-th and the $(k+1)$-th synchronisations, which we denote by introducing a fractional index $\Delta \in [0,1)$, the slave can obtain an estimate $\widehat{t}(k+\Delta)$ of the reference time as

$$\widehat{t}(k+\Delta) = t(k) + [t_{loc}(k+\Delta) - t_{loc}(k)] \cdot \frac{T}{T+u(k)} \quad (1)$$

where $t(k)$ is the (known) master transmission time for the k-th synchronisation packet, $t_{loc}(k)$ the local timestamp upon reception of the same packet, and $u(k)$ the correction term.

The computation of $u(k)$ can be done by solving a feedback control problem, tractable in the linear time-invariant framework. A tailored solution for the control problem is given in the following..

III. Synchronisation as a control problem

Assume, without loss of generality, that the origin of both the reference and the local clock are set at the transmission of the synchronisation packet corresponding to the initialisation of the slave clock at $k=0$, so that $t(0) = t_{loc}(0) = 0$.

Denote with f_o the nominal frequency of the slave node oscillator, and with $\delta_f(t)$ its variation over the reference time t. Irrespectively of the reason that caused $\delta_f(t)$, if no synchronisation action is taken, the offset evolves according to

$$o(t) := t - t_{loc}(t) = t - \int_0^t \frac{f_o + \delta_f(\tau)}{f_o} d\tau. \quad (2)$$

The slave acts by altering the expected time for the next synchronisation packet $t^e_{loc}(k+1)$ by a quantity $u(k)$, which means setting

$$t^e_{loc}(k+1) = t^e_{loc}(k) + T + u(k). \quad (3)$$

The expected arrival time $t^e_{loc}(k)$ and the actual arrival time $t^a_{loc}(k)$ for the k-th packet take the form

$$t^e_{loc}(k) = kT + \sum_{h=0}^{k-1} u(h), \quad t^a_{loc}(k) = kT + \int_0^{kT} \frac{\delta_f(\tau)}{f_o} d\tau. \quad (4)$$

Also, the slave measures the synchronisation error at the k-th period as the expected arrival time minus the actual one for the synchronisation packet, both measured with respect to the local clock, that is,

$$e(k) = t^e_{loc}(k) - t^a_{loc}(k). \quad (5)$$

Substituting (4) in (5) yields

$$e(k) = \sum_{h=0}^{k-1} u(h) - \int_0^{kT} \frac{\delta_f(\tau)}{f_o} d\tau, \quad (6)$$

which shows that synchronisation is exact when the sum over k of the corrections $u(k)$ equals the integral of the normalised frequency variation δ_f/f_o up to kT:

Expanding and rearranging, we then get

$$\begin{aligned} e(k+1) =& t^e_{loc}(k+1) - t^a_{loc}(k+1) \\ =& \sum_{h=0}^{k} u(h) - \int_0^{(k+1)T} \frac{\delta_f(\tau)}{f_o} d\tau \\ =& e(k) + u(k) - \int_{kT}^{(k+1)T} \frac{\delta_f(\tau)}{f_o} d\tau \\ =& e(k) + u(k) + d(k) \end{aligned} \quad (7)$$

which describes the evolution in time of the synchronisation error $e(\cdot)$ as a function of its past value, of $u(k)$, that assumes the role of the control signal, and of possible unexpected behaviours, captured by the term $d(k)$, which must be rejected.

Using the $\mathscr{Z}$-transform [20], where z^{-1} denotes the discrete-time unit delay, one can rewrite Equation (7) as

$$E(z) = P(z)\big(U(z) + D(z)\big), \quad P(z) := \frac{1}{z-1}. \quad (8)$$

Since $\delta_f(t)$ is caused by physical phenomena, obviously no synchronisation-related action can affect it. In principle,

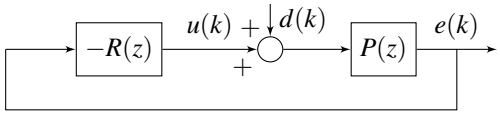

Figure 1: Feedback control scheme for the generic slave node.

one could use an arbitrarily complex model to take into account crystal imperfections, ageing, thermally induced frequency variations, oscillator nonlinearities, short-term jitter and any other possible cause of de-synchronisation. However, a controller can treat all the above as exogenous disturbances $d(k)$, while the controlled system given by Equation (8) is linear, time-invariant, device-independent, and uncertainty-free. This allows to provide guarantees on stability that do not depend on $d(k)$, and to tackle the synchronisation problem at the individual node level, in a completely decentralised manner.

IV. Control synthesis

In this Section we discuss the synthesis of the controller denoted by $R(z)$ in the scheme of Figure 1 (the error set point is identically zero, thus not reported for compactness). For feedback problems without uncertainty on $P(z)$, a convenient synthesis technique is to prescribe the closed-loop disturbance-to-error transfer function $E(z)/D(z)$ to match a desired one $F^o(z)$, which means practically to decide that the effect of the disturbance on the error should remain within certain bounds. From Figure 1,

$$\frac{E(z)}{D(z)} = \frac{P(z)}{1+R(z)P(z)} := F^o(z), \tag{9}$$

which readily provides $R(z)$ as

$$R(z) = \frac{P(z)-F^o(z)}{P(z)F^o(z)}. \tag{10}$$

A. Disturbance modelling

Given the device independence of model (8), $F^o(z)$ has to be chosen based on the expected disturbances. In this respect, $d(k)$ has four main components: crystal imperfections, ageing, short-term jitter, and thermal stress.

Crystal imperfections contribute a constant skew term, and ageing phenomena are extremely slow (acting typically on a time scale of days) with respect to any reasonable T. Both their effects thus vanish if $F^o(z)$ produces an output that tends to zero for any constant input; this requires $F^o(1) = 0$.

Short-term jitter is integrated over each synchronisation period, and the result of the integral contributes to $d(k)$. A control-theoretical approach to deal with this kind of problems, is to model the overall disturbance as a white noise, i.e., an unpredictable noise with a bounded variance. In this case, we can exploit a notable result in control-theory, which is related to the $\mathscr{H}_2$ norm of a system. The $\mathscr{H}_2$ norm measures the asymptotic variance of the response to white noise [20], i.e.,

$$\|F^o(z)\|_2^2 = \lim_{k\to\infty} \mathbb{E}\left[y(k)^2\right] - \mathbb{E}\left[y(k)\right]^2 \tag{11}$$

To minimise the effect of possible disturbances on the system's output, the $\mathscr{H}_2$ norm should be kept as small as possible. In this respect, the advantage of adopting FLOPSYNC-2 is threefold. First, the synchronisation-related information is moved from the content to the arrival time of the synchronisation packets, that can be made very short. Second, FLOPSYNC-2 uses a MAC-bypassing flooding scheme, thus there is no MAC-induced jitter in the synchronisation packets reception. Third, rebroadcasting is done in hardware: coupled to extremely predictable code paths, this practically eliminates software-induced jitter. Only two jitter terms thus remain, coming from oscillator and packet transmission.

The first term can be measured synchronising two nodes by wire. Experiments showed that for $T = 60\,\mathrm{s}$ the standard deviation was 610ns. The second term is seen at the transceiver interrupt line. In the experiments, we obtained a standard deviation of 50ns. Since the two jitter sources can be considered independent, the total uncertainty can be computed by summing their variances, yielding a total of 612ns. Hence, to keep the standard deviation of $e(k)$ below 1μs, $\|F^o(z)\|_2$ has to be less than 1.5. Of course, different hardware would give other results, but the characterisation procedure is straightforward. Moreover, the obtained $\|F^o(z)\|_2$ limit is not strict, as will emerge in the following, and it can be taken as a reasonable default for typical COTS-based WSN nodes.

The final source of disturbance is thermal stress. The time scale of this component is often comparable to the synchronisation period. Therefore, the choice of $F^o(z)$ needs to take this aspect into account. From this viewpoint, the harshest disturbance that a slave may face is a radiative heat rate step, which occurs for example in the case of a shade-sunlight transition [16]. We can describe a node as a single thermal capacity, that exchanges heat convectively with an external environment at the exogenously determined temperature θ_e, and is subject to a radiative thermal flux Φ_r. Hence, the dynamic behaviour of the node temperature, and of the crystal temperature θ_x, is ruled by the continuous-time linear model

$$\rho V c \frac{d\theta_x(t)}{dt} = S_r\Phi_r(t) - \gamma S_c(\theta_x(t) - \theta_e(t)) \tag{12}$$

where ρ, V, c are respectively the node volume, average density and average specific heat, S_r is the radiated surface, S_c that involved in convective heat exchange, and γ the convective heat transfer coefficient. Assume that a Φ_r step from 0 to $\overline{\Phi}_r$ is applied at $t = 0$, while θ_e stays constant at $\overline{\theta}_e$ and the node is initially at the equilibrium with the environment, i.e., $\theta_x(0) = \overline{\theta}_e$. The asymptotic temperature

reached by the node is

$$\overline{\theta}_x = \overline{\theta}_e + \frac{S_r}{\gamma S_c}\overline{\Phi}_r, \tag{13}$$

and the resulting transient is

$$\theta_x(t) = \overline{\theta}_e + (\overline{\theta}_x - \overline{\theta}_e)\left(1 - e^{-\frac{\gamma S_c}{\rho V c}t}\right), \tag{14}$$

while the maximum temperature variation rate is

$$\left.\frac{d\theta_x(t)}{dt}\right|_{t=0} = \frac{\gamma S_c}{\rho V c}(\overline{\theta}_x - \overline{\theta}_e) = \frac{S_r}{\rho V c}\overline{\Phi}_r. \tag{15}$$

Consider now a parallelepiped node with

- a width and length from 3 to 10cm,
- a height from 1 to 2cm,
- an average density from 700 to 1200kg/m^3,
- an average specific heat from 400 to 800J/kg°C,
- a heat exchange coefficient from 5 (still air) to 50 (medium-strong wind) W/m^2 °C.

Assume S_c to be the total surface and S_r the front one only, and take $\overline{\Phi}_r = 500$W/m^2 to consider an average sun inclination. The result is an asymptotic temperature from about 3 to 30°C above the external one, a transient duration of $2-80$min, and a maximum temperature rate between 1.5 and 10°C/min.

With a period T from a few seconds to a few minutes, and considering the parabolic temperature-to-frequency crystal characteristics, the thermal contribution $d_\theta(k)$ to $d(k)$ can steadily increase for several periods from the heat rate step, maintaining however a ramp-like shape if viewed on the horizon of a few periods. Therefore, since it is advisable to have the error approach zero before the oscillator temperature settles, $F^o(z)$ has asymptotically yield zero output also for any constant-*rate* input; hence, it has to contain *two* unity zeroes.

B. Controller structure selection

Summarising, we need an $F^o(z)$ with one unity zero to drive to zero the error (two if thermal stress is relevant at the synchronisation time scale, in the sense above) and an $\mathscr{H}_2$ norm below 1.5. We thus selected two transfer functions, to be used cooperatively as explained below, namely

$$F_1^o(z) = \frac{z-1}{z^2}, \qquad F_2^o(z) = \frac{(z-1)^2}{(z-\alpha)^3}, \quad 0<\alpha<1, \tag{16}$$

which respectively correspond, as per (10), to the controllers

$$R_1(z) = \frac{2z-1}{z-1}, \; R_2(z) = \frac{3(1-\alpha)z^2 - 3(1-\alpha^2)z + 1 - \alpha^3}{(z-1)^2}. \tag{17}$$

Figure 2 shows the $\mathscr{H}_2$ norms of $F_1^o(z)$ and $F_2^o(z)$, together with the error responses they produce for a unit step and a unit-slope ramp disturbance $d(k)$, applied at $k=0$. When considering $F_2^o(z)$, time responses were obtained with three representative values of α, commented on shortly.

As can be seen from the leftmost plot, $\|F_1^o(z)\|_2$ has an acceptable value, and the same is true for $\|F_2^o(z)\|_2$ in a wide enough span of α. Thus, both controllers perform satisfactorily in the face of jitter. Observing the other two plots, however, the same controllers exhibit a different behaviour in the presence of a skew that varies abruptly and then sustains the new constant value (centre plot, step response) or that increases for a long time with respect to T (rightmost plot, ramp response). Controller $R_1(z)$ is better when it is to eliminate a step-like skew, while in the ramp case it structurally cannot achieve error convergence while the skew is increasing. This means that in the presence of a heat rate step – that is, of a skew *ramp* – with $R_1(z)$ the error will start converging toward zero only when the skew becomes constant, which is when the temperature has settled. With $R_2(z)$, convergence is achieved also with a ramp-like skew. The rejection of a step-like skew is worse, however, and large values of α can cause the ramp response not to start converging before a certain number of steps.

As a result, we choose to employ $R_1(z)$ for the first two periods after the slave initialised its clock, to rapidly compensate for a skew that at the beginning is totally unknown, and then switch to $R_2(z)$. Moreover, we set $\alpha = 3/8$ – i.e., a value that eases computations to the advantage of speed – since this gives more or less the same $\mathscr{H}_2$ norm as $R_1(z)$, see the points marked in the leftmost plot of Figure 2, and only slightly deteriorates the error convergence in the ramp disturbance case.

FLOPSYNC-2 does not operate by explicitly estimating the skew. However, since $d(k)$ is the offset accumulated over one synchronisation period, $d(k)/T$ is readily interpreted as the average skew over that period. Also, since $u(k)$ is computed attempting to match $T+u(k)$ in the local time to T in the reference one, the quantity $-u(k)/T$ is in fact an estimate of the average skew just mentioned. Hence, the transfer function from the real (average) skew to its estimate takes the form

$$S(z) = \frac{-U(z)/T}{D(z)/T} = \frac{R(z)P(z)}{1+R(z)P(z)} \tag{18}$$

that with $R_2(z)$ becomes

$$S_2(z) = \frac{(1-\alpha)(3z^2 - 3\alpha z - 3z + \alpha^2 + \alpha + 1)}{(z-\alpha)^3} \tag{19}$$

To evidence the advantages inherently yielded by such a way of estimating the skew, we can look at a typical shade-sunlight transient like that shown in Figure 3. The crystal has a nominal frequency of 32kHz at 25°C, and a temperature coefficient of -0.035ppm/°C^2. Its temperature undergoes an exponential transient of amplitude and duration of about 20°C and 20min (topmost plot). The real skew has the behaviour shown by the dotted black line in the centre plot, descending toward zero at the nominal frequency temperature, and then increasing to the asymptotic value.

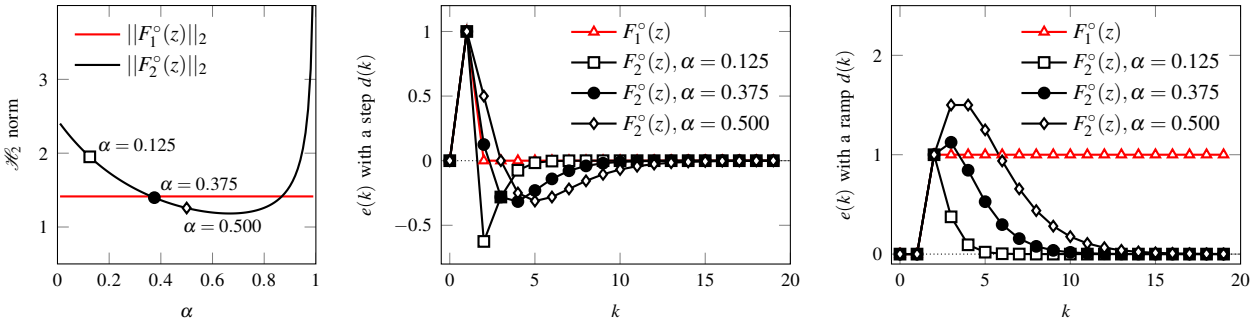

Figure 2: $\mathscr{H}_2$ norms and time responses of $F_1^o(z)$ and $F_2^o(z)$.

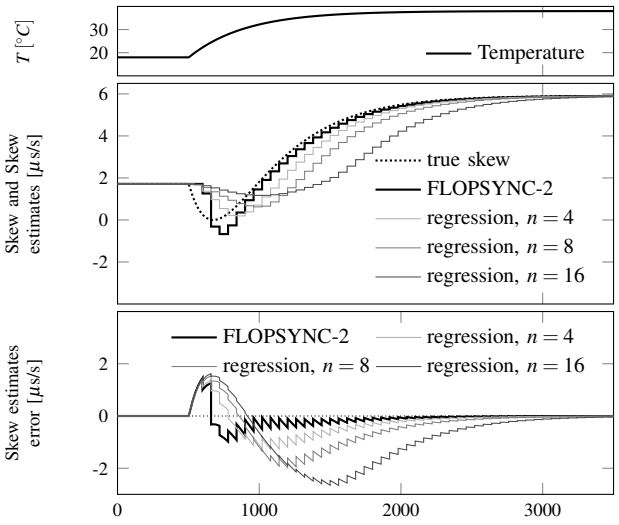

Figure 3: Skew estimates during a temperature transient.

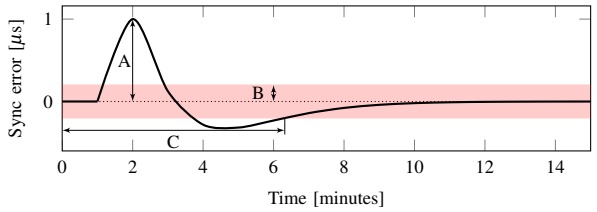

Figure 4: The FLOPSYNC-2 design requirements.

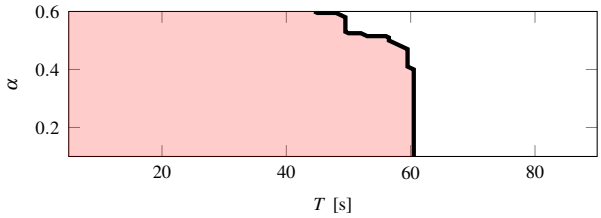

Figure 5: Feasible (T, α) and Pareto-optimal frontier.

The gray lines represent regression-based skew estimates with a period of 1 minute and a window of respectively 4, 8, and 16 past samples. The thick line is the skew estimated by FLOPSYNC-2, as per (19) with $\alpha = 3/8$. The bottom plot finally shows the skew estimate errors.

We could just notice that FLOPSYNC-2 is faster at recovering a good estimate, but there is a more general and important remark. In the real life of a WSN node, there is simply no such thing as a "true and constant skew to be estimated". Skew is influenced by thermal dynamics whose inputs (in this case, radiative heat) can vary with largely unpredictable amplitude and rate. This makes synchronisation techniques using regression-based skew estimation structurally weak, whence the numerous (and often complex) workarounds proposed in the literature [15]. Estimating the skew based on error feedback naturally leads to follow its dynamics without these workarounds.

C. Parametrisation

Suppose that a WSN needs deploying in a place where accessing the nodes is impossible, like the volcano of [17], and must operate correctly for a certain mission duration. In such a case, the choice of the synchronisation period is critical. Correct operation requires good synchronisation, which given the harsh thermal conditions, calls for a small T. Attaining the mission duration requires on the contrary to preserve the battery life of the nodes, thus claiming for as large a T as possible. In the absence of a method to configure the WSN *before* its deployment, heuristics can easily lead to excessive conservatism in either direction. As a result, operation may be correct but batteries can die too early, or batteries can last long enough, but synchronisation may be too poor. In both cases, the mission may fail-

Thanks to the use of dynamic models, FLOPSYNC-2 provides a solution to such problems. Increasing T prolongs battery life at the cost of a larger peak error for thermal events, while increasing α up to the value that gives the minimum $\|F_2^o(z)\|_2$ reduces jitter to the detriment of error convergence time. It is thus possible to perform an *offline* design space exploration with the proposed model; this yields the (T, α) couples that guarantee the required synchronisation quality under the assumed (worst-case) thermal stress, together with the Pareto-optimal values to target a WSN toward battery life, or minimum synchronisation error.

We created a configuration tool to easily perform the mentioned offline design space exploration. The user fills a form to provide the nominal crystal parameters, which

can be obtained from the data sheet, the expectable thermal stress, i.e.,

- a minimum external temperature $\theta_{e,min}$ and a maximum one $\theta_{e,max}$, such as the minimum temperature during winter and the maximum during summer for outdoor applications,
- a maximum magnitude $r_{\theta,max}$ for the temperature variation rate,
- a maximum magnitude $\Delta\theta_{max}$ for the temperature swing in a single thermal event, as it is very unlikely that said swing spans the entire $(\theta_{e,min}, \theta_{e,max})$ range,

and the synchronisation requirements. Given the shape of a typical error trace during a temperature transient with FLOPSYNC-2, the tool allows to select the maximum tolerable error peak magnitude after a thermal event (A in Figure 4) and the maximum time C for the error to fall below a prescribed range of values centred around zero with amplitude $B < A$ after such an event. The user is then presented with an output similar to Figure 5. The coloured region contains the feasible (T, α) couples, and the thick black line is the Pareto-optimal frontier.

The tool was validated by comparing its results with several experiments made with real WSN nodes recording the synchronisation error during thermal transients of various amplitude and rate, resulting in less than 20% errors in the maximum synchronisation error prediction. This result was achieved despite only using the *nominal* crystal parameters taken from its data sheet.

FLOPSYNC-2 guarantees stability and error convergence to zero *structurally*. Thanks to the robustness of the feedback loop, the sole consequences of incorrect couples of (T, α) is a larger error peak and/or a slower convergence. In "standard" applications, thus, one can use the proposed default values safely, and no configuration effort is required. In more critical cases, the sole knowledge (also approximate) of nominal parameters and environmental limit conditions, can be used to make sure *a priori* that the WSN will behave correctly, and will not loose performance due to poor synchronisation or waste energy due to excessive conservatism.

V. Implementation

FLOPSYNC-2 is made of four components: the flooding scheme, the controller, the virtual clock, and the resynchronisation scheme. The first two are implemented as periodic tasks with period T. The virtual clock is called every time the reference time is needed, while the resynchronisation module is executed when a slave joins the network, or loses synchronisation, for example due to a long-lasting interference that prevents the reception of several synchronisation packets.

A. The flooding scheme

The purpose of the flooding scheme is to receive and rebroadcast the synchronisation packets. It also measures their actual arrival time $t^a_{loc}(k)$, suitably corrected for the transmission delay. This is easily computed based on packet length, hop count, and radio data rate. Since in FLOPSYNC-2 synchronisation information is given by packet arrival times, the radio channel must be clear from access contention when synchronisation packets are flooded. Therefore, all the nodes turn off their MAC layer for a short time window every period T, and the flooding scheme takes full control of the transceiver. No further constraint is imposed on the radio usage outside the synchronisation window: a node can use an ordinary low power listening MAC, for example.

FLOPSYNC-2 rebroadcasts synchronisation packets in hardware, to eliminate software-induced retransmission jitter. The used hardware timer has an *input capture* channel, as in [5], but also an *output compare* one. The capture channel is connected to the transceiver *SFD*[1] interrupt line, thereby taking a local clock timestamp of the synchronisation packet arrival. A constant delay is then added in software to the timestamp, and the resulting value is written in the output compare register. The output compare feature allows to raise a pin on the microcontroller when the preset time is reached, thereby triggering the packet transmission in hardware. This allows to introduce a very deterministic rebroadcast delay. As the following experiments will show, this makes the synchronisation error nearly independent of the number of hops.

Finally, FLOPSYNC-2 has no requirement on the content of the synchronisation packet, which enhances power efficiency by reducing the consumption overhead at both the transmitter and the receiver end. As an example, the Glossy flooding scheme requires a one byte field in the packet used as retransmission counter. This allows to flood the network with a packet whose payload is just one byte. To the best of the authors' knowledge, no other scheme can work with such small packets. If the radio transceiver additionally supports variable length packets, *piggybacking* can be used to send commands to the entire network with the same periodic flooding used for synchronisation, while reducing the packet length to one byte when there are no commands to transmit.

B. The FLOPSYNC-2 controller

The purpose of the FLOPSYNC-2 controller is to compute $u(k)$, thus the expected arrival time $t^e_{loc}(k+1)$ for the next synchronisation packet: $t^e_{loc}(k+1)$ allows to schedule the next time when the flooding scheme will take over the radio, while $u(k)$ is used by the virtual clock. This computation is done by $R_2(z)$ except for the first two steps, where $R_1(z)$ runs. The controllers can be implemented efficiently with integer arithmetic, particularly if α is chosen as a rational number having a power of two as denominator.

[1] SFD or Start Frame Delimiter is the byte that marks the beginning of a packet in the 802.15.4 standard. Most radio transceivers have an interrupt line that is raised when it is received.

FLOPSYNC-2 also minimises idle listening, by adapting the time advance w of the radio activation with respect to the expected synchronisation packet arrival. This is crucial for power efficiency, and critical in terms of power/reliability tradeoff. A too short w results in a high risk of missing synchronisation packets, while a too long one wastes power [19].

To adapt w, the algorithm computes the error variance in batches of $N = 8$ error samples, by keeping only two variables (the sum of the errors and of their squares). The variance is used to compute the standard deviation σ, and w is set to 3σ, clamped between a minimum and a maximum (in the current implementation, 30μs and 5ms). This results in adapting w every $N \cdot T$ seconds. Note that the so computed w can be used by the radio stack for additional power optimisations, for example to efficiently implement slotted or TDMA MAC protocols.

As soon as the packet is received, the receiver is turned off, and the flooding scheme rebroadcasts it. If instead the packet is not received, the radio is kept in receive mode for a maximum of $2w + packetTime$, where $packetTime$ is the time necessary for the packet transmission. After that, the packet is considered lost, w is doubled, and since this lack of information does not allow to run the controller, the u value of the previous period is used also for the present one. Finally, a counter is incremented every time a synchronisation packet is lost, and reset to zero when one is received. If the counter exceeds a threshold (3 in the current implementation), the slave node undergoes resynchronisation.

The FLOPSYNC-2 controller, with dynamic w adaptation, is reported in Listing 1 to demonstrate its simplicity. When implemented for the reference architecture, only 28 bytes of RAM (plus a few ones of stack) are needed, and the full code, including the part handling packet losses not shown here for brevity, occupies only 604 bytes. Concerning execution speed, the code takes on average 4.4μs, increasing to 7.2μs when w is recomputed.

```
int uo = 0, uoo = 0;          // past control values
short eo = 0, eoo = 0;        // past errors
int sum = 0, squareSum = 0; // variance computation data
char count = 0, init = 0;     // counter and pre-init
unsigned short w = wMax;      // idle listening time
unsigned int eat = period;  // expected arrival time

while (1) {                   // each synchronisation k
 disableMacLayer();
 unsigned int timeout = (2*w) + packetTime;
 waitForSync(timeout);
 int at = rtc.getValue();    // actual arrival time
 rebroadcastWithGlossy();
 short e = eat - at;          // error on arrival time
 if (init == 0) {
   // quick skew estimation controller
   init = 1; uo = -2 * 512 * e; uoo = -512 * e;
 } else {
   if (init == 1) {
     init=2; uo/=2; // transition to second controller
   }
   // controller: alpha = 3/8, multiplied by 512
   int u = (2*uo) - uoo - (960*e) + (1320*eo) - (485*eoo);
   uoo = uo; uo = u; eoo = eo; eo = e;
   // idle listening minimisation
   sum += e; squareSum += e * e; ++count;
   if (count >= numSamples) {
     // variance computed as E[X^2]-E[X]^2
     int mean = sum / numSamples;
     int var = (squareSum/numSamples) - (mean*mean);
     // using the Babylonian method for square root
     int stddev = var/7;
     for (int j=0; j<3; j++)
       stddev = (stddev + var/stddev)/2;
     // set the slack time to 3 sigma
     w = stddev * 3;
     if (w > wMax) w = wMax;
     if (w < wMin) w = wMin;
     sum = 0; squareSum = 0; count = 0;
   } // end if
 } // end for
 eat += period + uo/512; // updating expected arrival time
 enableMacLayer();
 rtc.sleepUntil(eat - w);
}
```

Listing 1: FLOPSYNC-2 controller and variance estimator.

C. The virtual clock

The purpose of the virtual clock is to translate from the local time to the reference one, and *vice versa*. The translation takes as input the value of t_{loc} to convert as well as $t(k)$, the reference time when the last synchronisation packet was sent, the local time $t^e_{loc}(k)$ when the same packet was expected, and $u(k)$. Time translation is done by rewriting t_{loc} as $t^e_{loc}(k) + (t_{loc} - t^e_{loc}(k))$. The first term is converted into the reference time by replacing $t^e_{loc}(k)$ with $t(k)$, as this is the reference time when the last synchronisation packet was sent. The $t_{loc} - t^e_{loc}(k)$ term is then corrected with a linear interpolation employing $u(k)/T$ as the skew estimate (see Figure 3). The reference time estimate then comes from (1).

One can argue on the choice of $t^e_{loc}(k)$ instead of $t^a_{loc}(k)$, which is apparently a better estimate of the local time corresponding to $t(k)$. However, if $t^a_{loc}(k)$ were used, an instantaneous correction of the local clock would result, thereby impairing its monotonicity. On the contrary, the adopted choice makes the error constant across the boundary between two synchronisation periods, thus being the key point for its continuity and monotonicity.

D. The resynchronisation scheme

This component takes care of recovering from a synchronisation loss, as well as allowing a slave that has just joined the network to receive the first synchronisation packet. Its task consists of resetting the FLOPSYNC-2 controller's state variables to zero and initialising the idle listening time advance w to its maximum value.

Initial offset elimination is done with a timestamp request/response protocol, taking advantage of the ordinary MAC protocol used by the nodes when not running the flooding scheme — i.e., transmitting *sporadic* timestamps. To avoid any time uncertainties due to access contention in the request/response protocol, the response packet does not contain the current time, rather the reference time

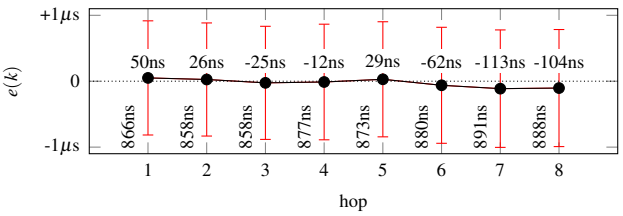

Figure 6: FLOPSYNC-2 average synchronisation error and standard deviation as a function of the hop count.

when the next synchronisation packet will be sent. As this occurs at a constant period T which is also sent in the response packet, the reference time corresponding to all future synchronisation packets can be determined. It should however be stressed that the reference time is only used by the virtual clock, while the synchronisation controller is fed by the difference between the expected and actual packet time only, both counted in the local clock t_{loc}.

VI. Experimental results

FLOPSYNC-2 has been implemented on a WSN node platform employing an ARM Cortex-M3 microcontroller running at 24MHz, and a CC2520 transceiver. The software is written in C++, as an application for the Miosix[2] operating system. The implementation uses Glossy for predictable flooding latency, and the *virtual high-resolution timer* (VHT) [16] without the skew compensation method, to achieve high local clock resolution with low consumption.

In all the reported experiments T is set to 60s, and α to $3/8$. To measure the error between synchronisation instants and test the virtual clock, the slaves send additional packets every 1.5s to the master. These packets are used only in performance evaluation tests and not in normal operation mode.

A. Synchronisation error distribution

To tests the statistical distribution of the synchronisation error, we performed a six-days-long experiment with nodes distributed in an office building, forming an eight-hop network, and exposed to interferences like *Wi-Fi* networks, to provide a realistic setting.

Figure 6 shows the average synchronisation error and its standard deviation as a function of the hop count. Observe that the average is extremely low, a few tenths of *nanoseconds*. For small hop counts, this is even less than the tick resolution of the employed timer, which is 42ns. The merit for so high a precision is due to the integral action in the FLOPSYNC-2 controller that can be viewed as a means to account for an infinite number of past error samples (opposite for example to the forcedly limited window of regression-based skew compensation). Also, both the average error and its standard deviation show

[2]The source code, including also the FLOPSYNC-2 configuration tool, is available for download as free software at http://gitorious.org/flopsync.

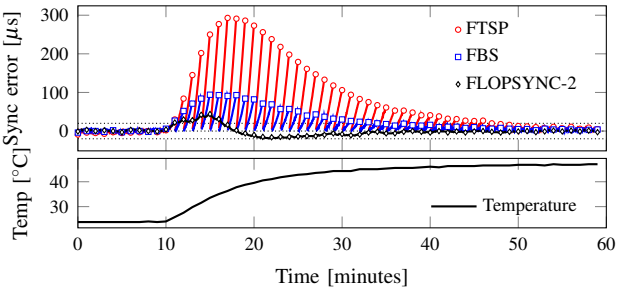

Figure 7: Synchronisation error during the temperature transient caused by a shade to sunlight transition.

a significantly weak dependence on the hop count: this is a merit of the hardware flooding protocol, that practically eliminates jitter accumulation from one hop to the next one.

An important fact to notice is that during the experiment no de-synchronisations were observed, although interferences did cause synchronisation packet losses. This proves that resynchronisations are infrequent, and therefore the timestamp request/response protocol for offset compensation outlined in section V-D is more efficient than including a timestamp in each synchronisation packet.

Finally, the time spent in idle listening was logged throughout the experiment, and on average it was 21 μs per period. This value will be used in Section VI-C to estimate the FLOPSYNC-2 consumption overhead.

B. Effects of temperature stress

One of the most extreme tests, yet common for real devices, that a clock synchronisation scheme has to withstand is the skew variation caused by a shade-sunlight transition. The capabilities of FLOPSYNC-2 are here compared with FTSP [10], a representative clock synchronisation scheme based on linear regression, and FBS [2] a representative example using non tailored control-based schemes. During the test, three nodes respectively running the mentioned synchronisation schemes, were placed in an enclosure and exposed to the sun. The enclosure has the double purpose of providing a realistic setting for a network deployed in an outdoor environment, and of keeping the temperature of the nodes as uniform as possible, so as to better compare their response. FTSP was configured with a window of 8 data samples, as in the TinyOS implementation, while FBS was configured with $Ki = Kp = 0.7847$, as suggested in [2].

The result is shown in Figure 7. Compared with FTSP, FLOPSYNC-2 performs significantly better: error feedback quickly compensates for the varying skew, achieving a maximum error of 45μs, while with FTSP the error rises up to 293μs. Also, FLOPSYNC-2 requires just 7 minutes to steer the error into a $\pm 20\mu$s range, while FTSP requires 38 minutes.

In comparison with FBS, FLOPSYNC-2 reduces the synchronisation error to $\pm 20\mu$s within seven minutes and keeps the error in the same range even *while temperature is*

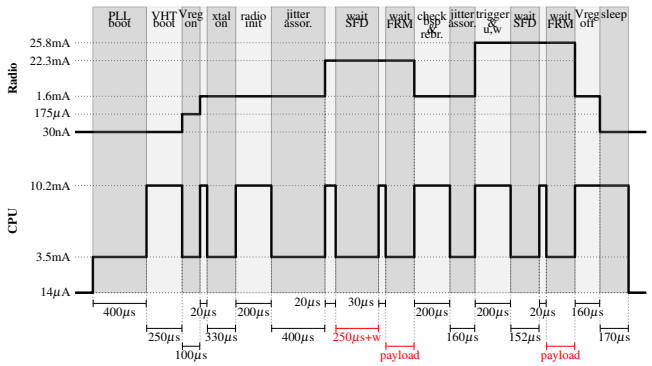

Figure 8: Current consumption trace for a slave node.

still increasing. On the contrary, the PI controller of FBS exhibits a higher maximum error of 94μs and does not adequately compensate for the error until the temperature has settled. As a result, FBS takes 28 minutes to bring the error in the same $\pm 20\mu$s range. As for clock monotonicity and continuity, FTSP and FBS rely on timestamp transmission and clock corrections. Although when temperature is not changing FTSP and FBS can accommodate for a consequently constant skew, this is not true during a temperature change, where instantaneous clock jumps are observed. These jumps may cause backward corrections of the local clock. FLOPSYNC-2, instead, guarantees clock monotonicity.

C. Energy consumption model

This Section shows the FLOPSYNC-2 energy efficiency, introducing a consumption model. During the execution, we profiled the implementation, extracting the operating state (active, deep sleep, etc.) transitions for both the microcontroller CPU and the transceiver. The so obtained timing information, combined with current consumptions from data sheets, gives us an estimate of the current consumption.

Figure 8 shows the current consumption profile for a slave node. The master has a similar profile, except for the packet reception. At the beginning of each synchronisation period, the CPU wakes out of deep sleep, starts its internal PLL and enters the run state at the full clock frequency. It then performs VHT resynchronisation, and brings the transceiver out of deep sleep, which requires waiting for the transceiver voltage regulator and oscillator to start. The transceiver is then configured, and after that the radio and CPU remain both in the sleep state for some slack time, to absorb the jitter of the PLL and oscillator startup. Once the slack time is expired, the CPU instructs the radio to start receiving, sets a timeout interrupt, and sleeps. When the synchronisation packet is received, the CPU is awakened, and after a fixed amount of time, the timer compare channel triggers packet rebroadcast. While the packet is being sent the CPU runs the controller and variance estimator, and then sleeps. Finally, when the packet is sent, the transceiver enters deep sleep, shortly followed by the CPU.

The FLOPSYNC-2 average consumption overhead can be obtained by integrating the area below the current profile, dividing by T, and subtracting the consumption of the node in deep sleep. The result (parametric in the packet payload, w, and T) is

$$I_{ref} = \frac{25.6\mu\text{C} + 0.94\mu\text{C} \cdot payload\_bytes}{T} \tag{20}$$

for the master node, and

$$I_{sync} = \frac{37.8\mu\text{C} + 1.76\mu\text{C} \cdot payload\_bytes + 25.8\text{mA} \cdot w}{T} \tag{21}$$

for the generic slave. Summarising, with a two-byte payload (Glossy retransmission counter and checksum[3]), and the average value of w taken from Section VI-A, the current consumption overhead of FLOPSYNC-2 is 458nA for the master node, and 698nA for the slaves.

VII. Related work

A first pioneering work on WSN synchronisation is Post-facto synchronisation [3], that questions the applicability of established clock synchronisation schemes such as NTP [11] to WSNs. The proposed solution synchronises clocks only when needed by listening on a periodically transmitted synchronisation packet. Another approach is TPSN [6], that creates a spanning tree of the network, and uses pairwise node synchronisation along the edges. DMTS [12] enhances simplicity by having the reference node periodically broadcast a packet with its time. The other nodes compare their local time with the received one, and perform a correction when needed. The simplest approaches, like those sketched above, perform only clock synchronisation without skew compensation, thus the error grows rapidly as the synchronisation period is increased, and suffer from clock jumps at every synchronisation.

RBS [4] exploits the broadcast nature of wireless links by having each node record a timestamp when a packet is received. Nodes then exchange their timestamps over the radio, and compute a translation table between their clocks; notably, RBS introduces the use of linear regression to estimate clock skew. FTSP [10] proposes the use of a flooding scheme to efficiently distribute the synchronisation packet, coupled with MAC-level timestamping and linear regression for skew compensation. Tiny-Sync [18] proposes a different skew compensation scheme by constraining the possible skew values through a set of inequalities.

Subsequent research focuses on improving individual components used to build a synchronisation scheme. Glossy [5] proposes a particularly efficient flooding scheme for distributing synchronisation packets, which we build

[3]802.15.4 packets have a two byte CRC, but this is a high overhead for a one-byte payload packet. Thus, CRC was disabled when sending sync packets, and a one-byte checksum is added to the payload.

upon. TCTS [15] enhances the regression-based skew compensation of FTSP with temperature sensing and a temperature to frequency model, to withstand thermal drift, while our work can compensate for the same drift without measuring temperature. VHT [16] combines a low power, low frequency oscillator with a higher frequency one, to improve both resolution and power efficiency.

Although the problem of event ordering in distributed systems is well known in the literature [8], most of the mentioned algorithms are based on instantaneous clock synchronisation, followed by skew compensation. Such an approach neglects monotonicity and continuity of the local clock of each node, and causes event ordering issues even between timestamps of a single node. This is especially true when the skew estimate is not correct, such as during a temperature change. Works that address this problem by only changing the clock *rate* exist in the literature, such as [1, 13], but the first one relies on additional hardware to physically tune the crystal oscillator frequency, while the second uses timestamp transmission and a PLL-like algorithm.

This is not the first work that proposes the use of feedback control for clock synchronisation. Previous approaches use established control schemes instead of designing a tailor-made controller. Examples include FBS [2] and FLOPSYNC [9] which are based on proportional-integral (PI) control and can only compensate for a constant or slowly varying clock skew. Another example is SCTS [13], which uses a PLL control loop. While a PLL is specifically designed to perform clock synchronisation, it was originally meant to be operated on a cycle-by-cycle basis. When applied to WSN synchronisation, the algorithm is instead operated once every few million clock cycles, and temperature variations can no longer be considered a slowly varying disturbance.

Virtually all the proposed approaches use the difference between a local and a received timestamp as the synchronisation error measure. In these schemes, the synchronisation packets need to contain timestamps, which is inefficient considering that state-of-the-art synchronisation schemes achieve errors in the order of a microsecond. At these resolutions a 32 bit timestamp overflows too quickly, so a synchronisation packet may contain up to 8 bytes of timestamp, resulting in an overhead in both power and radio bandwidth.

VIII. Conclusion

This paper addressed the problem of efficient clock synchronisation for the nodes of a wireless sensor network. Synchronisation is a very challenging problem for WSNs, due to the unpredictable deployment conditions and to physical interference that cause drifts in the internal clock of each node. The proposed synchronisation scheme is based on a control-theoretical formulation and a tailored controller structure.

The scheme was implemented on top of a microcontroller operating system, demonstrating that FLOPSYNC-2 preserves clock monotonicity for each node without relying on any external support. FLOPSYNC-2 can be efficiently implemented, allowing nodes to achieve sub-μs synchronisation error with sub-μA current consumption overhead.

References

[1] M. Buevich, N. Rajagopal, and A. Rowe. "Hardware Assisted Clock Synchronization for Real-Time Sensor Networks". In: *IEEE 34th Real-Time Systems Symposium (RTSS)*. 2013.

[2] J. Chen, Q. Yu, Y. Zhang, H. Chen, and Y. Sun. "Feedback-Based Clock Synchronization in Wireless Sensor Networks: A Control Theoretic Approach". In: *IEEE Trans. on Vehicular Tech.* 59.6 (2010).

[3] J. Elson and D. Estrin. "Time synchronization for wireless sensor networks". In: *15th International Parallel and Distributed Processing Symposium*. 2001.

[4] J. Elson, L. Girod, and D. Estrin. "Fine-grained network time synchronization using reference broadcasts". In: *Symposium on Operation Systems Design and Implementation*. 2002.

[5] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. "Efficient network flooding and time synchronization with Glossy". In: *Information Processing in Sensor Networks (IPSN)*. 2011.

[6] S. Ganeriwal, R. Kumar, and M. Srivastava. "Timing-sync Protocol for Sensor Networks". In: *International Conference on Embedded Networked Sensor Systems*. 2003.

[7] H. Kopetz and W. Ochsenreiter. "Clock Synchronization in Distributed Real-time Systems". In: *IEEE Trans. Comput.* 36.8 (Aug. 1987), pp. 933–940.

[8] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* (1978).

[9] A. Leva and F. Terraneo. "Low power synchronisation in wireless sensor networks via simple feedback controllers: The FLOPSYNC scheme". In: *American Control Conference (ACC)*. 2013.

[10] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. "The Flooding Time Synchronization Protocol". In: *Conference On Embedded Networked Sensor Systems*. 2004.

[11] D. Mills. "Internet time synchronization: the network time protocol". In: *IEEE Trans. on Communications* 39.10 (1991), pp. 1482–1493.

[12] S. Ping. "Delay Measurement Time Synchronization for Wireless Sensor Networks". In: *Intel Research*. 2003.

[13] F. Ren, C. Lin, and F. Liu. "Self-correcting time synchronization using reference broadcast in wireless sensor network". In: *IEEE Wireless Communications* (2008).

[14] A. Rowe, V. Gupta, and R. Rajkumar. "Low-power Clock Synchronization Using Electromagnetic Energy Radiating from AC Power Lines". In: *Sensys*. 2009, pp. 211–224.

[15] T. Schmid, Z. Charbiwala, R. Shea, and M. Srivastava. "Temperature compensated time synchronization". In: *IEEE Emb. Sys. Lett.* (2009).

[16] T. Schmid, P. Dutta, and M. Srivastava. "High resolution, low-power time synchronization an oxymoron no more". In: *Information Processing in Sensor Networks (IPSN)*. 2010.

[17] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. "Deploying a wireless sensor network on an active volcano". In: *Internet Computing* 10.2 (2006), pp. 18–25.

[18] S. Yoon, C. Veerarittiphan, and M. Sichitiu. "Tiny-sync: Tight time synchronization for wireless sensor networks". In: *ACM Trans. on Sensor Networks* (2007).

[19] Y. Zeng, B. Hu, and H. Feng. "Time Division Flooding Synchronization Protocol for Sensor Networks". In: *International Conference on Mobile and Ubiquitous Systems: Networking Services*. 2007.

[20] K. Zhou, J. Doyle, and K. Glover. *Robust and Optimal Control*. Prentice Hall PTR, 1996. ISBN: 9780134565675.