Time synchronization

Objectives of this Chapter

Time is an important aspect for many applications and protocols found in wireless sensor networks. Nodes can measure time using local clocks, driven by oscillators. Because of random phase shifts and drift rates of oscillators, the local time reading of nodes would start to differ – they loose synchronization – without correction.

The time synchronization problem is a standard problem in distributed systems. In wireless sensor networks, new constraints have to be considered, for example, the energy consumption of the algorithms, the possibly large number of nodes to be synchronized, and the varying precision requirements.

This chapter gives an introduction to the time synchronization problem in general and discusses the specifics of wireless sensor networks. Following this, some of the protocols proposed for sensor networks are discussed in more detail.

Chapter Outline

8.1 Introduction to the time synchronization problem

In this section, we explain why time synchronization is needed and what the exact problems are, followed by a list of features that different time synchronization algorithms might have. We also discuss the particular challenges and constraints for time synchronization algorithms in wireless sensor networks [238, 239].

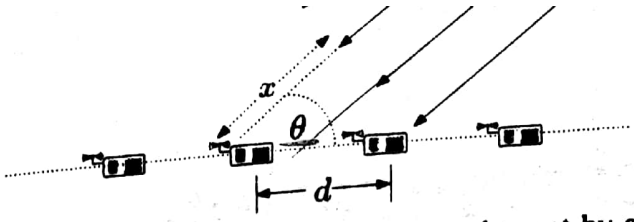


Figure 8.1 Determination of angle of arrival of a distant sound event by an array of acoustic sensors

8.1.1 The need for time synchronization in wireless sensor networks

Time plays an important role in the operation of distributed systems in general and in wireless sensor networks in particular, since these are supposed to observe and interact with physical phenomena.

A simple example shall illustrate the need for accurate timing information (Figure 8.1). An acoustic wavefront generated by a sound source a large distance away impinges onto an array of acoustic sensors and the angle of arrival is to be estimated. Each of the sensors knows its own position exactly and records the time of arrival of the sound event. In the specific setup shown in the figure, the angle θ can be determined when the lengths d and x are known, using the trigonometric relationship $x = d \cdot \sin\theta$, and accordingly $\theta = \arcsin \frac{x}{d}$.[1] The sensor distance d can be derived from the known position of the sensors and the distance x can be derived from the time difference Δ_t between the sensor readings and the known speed of sound $c \approx 330$ m/s, using $x = c \cdot \Delta_t$. Assuming $d = 1$ m and $\Delta_t = 0.001$ s gives $\theta \approx 0.336$ (in radians). If the clocks of the sensors are only within 500 µs accurate, the true time difference can be in the range between 500 and 1500 µs, and thus the estimates for θ can vary between $\theta \approx 0.166$ and $\theta \approx 0.518$. Therefore, a seemingly small error in time synchronization can lead to significantly biased estimates.

There are at least two ways to get a more reliable estimate. The first one (and the one focused on in this chapter) is to keep the sensors clocks as tightly synchronized as possible, using dedicated **time synchronization algorithms**. The second one is to combine the readings of multiple sensors and to "average out" the estimation errors. There are many other applications requiring accurate time synchronization, for example, beamforming [237, 856, 907].

However, not only WSN applications but also many of the networking protocols used in sensor networks need accurate time. Prime examples are MAC protocols based on TDMA or MAC protocols with coordinated wakeup, like the one used in the IEEE 802.15.4 WPAN standard [468]. Sensor nodes running a TDMA protocol need to agree on boundaries of time slots; otherwise their transmissions would overlap and collide.

It is important to note that the time needed in sensor networks should adhere to **physical time**, that is two sensor nodes should have the same idea about the duration of 1 s and additionally a sensor node's second should come as close as possible to 1 s of **real time** or **coordinated universal time (UTC)**.[2] The physical time has to be distinguished from the concept of **logical time** that allows

[1] To keep the example simple, we assume that further sensors provide the information necessary to distinguish between the angles θ and $2\pi - \theta$.

[2] The duration of a second is precisely defined in terms of the number of transitions between the two hyperfine levels of the ground state of Caesium-133 [643] and this information is provided by a number of atomic clocks spread around the world. The operators of these clocks work together to provide the international UTC time scale. The UTC time does not use the atomic times directly but occasionally seconds are inserted or deleted to keep the UTC time in synch with astronomical timescales as they are used for navigation purposes. See the website of your favorite standardization institute, for example, `www.ptb.de` or `www.nist.gov`. Many countries have (sometimes several) local times that are based on UTC; for example, the central European daylight saving time CEDST is just UTC plus 2 h. These local times are transmitted as reference times.

to determine the ordering of events in a distributed system [464, 526] but does not necessarily show any correspondence to real time.

Many time synchronization algorithms have appeared over time; overviews and surveys of algorithms developed for classical distributed systems can be found in references [24, 179, 677]. We restrict the discussion to algorithms relevant for wireless sensor networks, that is, algorithms that care about their energy consumption and which can run in large-scale networks.

8.1.2 Node clocks and the problem of accuracy

Almost all clock devices of sensor nodes and computers share the same common structure [24, 179]: The node possesses an **oscillator** of a specified frequency and a **counter register**, which is incremented in hardware after a certain number of oscillator pulses. The node's software has only access to the value of this register and the time between two increments determines the achievable **time resolution**: events happening between two increments cannot be distinguished from their timestamps.

The value of the **hardware clock** of node i at real time t can be represented as $H_i(t)$. It can be understood as an abstraction of the counter register providing an ever-increasing time value. A common approach to compute a local **software clock** $L_i(t)$ from this value is to apply an affine transformation to the hardware clock:[3]

$$L_i(t) := \theta_i \cdot H_i(t) + \phi_i.$$

ϕ_i is called **phase shift** and θ_i is called **drift rate**. Given that it is often neither possible nor desirable to influence the oscillator or the counter register, one can change the coefficients θ_i and ϕ_i to do **clock adjustment**. Now we can define the notion of **precision** of the clocks within a network, where we distinguish two cases:

External synchronization The nodes $1, 2, \ldots, n$ are said to be **accurate** at time t within a bound δ if

$$|L_i(t) - t| < \delta$$

holds for all nodes $i \in \{1, 2, \ldots, n\}$.

Internal synchronization The nodes $1, 2, \ldots, n$ are said to **agree** on the time with a bound of δ if

$$|L_i(t) - L_j(t)| < \delta$$

holds for all $i, j \in \{1, 2, \ldots, n\}$.

To achieve external synchronization, a reliable source of real time/UTC time must be available, for example, a GPS receiver [401]. Clearly, if nodes $1, 2, \ldots, n$ are externally synchronized with bound δ, they are also internally synchronized with bound 2δ.

There are three problems:

- Nodes are switched on at different and essentially random times, and therefore, without correction, their initial **phases** ϕ_i are random too.

[3] We adopt the convention to write real times with small letters (for example, t, t') and values of local clocks with capital letters. For simplicity, we do not consider overruns of the counter register.

- Oscillators often have a priori a slight random deviation from their nominal frequency, called **drift** and sometimes **clock skew**. This can be due to impure crystals but oscillators also depend on several environmental conditions like pressure, temperature, and so on, which in a deployed sensor network might well differ from laboratory specifications. The clock drift is often expressed in **parts per million (ppm)** and gives the number of additional or missing oscillations a clock makes in the amount of time needed for one million oscillations at the nominal rate. In general, cheaper oscillators – like those used in designs for cheap sensor nodes – have larger drifts with higher probability. In several publications from the field of wireless sensor networks, clock drifts in the range between 1 and 100 ppm are assumed [236, 839]. For Berkeley motes, the datasheets specify a maximum drift rate of 40 ppm. A deviation of 1 ppm amounts to 1 s error every ≈11.6 days, a deviation of 100 ppm to 1 s every ≈2.78 h.
- The oscillator frequency is time variable. There are short-term variations – caused by temperature changes, variations of electric supply voltage, air pressure, and so on – as well as long-term variations due to oscillator aging, see VIG [843] and `http://www.ieee-uffc.org/fcmain.asp?view=review` for detailed explanations. It is often safe to assume that the oscillator frequency is reasonably stable over times in the range of minutes to tens of minutes. On the other hand, this also implies that time synchronization algorithms should resynchronize once every few minutes to keep track of changing frequencies.

This implies that even if two nodes have the same type of oscillator and are started at the same time with identical logical clocks, the difference $|L_i(t) - L_j(t)|$ can become arbitrarily large as t increases. Therefore, a time synchronization protocol is needed.

How often must such a time synchronization protocol run? Suppose that a node adjusts only its **phase shift** ϕ_i as a result of a synchronization round. If the node's oscillator drift rate is constant and known to be x ppm and if the desired accuracy is δ s, then after at most $\frac{\delta}{x \cdot 10^{-6}}$ s the accuracy constraint is violated. For $x = 20$ ppm and an accuracy of 1 ms, a **resynchronization** is needed every 50 s. More advanced schemes try to estimate and correct not only the phase shift ϕ_i but also the current drift θ_i, hopefully prolonging the periods before resynchronization is required. Again, a one-time synchronization is not useful, since the drift rate is time varying. It is, however, often possible to bound the maximum drift rate, that is there is a $\rho_i > 0$ such that

$$\frac{1}{1+\rho_i} \leq \frac{\mathrm{d}}{\mathrm{d}t} H_i(t) \leq 1 + \rho_i \tag{8.1}$$

and this can be used to find a conservative resynchronization frequency.

8.1.3 Properties and structure of time synchronization algorithms

Time synchronization protocols can be classified according to certain criteria:

Physical time versus logical time In wireless sensor networks, applications and protocols mostly require physical time.

External versus internal synchronization Algorithms may or may not require time synchronization with external timescales like UTC.

Global versus local algorithms A global algorithm attempts to keep *all* nodes of a sensor network (partition) synchronized. The scope of local algorithms is often restricted to some geographical neighborhood of an interesting event. In global algorithms, nodes are therefore required to keep synchronized with not only single-hop neighbors but also with distant nodes (multihop). Clearly, an algorithm giving global synchronization also gives local synchronization.

Absolute versus relative time Many applications like the simple example presented in Section 8.1.1 need only accurate *time differences* and it would be sufficient to estimate the drift instead of phase offset. However, absolute synchronization is the more general case as it includes relative synchronization as a special case.

Hardware- versus software-based algorithms Some algorithms require dedicated hardware like GPS receivers or dedicated communication equipment while software-based algorithms use plain message passing, using the same channels as for normal data packets.

A priori versus a posteriori synchronization In a priori algorithms, the time synchronization protocol runs all the time, even when there is no external event to observe. In a posteriori synchronization (also called **post-facto synchronization** [235]), the synchronization process is triggered by an external event.

Deterministic versus stochastic precision bounds Some algorithms can (under certain conditions) guarantee absolute upper bounds on the synchronization error between nodes or with respect to external time. Other algorithms can only give stochastic bounds in the sense that the synchronization error is with some probability smaller than a prescribed bound.

Local clock update discipline How shall a node update its local clock parameters ϕ_i and θ_i? An often-found requirement is that backward jumps in time should be avoided, that is for $t < t'$ it shall not happen that $L_i(t) > L_i(t')$ after an adjustment.[4] An additional requirement might be to avoid sudden jumps, that is the difference $L_i(t') - L_i(t)$ for times t immediately before and t' immediately after readjustment should be small.

The most important **performance metrics** of time synchronization algorithms are the following:

Precision For deterministic algorithms, the maximum synchronization error between a node and real time or between two nodes is interesting; for stochastic algorithms, the mean error, the error variance, and certain quantiles are relevant.

Energy costs The energy costs of a time synchronization protocol depend on several factors: the number of packets exchanged in one round of the algorithm, the amount of computation needed to process the packets, and the required resynchronization frequency.

Memory requirements To estimate drift rates, a history of previous time synchronization packets is needed. In general, a longer history allows for more accurate estimates at the cost of increased memory consumption; compare Section 8.2.2.

Fault tolerance How well can the algorithm cope with failing nodes, with error-prone and time-variable communication links, or even with network partitions? Can the algorithm handle mobility?

It is useful to decompose time synchronization protocols for wireless sensor networks into four conceptual building blocks, the first three of which are already identified in reference [24]:

- The **resynchronization event detection block** identifies the points in time where resynchronization is triggered. In most protocols, resynchronizations are triggered periodically with a period depending on the maximum drift rate. A single resynchronization process is called a **round**. If rounds can overlap in time, sequence numbers are needed to distinguish them and to let a node ignore all but the newest resynchronization rounds.

[4] Users of make tools tend to have strong opinions on this...

- The **remote clock estimation block** acquires clock values from remote nodes/remote clocks. There are two common variants. First, in the **time transmission technique**, a node i sends its local clock $L_i(t)$ at time t to a neighboring node j, which receives it at local time $L_j(t')$ (with $t' > t$). Basically, node j assumes $t \approx t'$ and uses $L_i(t)$ as estimation for the time $L_i(t')$. This estimation can be made more precise by removing known factors from the difference $t' - t$, for example the time that i packet occupies the channel and the propagation delay. Second, in the **remote clock reading** technique, a node j sends a request message to another node i, which answers with a response packet. Node j estimates i's clock from the round-trip time of the message and the known packet transmission times. Finally, node j may inform node i about the outcome. This technique is discussed in more detail below.
- The **clock correction block** computes adjustments of the local clock based on the results of the remote clock estimation block.
- The **synchronization mesh setup block** determines which nodes synchronize with each other in a multihop network. In fully connected networks, this block is trivial.

8.1.4 Time synchronization in wireless sensor networks

In wireless sensor networks, there are some specifics that influence the requirements and design of time synchronization algorithms:

- An algorithm must scale to large multihop networks of unreliable and severely energy-constrained nodes. The scalability requirement refers to both the number of nodes as well as to the average node degree/node density.
- The precision requirements can be quite diverse, ranging from microseconds to seconds.
- The use of extra hardware only for time synchronization purposes is mostly ruled out because of the extra cost and energy penalties incurred by dedicated circuitry.
- The degree of mobility is low. An important consequence is that a node can reach its neighbors at any time, whereas in networks with high degree of mobility, intermittent connectivity and sporadic communication dominates (there are some publications explicitly targeting MANETs [84, 695]).
- There are mostly no fixed upper bounds for packet delivery delay, owing to the MAC protocol, packet errors, and retransmissions.
- The propagation delay between neighboring nodes is negligible. A distance of 30 m needs 10^{-7} s for speed of light $c \approx 300.000.000$ m/s.
- Manual configuration of single nodes is not an option. Some protocols require this, for example, the network time protocol (NTP) [553, 554, 556], where each node must be configured with a list of time servers.
- It will turn out that the accuracy of time synchronization algorithms critically depends on the delay between the reception of the last bit of a packet and the time when it is timestamped. Optimally, timestamping is done in lowest layers, for example, the MAC layer. This feature is much easier to implement in sensor nodes with open firmware and software than it would be using commodity hardware like commercial IEEE 802.11 network cards.

Many of the traditional time synchronization protocols try to keep the nodes synchronized all the time, which is reasonable when there are no energy constraints and the topology is sufficiently stable. Accordingly, energy must be spent all the time for running time synchronization protocols. For several sensor network applications this is unnecessary, for example, when the main task of a network is to monitor the environment for rare events like forest fires. With **post-facto synchronization** [235] (or a posteriori synchronization), a time synchronization on demand can be achieved. Here, nodes are unsynchronized most of the time. When an interesting external event

is observed at time t_0, a node i stores its local timestamp $L_i(t_0)$ and triggers the synchronization protocol, which for example, provides global synchronization with UTC time. After the protocol has finished at some later time t_1, node i has learned about its relative offset Δ to UTC time, that is, $t_1 = L_i(t_1) + \Delta$. Node i can use this information to relate the past event at t_0 also to UTC time. After node i has delivered the information about the event, it can go into sleep mode again, dropping synchronization. In a nutshell, post-facto synchronization is synchronization on demand and for a short time, to report about an important event.

Before discussing some of the proposals for time synchronization protocols suitable for sensor networks, let us briefly discuss some of the "obvious" solutions and why they do not fit.

- *Equip each node with a GPS receiver*: GPS receivers still cost some few dollars, need a separate antenna, need energy continuously to keep in synch (acquiring initial synchronization takes minutes!), and have form factors not compatible with the idea of very small sensor nodes [401]. Furthermore, to be useful, a GPS receiver needs a line of sight to at least four of the GPS satellites, which is not always achievable in hilly terrains, forests, or in indoor applications. One application of GPS in a sensor network for wildlife tracking is reported by JUANG et al. [388].
- *Equip each node with some receiver for UTC signals*:[5] the same considerations apply as for a GPS receiver.
- *Let some nodes at the edge of the sensor network send strong timing signals*: Such a solution can be used indoors and in flat terrain but requires a separate frequency and thus a separate transceiver on each node to let the time server not distort ongoing transmissions.

In the following sections, we present several proposals for time synchronization protocols in wireless sensor networks.

8.2 Protocols based on sender/receiver synchronization

In this kind of protocols, one node, called the *receiver*, exchanges data packets with another node, called the *sender*, to let the receiver synchronize to the sender's clock. One of the classic protocols for this is the network time protocol (NTP), widely used in the Internet [553, 554, 556]. In general, sender/receiver based protocols require bidirectional links between neighboring nodes.

8.2.1 Lightweight time synchronization protocol (LTS)

The lightweight time synchronization (LTS) protocol presented by VAN GREUNEN and RABAEY [839] attempts to synchronize the clocks of a sensor network to the clocks held by certain reference nodes, which, for example, may have GPS receivers. The protocol has control knobs that allow to trade off energy expenditure and achievable accuracy, and it gives stochastic precision bounds under certain assumptions about the underlying hardware and operating system. LTS makes no restrictions with respect to the local clock update discipline and it does not try to estimate actual drift rates.

LTS subdivides time synchronization into two building blocks:

- A pair-wise synchronization protocol synchronizes two neighboring nodes.
- To keep all nodes or the set of interesting nodes synchronized to a common reference, a **spanning tree** from the reference node to all nodes is constructed. If the single-hop synchronization errors are independent and identically distributed and have mean zero, the leaf nodes of the tree also

[5] For example, in Germany, a DCF77 receiver can be used for this.

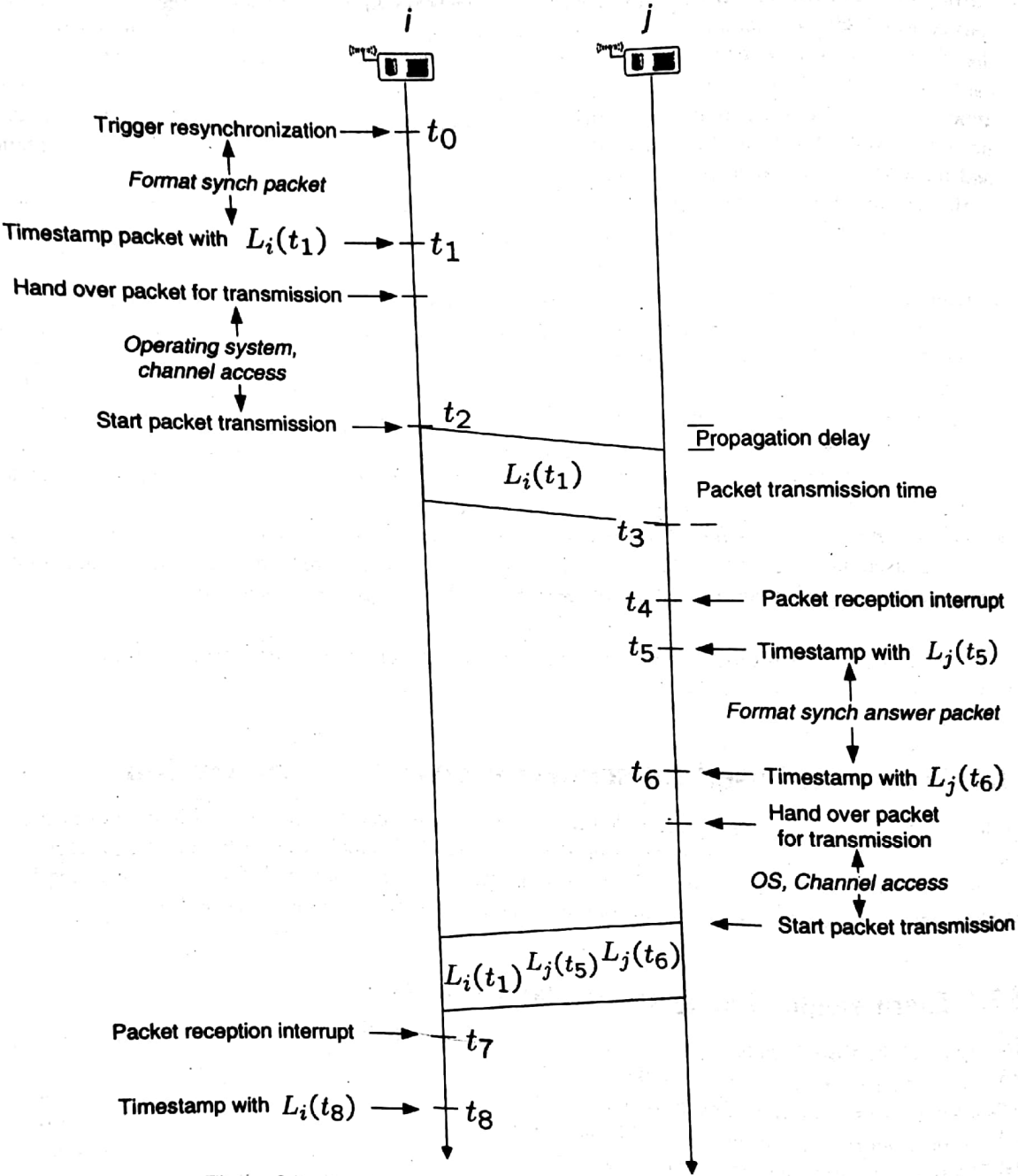


Figure 8.2 Partial sketch of operation in sender/receiver synchronization

have an expected synchronization error of zero but the variance is the sum of the variances along the path from the reference node to the leaf node. Therefore, this variance can be minimized by finding a minimum-height spanning tree.

Pair-wise synchronization

We first explain the pair-wise synchronization protocol (Figure 8.2). The protocol uses a remote clock reading technique. Suppose a node i wants to synchronize its clock to that of a node j.

After the resynchronization is triggered at node i, a synchronization request packet is formatted and timestamped at time t_1 with time $L_i(t_1)$. Node i hands the packet over to the operating system and the protocol stack, where it stays for some time. The medium access delay (Section 5.1.1) can be highly variable and make up for a significant fraction of this time. Often, this delay is a random variable. When node i is sending the first bit at time t_2, node j receives the last bit of the packet at time $t_3 = t_2 + \tau + t_p$, where τ is the propagation delay and t_p is the packet transmission time (packet length divided by bitrate). Some time later (interrupt latency), at time t_4, the packet arrival is signaled to node j's operating system or application through an interrupt and it is timestamped at time t_5 with $L_j(t_5)$. At t_6, node j has formatted its answer packet, timestamps it with $L_j(t_6)$, and hands it over to its operating system and networking stack. This packet includes also the previous timestamps $L_j(t_5)$ and $L_i(t_1)$. Node i receives the packet reception interrupt at time t_7 (which is t_6 plus operating system/networking overhead, medium access delay, propagation delay, packet transmission time, and interrupt latency) and timestamps it at time t_8 with $L_i(t_8)$.[6]

Let us now analyze how node i infers its clock correction. More precisely, node i wants to estimate $O = \Delta(t_1) := L_i(t_1) - L_j(t_1)$. To do this, we make the assumption that there is no drift between the clocks in the time between t_1 and t_8, that is $O = \Delta(t^*)$ for all $t^* \in [t_1, t_8]$, and in fact node i estimates O by estimating $\Delta(t_5)$. From the figure, the timestamp $L_j(t_5)$, which node i's gets back, is generated at some unknown time between t_1 and t_8. However, we can reduce this uncertainty by the following observations:

- There is one propagation delay τ plus one packet transmission time t_p between t_1 and t_5.
- There is another time $\tau + t_p$ between t_5 and t_8 for the response packet. For stationary nodes, we can safely assume that propagation delays are the same in both directions.
- The time between t_5 and t_6 is also known to node i from the difference $L_j(t_6) - L_j(t_5)$.

Therefore, the uncertainty about t_5 can be reduced to the interval $I = [L_i(t_1) + \tau + t_p, L_i(t_8) - \tau - t_p - (L_j(t_6) - L_j(t_5))]$. If we assume that the times spent in the operating system and networking stack, the interrupt latencies as well as the medium access delay are the same in both directions, then node i would conclude that j has generated its timestamp $L_j(t_5)$ at time (from i's point of view)

$$L_i(t_5) = \frac{L_i(t_1) + \tau + t_p + L_i(t_8) - \tau - t_p - (L_j(t_6) - L_j(t_5))}{2}.$$

Therefore,

$$O = \Delta(t_5) = L_i(t_5) - L_j(t_5) = \frac{L_i(t_8) + L_i(t_1) - L_j(t_6) - L_j(t_5)}{2}.$$

Now node i can adjust its local clock by adding the offset O to it. This way, node i synchronizes to j's local time, at the cost of two packets. When the goal is additionally to let node j learn about O, a third packet, sent from i to j and including O, is needed. In this case, the whole synchronization needs three packets.

The maximum synchronization error of this scheme is $|I|/2$ if the times τ and t_p are known with high precision. The actual synchronization error can essentially be attributed to different interrupt latencies at i and j, to different times between getting a receive interrupt and timestamping the packet, and to different channel access times. These uncertainties can be reduced significantly if the requesting node can timestamp its packet as lately as possible, best immediately before transmitting or right after obtaining medium access [272].

[6] Another account of the different variable delays in this process can be found in reference [430].

Several authors including ELSON et al. [236] propose to let the receiver timestamp packets as early as possible, for example, in the interrupt routine called upon *packet arrival* from the transceiver. For the case of Berkeley motes, ELSON et al. [236] show that for several receivers tasked with timestamping the same packet in their interrupt routines, the pair-wise differences in the actual timestamp generation times are normally distributed with zero mean and a standard deviation of $\sigma =$ 11.1 µs (compare Figure 8.6). VAN GREUNEN and RABAEY [839] make the additional assumption that the differences in channel access times obey the same distribution. Depending on the degree of correlation between access time difference and timestamping difference, the variance of the sum of these variables is at most four times the variance σ^2 of either component. Accordingly, VAN GREUNEN and RABAEY [839] characterize this sum (i.e. the overall error in the above estimation of O) as a normal random variable with variance $4\sigma^2$ and standard deviation 2σ. It is well known for the normal distribution that 99% of all outcomes have a difference of at most 2.3 times the standard deviation from the mean, and therefore under these assumptions, the maximum error after adjusting i's clock to j's is with 99% probability smaller than $2.3 \cdot 2 \cdot \sigma$ µs.

Networkwide synchronization

Given the ability to carry out pair-wise synchronizations, LTS next solves the task to synchronize all nodes of a (connected) sensor network with a reference node. If a specific node i has a distance of h_i hops to the reference node, and if the synchronization error is normally distributed with parameters $\mu = 0$ and $\sigma' = 2\sigma$ at each hop, and if furthermore the hops are independent, the synchronization error of i is also normally distributed with variance $\sigma_i^2 = 4h_i\sigma^2$.[7] On the basis of this observation, LTS aims to construct a **spanning tree** of minimum height and only node pairs along the edges of the tree are synchronized. If the synchronization process along the spanning tree takes a lot of time, the drift between the clocks will introduce additional errors.

Two different variants are proposed, a centralized and a distributed one.

Centralized multihop LTS

The reference node – for example, a node with a GPS receiver or another high-quality time reference – constructs a spanning tree $\mathcal{T}$ and starts synchronization: First the reference node synchronizes with its children in $\mathcal{T}$, then the children with their children, and so forth. Hence, each node must know its children. There are several algorithms available for distributed construction of a spanning tree [31, 526]; for LTS, two specific ones are discussed in reference [839], namely the distributed depth-first search (DDFS) and the Echo algorithms.

The reference node also has to take care of frequent resynchronization to compensate for drift. It is assumed that the reference node knows four parameters: the maximum height h of the spanning tree, the maximum drift ρ such that Equation 8.1 is satisfied for all nodes in the network, the single-hop standard deviation $2 \cdot \sigma$ (discussed above), and the desired accuracy δ. The goal is to always have a synchronization error of leaf nodes smaller than δ with 99% probability.[8] Immediately after resynchronization, a leaf node's accuracy is smaller than $h \cdot 2.3 \cdot 2 \cdot \sigma$ and it is allowed to grow at most to level δ. With maximum drift rate ρ, this growth takes $\frac{\delta - 2 \cdot 2.3 \cdot h \cdot \sigma}{\rho}$ time. The actual choice of the synchronization period should be somewhat smaller to account for the drift occurring during a single resynchronization, possibly harming the initial accuracy $2.3 \cdot 2 \cdot h \cdot \sigma$.

A critical issue is the communication costs. A single pair-wise synchronization costs three packets, and synchronizing a network of n nodes therefore costs on the order of $3n$ packets, not taking channel errors or collisions into consideration. Additionally, significant energy is needed to construct

[7] Please note that *any single* node i is with 99% probability synchronized within an error bound of $2.3 \cdot 2 \cdot \sqrt{h_i} \cdot \sigma$. However, this does *not* imply that all nodes simultaneously are synchronized with 99% within the same bounds. This can be easily seen from the Bonferroni inequations.

[8] We will leave out the 99% qualification henceforth.

the spanning tree, and it is proposed to repeat this construction upon each synchronization round to achieve some fault tolerance.

For reasons of fault tolerance, it is also beneficial to have multiple reference nodes: If one of them fails or if the network becomes partitioned, another one can take over. A leader election protocol is useful to support dynamic reference nodes.

Distributed multihop LTS

The second variant is the **distributed multihop LTS** protocol. No spanning tree is constructed, but each node knows the identities of a number of reference nodes along with suitable routes to them. It is the responsibility of the nodes to initiate resynchronization periodically.

Consider the situation shown in Figure 8.3 and assume that node 1 wants to synchronize with the reference node R. Node 1 issues a synchronization request toward R, which results in a sequence of pair-wise synchronizations: node 4 synchronizes with node R, node 3 synchronizes with node 4, and so forth until node 1 is reached. Two things are noteworthy:

- As a by-product, nodes 2, 3, and 4 also achieve synchronization with node R.
- Given the same accuracy requirement δ and the same drift rate ρ for all nodes, the resynchronization frequency for a node i that is h_i hops away from the reference node is given by $\frac{\delta - 4 \cdot 2.3 \cdot h_i \cdot \sigma}{\rho}$. Therefore, in the figure, nodes 1 and 6 have the shortest resynchronization period. If these two nodes always request resynchronization with node R, the intermediate nodes 2, 3, 4, and 5 never have to request resynchronization by themselves.

A node should choose the closest reference node to minimize its synchronization error. This way, a minimum weight tree is not constructed explicitly, but it is the responsibility of the routing protocol to find good paths. For certain network setups and routing protocols, the issue of routing/synchronization cycles may arise, which have to be avoided. From the previous example, it is also beneficial if a node can take advantage of ongoing synchronizations. Consider for example node 5 in Figure 8.3. Instead of synchronizing node 5 independently with the reference node R (which would cause nodes 3 and 4 to handle two synchronization requests simultaneously), node 5 can ask all nodes in its neighborhood about ongoing synchronization requests. If there is any, node 5 can wait for some time and then attempt to synchronize with the responder.

Another optimization of LTS is also explained in Figure 8.3, using dashed lines. Suppose again that node 5 wants to synchronize. As explained above, one option would be to let node 5 join an ongoing synchronization request at node 3. On the other hand, it might be necessary to keep nodes 7 and 8 also synchronized with R. To achieve this, node 5 can issue its request through nodes 7 and 8 to R and synchronize the intermediate hops as a by-product. This is called **path diversification.**

The properties of the LTS variants were investigated by simulating 500 randomly distributed nodes in a 120 m (120 m) rectangle, each node having a transmit range of 10 m. The single reference

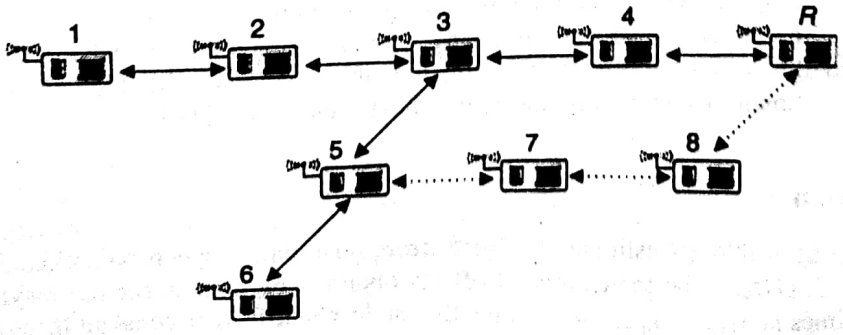


Figure 8.3 Distributed multihop LTS

$$L(\rho_x) = \log\left[\prod_{k=1}^{n-1} \frac{1}{\sqrt{2\pi\sigma_*^2}} \exp\left(\frac{}{2\sigma_*^2}\right.\right.$$

$$= -\frac{n}{2}\log(2\pi\sigma_*^2) + \frac{1}{2\sigma_*^2}\sum_{k=1}^{n-1}(O(t_k) - \rho_x(t_k - t_{k-1}))^2.$$

Setting the derivative $\frac{\mathrm{d}}{\mathrm{d}\rho_x}L(\rho_x)$ to zero and solving for ρ_x yields

$$\rho_x = \frac{\sum_{k=1}^{n-1} O(t_k)\cdot(t_k - t_{k-1})}{\sum_{k=1}^{n-1}(t_k - t_{k-1})^2}.$$

8.2.3 Timing-sync protocol for sensor networks (TPSN)

The Timing-Sync Protocol for Sensor Networks (TPSN) [271, 272] is another interesting sender/receiver based protocol. Again, we first explain the approach to pair-wise synchronization before turning to the multihop case.

Pair-wise synchronization

The pair-wise synchronization protocol of TPSN has some similarities with LTS. It operates in an asymmetric way: Node i synchronizes to the clock of another node j but not vice versa (Figure 8.4). The operation is as follows:

- Node i initiates resynchronization at time t_0. It formats a **synchronization pulse packet** and hands it over to the operating system and networking stack at time t_1.
- The networking stack executes the MAC protocol and determines a suitable time for transmission of the packet, say t_2. Immediately before transmission, the packet is timestamped with $L_i(t_2)$. By timestamping the packet immediately before transmission and not already when the packet has been formatted in the application layer, two sources of uncertainty are removed: the operating system/networking stack and the medium access delay. The remaining uncertainty is the small time between timestamping the packet and the true start of its transmission. This delay is created, for example, by the need to recompute the packet checksum immediately before sending it.
- After propagation delay and packet transmission time, the last bit arrives at the receiver at time t_3, and some time after this the packet receive interrupt is triggered, say at time t_4. The receiver timestamps the packet already in the interrupt routine with $L_j(t_4)$.
- Node j formats an **acknowledgement packet** and hands it over at time t_5 to the operating system and networking stack. Again, the networking stack executes the MAC protocol and sends the packet at time t_6. Immediately before transmission, the packet is timestamped with $L_j(t_6)$, and the packet carries also the other timestamps $L_i(t_2)$ and $L_j(t_4)$.

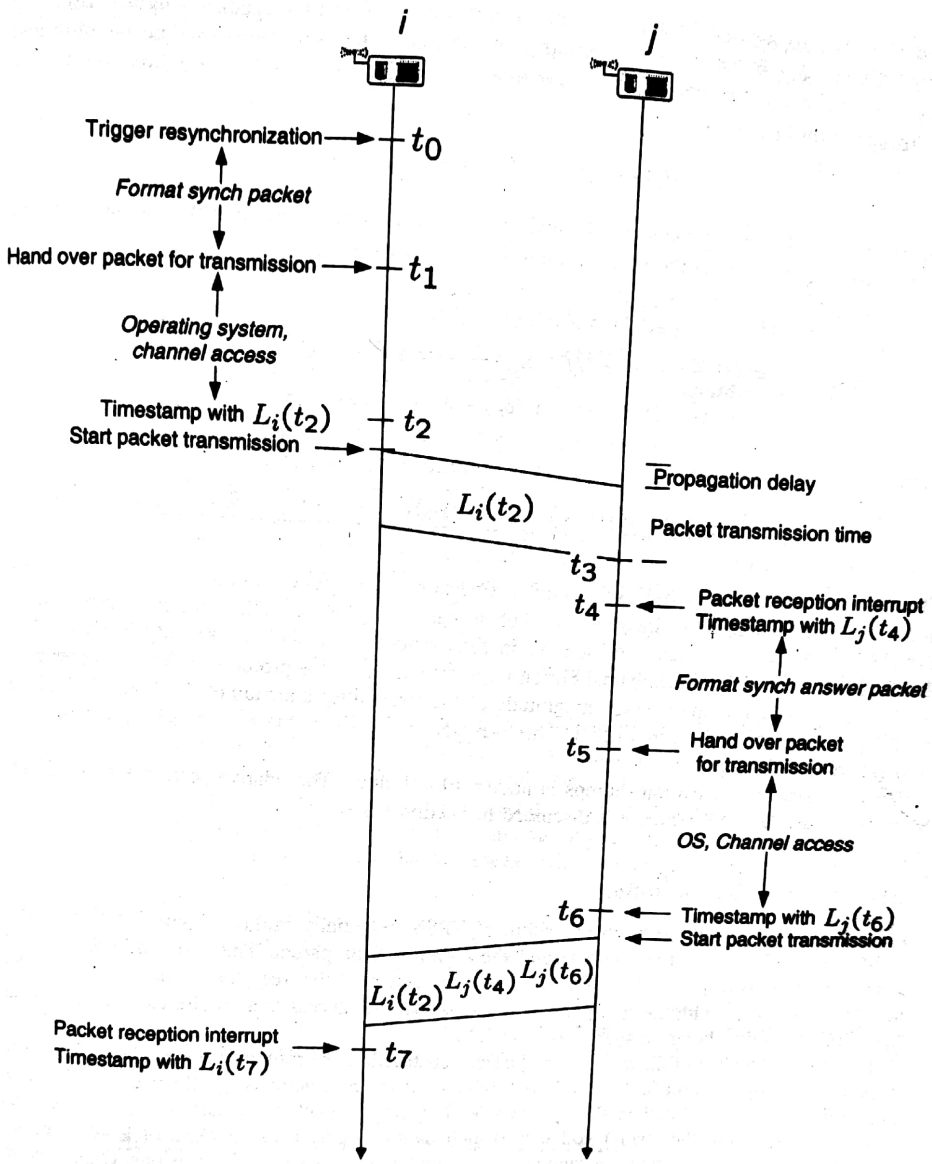


Figure 8.4 Sender/receiver synchronization in TPSN

- Finally, node i timestamps the incoming acknowledgement packet as early as possible with $L_i(t_7)$.

If O is the true offset of node i's clock to node j's clock, we have for the synchronization pulse packet

$$L_j(t_4) = L_i(t_2) + O + \tau + t_p + \delta_{s,i} + \delta_{r,j},$$

where τ is the propagation delay, t_p is the packet transmission time (packet length divided by channel bitrate), $\delta_{s,i}$ is the (small) uncertainty at the transmitter side between timestamping and actual start of transmission, and $\delta_{r,j}$ is the **receiver uncertainty** at j. In the other direction, for the acknowledgement packet, we have

$$L_i(t_7) = L_j(t_6) - O + \tau + t_p + \delta_{s,j} + \delta_{r,i}$$

(assuming that acknowledgement packet and synchronization pulse packet have the same length and the propagation delay is the same in both directions). Now

$$\begin{aligned}&(L_j(t_4) - L_i(t_2)) - (L_i(t_7) - L_j(t_6))\\ &= (O + \tau + t_p + \delta_{s,i} + \delta_{r,j}) - (-O + \tau + t_p + \delta_{s,j} + \delta_{r,i})\\ &= 2 \cdot O + (\delta_{s,i} - \delta_{s,j}) + (\delta_{r,j} - \delta_{r,i}).\end{aligned}$$

Therefore,

$$O = \frac{(L_j(t_4) - L_i(t_2)) - (L_i(t_7) - L_j(t_6))}{2} - \frac{\delta_{s,i} - \delta_{s,j}}{2} - \frac{\delta_{r,j} - \delta_{r,i}}{2}.$$

The key feature of this approach is that node i timestamps the outgoing packet as lately as possible and node j timestamps the incoming packet as early as possible. This requires support from the MAC layer, which is easier to achieve in sensor nodes than with commodity hardware like IEEE 802.11 network interface cards. For an implementation of this protocol on MICA motes with a 115 kbps transceiver, the average magnitude of the transmitter uncertainty $\delta_{s,i} - \delta_{s,j}$ is 1.15 µs (the expectation is clearly zero) [272]. The average magnitude of the total synchronization error was found to be ≈17 µs.

This protocol allows arbitrary jumps in node i's local clock. The relative performance of TPSN compared with the RBS protocol is discussed in Section 8.3.1.

Networkwide synchronization

The networkwide synchronization algorithm of TPSN essentially builds a **spanning tree** where each node knows its level in the tree and the identity of its parent. The **root node** is assigned level 0 and it is its responsibility to trigger the construction of the tree. All reachable nodes in the network synchronize with the root node. If the root node has access to a precise external timescale like UTC, all nodes therefore synchronize to UTC.

The protocol works as follows. To start the tree construction, the root node sends a *level_discovery* packet containing its level 0. All one-hop neighbors of the root node assign themselves a level of one plus the level indicated in the received *level_discovery* packet and accept the root as their parent. Subsequently, the level 1 nodes send their own *level_discovery* packets of level 1 and so forth. The level 1 nodes choose a random delay to avoid excessive MAC collisions. Once a node has received a *level_discovery* packet, the packet originator is accepted as parent and all subsequent packets are dropped. After a node has found a parent, it periodically resynchronizes to the parent's clock.

A node might fail to receive *level_discovery* packets because of MAC collisions or because it is deployed after initial tree construction. If a node i does not receive any *level_discovery* packet within a certain amount of time, it asks its one-hop neighborhood about an already existing tree by issuing a *level_request* packet. The neighboring nodes answer by sending their own level. Node i collects the answers from some time window and chooses the neighbor with the smallest level as its parent.

The tree maintenance is integrated with resynchronization. To account for drift, a node i must run the pair-wise algorithm with its parent j periodically. If this fails subsequently for a number of times, node i concludes that its parent has moved or passed away. If the level of i is two or larger, it sends a *level_request* packet, collects the answers for some time and assigns itself a new level from the lowest-level answer packet. If i is at level one, it concludes that the root node has died. There are several possibilities to resolve this situation. One of them is to run a leader election protocol among level 1 nodes.

This approach has the following properties:

- The resulting spanning tree is not necessarily a minimal one, since MAC collisions and random delays may lead to a situation where a node receives a *level_discovery* from a higher-level node first. However, there is a trade-off between the synchronization accuracy (longer paths imply larger average error) and the overhead for tree construction. Algorithms for finding minimal spanning trees are more elaborate.
- If two nodes i and j are geographically close together and receive the same level ν *level_discovery* in the tree setup phase, both assign themselves level $\nu + 1$ and try to resend the *level_discovery* packet. One of them wins contention. Since both are close together, their one-hop neighborhoods are almost identical. As a result, all so-far-unsynchronized neighbors accept node i as their parent and create significant resynchronization load for i, whereas node j spends almost no energy because it has no children. To avoid unfairness, the tree construction should be repeated periodically, which in turn creates network load.
- The average magnitude of the synchronization error between a level ν node and the root node increases with ν, but gracefully. For one hop, the average synchronization error is $\approx 17\,\mu s$ and for five hops $\approx 23\,\mu s$.
- It is possible to achieve post-facto synchronization. In this case, no spanning tree is constructed. Consider a scenario in which a node i_0 wants to communicate an event (which happened at time t) to another node i_n over a number of intermediate hops $i_1, i_2, \ldots, i_{n-1}$. Node i_0 sends the packet with its local timestamp $L_{i_0}(t)$ to i_1. Subsequently, node i_1 synchronizes its clock to that of node i_0 and forwards the packet to node i_2, and so forth. Finally, all nodes including node i_n have synchronized to node i_0 and i_n has the packet with timestamp $L_{i_0}(t)$ and can thus decide about the age of the event.

8.3 Protocols based on receiver/receiver synchronization

In sender/receiver based synchronization, the receiver of a timestamped packet synchronizes with the sender of the packet. In receiver/receiver synchronization approaches, multiple receivers of the same timestamped packet synchronize with each other, but not with the sender. We discuss two protocols belonging to this class.

8.3.1 Reference broadcast synchronization (RBS)

The **reference broadcast synchronization (RBS)** protocol [236] consists of two components. In the first one, a set of nodes within a single broadcast domain (i.e. a set of nodes that can hear each other) estimate their peers' clocks. The second component allows to relate timestamps between distant nodes with several broadcast domains between them. We explain each component in turn.

Synchronization in a broadcast domain

The basic idea is as follows. A sender sends periodically a (not necessarily timestamped) packet into a broadcast channel and all receivers timestamp this packet. The receivers exchange their

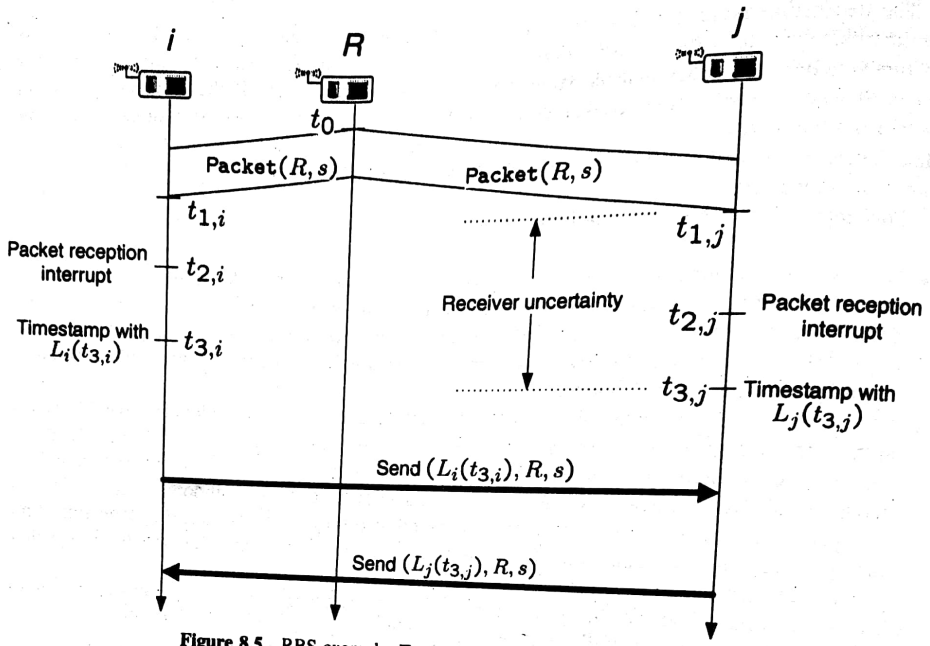


Figure 8.5 RBS example: Two nodes i and j and a sender R

timestamps and can use this data to learn about their neighbors' clocks. By repeating this process, the nodes can not only learn about their mutual phase offsets but also about their drift rates. The nodes do not adjust their local clocks but construct a table that, for each neighbor, stores the necessary parameters to convert clock values.

An example

An example is shown in Figure 8.5. Two nodes i and j want to synchronize. At time t_0, another node R broadcasts a **pulse packet**, which includes its identification R and a sequence number s.[9] Since nodes i and j have a different distance to node R, the propagation delays differ: The last bit reaches node i at real time $t_{1,i}$ and node j at time $t_{1,j}$, and the propagation delays are τ_i and τ_j, respectively. In both nodes, a packet reception interrupt is generated, say, at times $t_{2,i}$ and $t_{2,j}$ for nodes i and j. Some short time later, at time $t_{3,i}$, node i timestamps the packet with its local timestamp $L_i(t_{3,i})$; node j behaves similarly, producing a timestamp $L_j(t_{3,j})$. After this, nodes i and j exchange their timestamps and the identity (sender address, sequence number) of the corresponding pulse packet. Both nodes can easily compute the relative phase shifts of their clocks by assuming $t_{3,i} = t_{3,j}$. Specifically, node i stores the value $O(t_{3,i}) = L_i(t_{3,i}) - L_j(t_{3,j})$ as the phase offset in a local table without readjusting its clock. Clearly, this scheme can benefit from having receivers timestamp incoming packets as quickly as possible in the interrupt routine.

Adopting the terminology introduced for TPSN, the time between receiving the last bit and timestamping the packet is called receiver uncertainty. This is denoted by $\delta_{r,i}$ and $\delta_{r,j}$ for nodes i and j, respectively.

[9] The pulse packet does not need to be a dedicated time synchronization packet, normal data packets carrying sequence numbers suffice.

Achievable precision for a single pulse

Ultimately, the whole computation is perfectly precise when the assumption $t_{3,i} = t_{3,j}$ is really true. Let us briefly analyze the sources of possible synchronization errors.

- The *propagation delay*: In a sensor network, the broadcast domain is typically small and the propagation delay of the packet is negligible. Furthermore, it is only the *difference* in propagation delay between i and j that matters and this difference tends to be even smaller.
- The delay between receiving the last bit and generating the packet reception interrupt: This might be due to hardware processing delays (like checksum computations) and also short-term blocking of interrupts in case of critical sections or servicing interrupts of higher priority. Again, it is the *difference* between i and j that counts for synchronization errors.
- The delay between receiving the packet interrupt and timestamping the packet: If the timestamp is already generated in the interrupt routine, this delay is small.
- The drift between timestamping and exchanging the observed timestamps also contributes to synchronization errors. The more the time that elapses, the larger this error will be.

Compared to sender/receiver based approaches, the time required for R to format its packet, move it through its operating system and networking software, as well as the medium access delay are completely irrelevant since the common point of reference for i and j is the time instant t_0 where the packet appears on the medium.

ELSON et al. [236] have characterized the differences between the times where receivers timestamp a pulse packet among a set of five Berkeley motes. All motes are equipped with a 19.200 b/s transceiver and run TinyOS. For the measurements, the motes raise an I/O pin at the same time they timestamp the packet, and the I/O signals were picked up by an external logic analyzer. Another node sent 160 pulse packets at random times, and for each pulse packet and for each of the 10 possible receiver pairs, the difference of the signal transition times was captured. The results, shown in Figure 8.6, indicate that the error between receivers seem to have a normal distribution with sample mean zero and a standard deviation of $\sigma = 11.1$ µs (according to a Chi-square test with confidence level 99.8 %), which is significantly smaller than the time needed to transmit a single bit at a bitrate of 19.200 bps.

Comparison of RBS and TPSN

GANERIWAL et al. [272] compare RBS and TPSN, both analytically and by comparing implementations on MICA motes. Both protocols timestamp received packets already in the receiver interrupt.

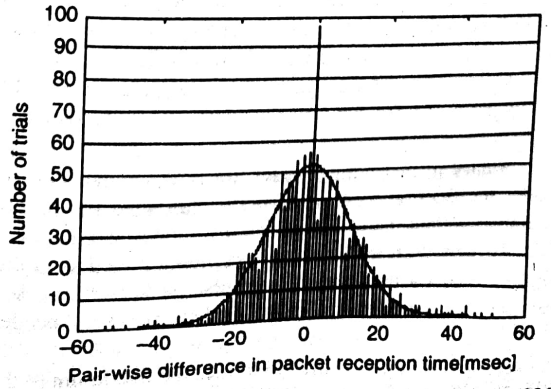


Figure 8.6 Distribution of differences in packet reception time. Reproduced from [236, Fig. 2] by permission of Jeremy Elson.

As compared to other sender/receiver based protocols, TPSN removes much of the uncertainty at the sender (operating system and networking stack, medium access delay) by timestamping an outgoing packet immediately before transmission. For the case of TPSN, we have derived the equation

$$O = \frac{(L_j(t_4) - L_i(t_2)) - (L_i(t_7) - L_j(t_6))}{2} - \frac{\delta_{s,i} - \delta_{s,j}}{2} - \frac{\delta_{r,j} - \delta_{r,i}}{2}$$

where $L_x(t_y)$ are the timestamps and $\delta_{s,i}$ and $\delta_{r,i}$ are transmitter and receiver uncertainty at node i, respectively.

Now, let us consider RBS. Say, we have two receivers i and j, and node i wants to estimate the offset to node j. Let the true offset be O. From Figure 8.5, we can write (with $t_1 = t_{1,i}$ being the time where i timestamps the pulse packet):

$$L_i(t_1) = L_j(t_1) + O - (\delta_{r,i} - \delta_{r,j}) - (\tau_i - \tau_j)$$

resulting in

$$O = L_i(t_1) - L_j(t_1) + (\delta_{r,i} - \delta_{r,j}) + (\tau_i - \tau_j).$$

Assuming that (i) the difference in propagation delay is negligible, and (ii) he transmitter uncertainty is small (Section 8.2.3), the error in estimating the offset is dominated by the term $\frac{\delta_{r,j} - \delta_{r,i}}{2}$ in the case of TPSN and by $\delta_{r,i} - \delta_{r,j}$ for RBS.

For neither TPSN nor RBS, the possible clock drift during exchange of the synchronization packets has been considered.

Using multiple pulses

The sender can send pulse packets regularly, and for each pulse packet, the nodes i and j exchange their local observations. The least-squares linear regression proposed for RBS is almost equivalent to the minimum-variance unbiased estimator discussed in Section 8.2.2. An outlier-rejection technique is used additionally. By disregarding those observations older than a certain threshold (a few minutes in RBS), time-varying drift rates can be accommodated.

Multiple nodes and RBS costs

It is straightforward to extend this technique to $m > 2$ nodes in a broadcast domain: The pulse packets are picked up by all m nodes and all nodes exchange their observations. This way, a single one of these nodes, say node i, can estimate drift and phase offsets for all its peers in the broadcast domain. By storing these values for each peer in a table, node i can convert its local timestamps to the time base of any of its peers. As for the pulse senders, there are at least the following choices:

- In networks where stationary infrastructure nodes are present (like, for example, IEEE 802.15.4 in the beaconed mode, see Section 5.5), these nodes can send the pulse packets. They can do even more – they can also collect the observations from the sensor nodes, compute for each node pair the offsets and drift rates with the least-squares method, and send the results back.
- Each node acts as both a receiver and a sender of pulse packets. In this scenario, both the sender identification and the sequence numbers in pulse packets are really needed.

The precision goal in a broadcast domain is then to reduce the maximum of the phase errors between all node pairs, called **group dispersion**. It is shown that the group dispersion depends both on the group size (larger groups have larger dispersion) and on the number of observations used to estimate

phase shift and drift rate (increasing the number of observations decreases the group dispersion). For two nodes, it is shown that increasing the number of observations beyond 30 provides no additional gains in precision. The RBS protocol is also compared with NTP, which uses a sender/receiver based scheme. In the investigated scenario, RBS has a much smaller group dispersion than what could be achieved with NTP. Furthermore, the performance of RBS is almost insensitive to the background load in the broadcast domain, whereas NTPs group dispersion increases with increasing load. This can be explained by the influence of medium access delays in NTP. In the given scenario, RBS is approximately eight times better than NTP without background load.

This technique of synchronizing introduces some overhead, which is not yet fully characterized:

- To exchange the observations of a group of m nodes in a broadcast domain, a number of $2m$ packets is needed when the observations are collected and evaluated at a central node and afterward sent back to the receivers. In scenarios without central nodes, exchanging observations takes $m \cdot (m-1) = O(m^2)$ packets. In dense sensor networks, a node can be a member in several (partially overlapping) broadcast domains, and the workload increases accordingly.
- The rate of pulse packets needed to keep the group dispersion below a certain level depends on the group size. A more precise characterization of this relationship needs to be developed.
- The least-squares regression requires significant computation.

RBS and post-facto synchronization

When looking at the overhead, one gets the feeling that RBS can become quite expensive in terms of the number of exchanged packets and computation. However, one of the most important features of RBS is that it can participate in **post-facto synchronization**. In this mode, the nodes do not synchronize with each other until an external event of interest happens. Each node i timestamps this event with a local timestamp $L_i(t_0)$. This event triggers synchronization among nodes and the nodes learn about their relative phase shifts and drift rates. If synchronization is acquired quickly enough after the event occurred, it is safe to assume that the relative drift has not changed. Therefore, node j can use its estimation of node i's phase shift and drift rate to accurately estimate $L_j(t_0)$, that is, to make a *backward extrapolation*.

Network synchronization over multiple hops

What we have seen so far is local synchronization in a broadcast domain. In most cases, a broadcast domain covers only a tiny fraction of a whole sensor network. So the question is: How can an a node learn at which time, measured with respect to its local timescale, a remote event occurred?

Timestamp conversion approach

The basic idea is to not produce a global timescale, but to convert a packet's timestamp at each hop into the next hop node's timescale until it reaches the final destination. Figure 8.7 serves as an example. Let us assume that node 1 has observed an event that it wants to report to the sink node. The human operator at the sink node wants to know when the event happened and he needs this information in an accepted timescale like UTC. Node 1 timestamps the event at time $L_1(t)$, and the packet has to pass through nodes $1 \rightarrow 3 \rightarrow 5 \rightarrow 9 \rightarrow 8 \rightarrow 14 \rightarrow 15 \rightarrow$ Sink. Furthermore, assume that nodes form broadcast domains as indicated by the circles. Observe that nodes 5, 8, and 9 are members of multiple broadcast domains. The source node 1 is in the same broadcast domain as the next node 3 and thus node 1 can use its estimates of the phase shift and drift rate of node 3 to convert the timestamp $L_1(t)$ into $L_3(t)$. Node 1 timestamps the event packet not with its own timestamp but with $L_3(t)$. Node 3, upon receiving this packet, makes a routing decision and infers that node 5 is the next hop. Fortunately, node 5 is in the same broadcast domain as node 3 and node 3 can thus convert the received timestamp $L_3(t)$ into $L_5(t)$. This process is continued until the packet reaches the sink. The last node before the sink, node 15, is in the same broadcast

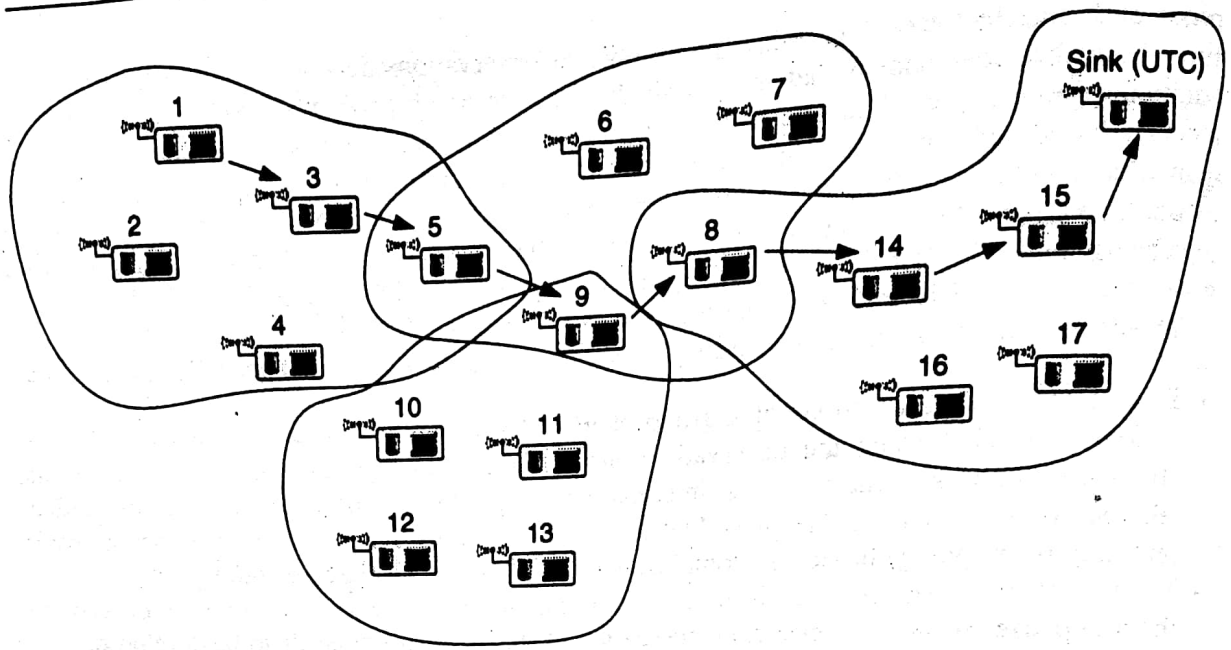


Figure 8.7 RBS example: Multiple broadcast domains

domain as the sink, which keeps UTC time. Again, node 15 knows its phase shift and drift rate with respect to the sink and can thus convert the timestamp $L_{15}(t)$ to UTC time.

The following points are noteworthy:

- The sink node gets an event timestamp in the UTC timescale.
- Each node keeps its local timescale and timestamp conversion can be integrated with packet forwarding. In fact, when node i has to decide about a possible forwarder, only candidates j that have a broadcast domain in common with i are suitable candidates. This requires that for node i and a possible forwarder j there exists a third node whose pulse packets can be heard by both i and j. These might not always exist in sparse sensor networks.
- There is no information related to time synchronization that must be globally available.
- If the synchronization error over a single hop is characterized by a normal distribution with zero mean and standard deviation σ and if the synchronization errors of different hops are independent, then after n hops the synchronization error is Gaussian with zero mean and standard deviation $\sigma \cdot \sqrt{n}$. This growth rate is also confirmed by measurements.

How to create the broadcast domains?

One of the open points in RBS is how the broadcast domains are constituted and how it can be ensured that a time conversion path actually exists between two desired nodes, say, a sensor and a sink node.

Let us look at two different scenarios (Figure 8.8). In the first scenario, a number of static nodes act as dedicated pulse senders and also as packet forwarders; these are shown as circles in the figure. Ordinary sensor nodes (displayed as rectangles) in the range of a pulse sender do not necessarily have to be immediate neighbors to be synchronized. The sensor nodes timestamp the pulse packets, transmit their observations to the pulse sender, which computes for each pair of sensor nodes the

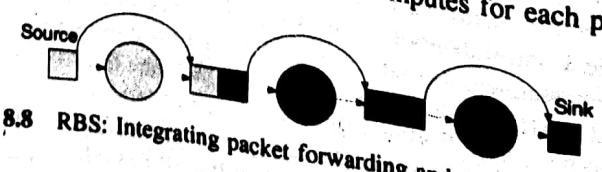


Figure 8.8 RBS: Integrating packet forwarding and timestamp conversion

relative phase shifts and drift rates, and distributes the results back. A broadcast domain is thus formed by the transmission range of the pulse sender. A sensor node i about to forward a packet has two options:

- It can by itself determine another sensor node j in the same broadcast domain to forward the packet to. If both are one-hop neighbors, node i can convert the timestamp and send the packet directly to j. This is indicated by solid arcs in Figure 8.8. In the other case, node i can ask the pulse sender to forward the packet to node j without manipulating it. In both cases, the pulse sender is not required to keep the table with conversion information.
- It can forward the packet to the pulse sender. It is the pulse sender's responsibility to look up a successor node j and to convert the timestamp (indicated by the dashed lines in Figure 8.8). Accordingly, the pulse sender is required to keep the table with conversion parameters.

In either case, the pulse sender *does not make use of its local clock* and thus need not be synchronized with its sensor nodes.[10] In both cases, the broadcast domains should be chosen as follows:

- Each sensor node is a member of at least one broadcast domain.
- When a packet is to be delivered from a source node to a sink node in a different broadcast domain, there must be a series of neighboring broadcast domains on the path between source and sink and two neighboring broadcast domains must overlap in at least one node. These **gateway nodes** (the rectangular nodes with two different grey levels in Figure 8.8) know the conversion parameters for all nodes in both broadcast domains and can convert timestamps accordingly. There needs to be enough overlap between domains to ensure the presence of gateway nodes. Ideally, between two domains there is more than one gateway node to allow for some balancing of forwarding load.
- The number of conversions necessary between a source and a sink node should be small to avoid loss of precision; this calls for large broadcast domains. On the other hand, in larger domains, more packets have to be exchanged and a larger transmit power has to be used to reach all neighbors, draining batteries more quickly.

This can be considered as a clustering problem (see Chapter 10), with the goal to find **overlapping clusters**. A dedicated clustering protocol for the purposes of RBS is presented by [567]. This protocol allows to adjust the cluster size to find a trade-off between minimizing the number of hops and minimizing the number of packets/transmit power.

If there are no dedicated nodes available or when really all nodes including pulse senders need to be synchronized, the broadcast domains must be smaller and all members of a broadcast domain must be one-hop neighbors to be able to exchange the pulse observations as well as to forward data packets. A broadcast domain is thus a **clique** of nodes. In such a scenario each node acts both as a receiver as well as a sender of pulse packets, and forwarding decisions are restricted to those one-hop neighbors that share at least one broadcast domain with the forwarding node.

8.3.2 Hierarchy referencing time synchronization (HRTS)

The TSync protocol presented by DAI and HAN [189] combines a receiver/receiver and a sender/receiver based technique. It consists of two subprotocols, which may act independent of each other. The Hierarchy Referencing Time Synchronization (HRTS) protocol discussed below is a receiver/receiver technique synchronizing all nodes in a broadcast domain and additionally constructing a synchronization tree to synchronize a whole network. The synchronization is triggered periodically

[10] To achieve such a synchronization by means of RBS, *another* pulse sender would be needed in range of the original pulse sender and its associated nodes ...

by the root of the tree. The other protocol, called ITR (Individual-based Time Request) protocol, is discussed briefly in Section 8.4.

Synchronization in a broadcast domain

We explain the approach by an example, shown in Figure 8.9. A dedicated node, called a **root node** or **base station** (node R in Figure 8.9) triggers time synchronization at time t_1 (corresponding to local time $L_R(t_1)$) by broadcasting a **sync_begin** announcement packet. This packet includes two parameters, a **level** value (explained below) and the identification of one of node R's one-hop neighbors, say node i. The selected node i timestamps the received packet at time t_2 with its local time $L_i(t_2)$. Another node j timestamps the same packet at time t_2' with $L_j(t_2')$. Similar to RBS, t_2 and t_2' can differ slightly because of random receiver uncertainties. Since node i has found its identification in the packet, it formats an answer packet and timestamps it for transmission at time t_3 with its local time $L_i(t_3)$. Both timestamps $L_i(t_2)$ and $L_i(t_3)$ are included in the answer packet. The root node R receives the packet and timestamps it at time t_4 with $L_R(t_4)$. [11] The root node R

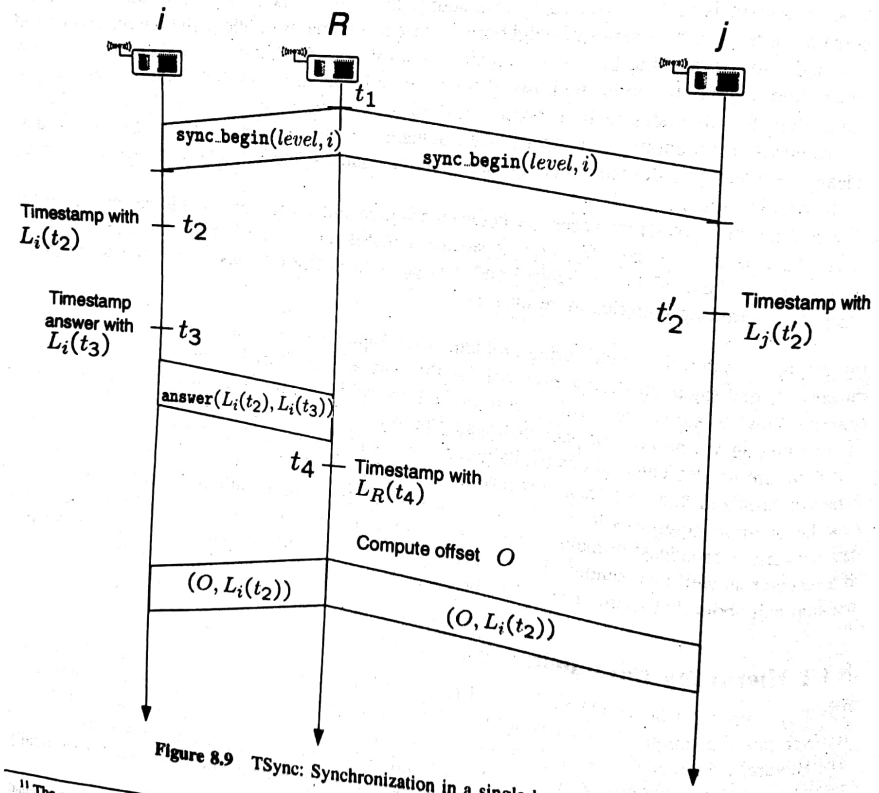


Figure 8.9 TSync: Synchronization in a single broadcast domain

[11] The explicit inclusion of a node identification requires node R to know its neighborhood. Alternatively, node R can omit the identification and all neighbors start a timer with a random value. Node R then just picks the first answer.

In the next step, the root node R broadcasts the values O and $L_i(t_2)$ to all nodes. Now, let us look how the individual nodes adjust their clocks:

- Node i simply subtracts the offset $O_{R,i}$ from its local clock.
- Assume that another node j in the same broadcast domain has phase offset $O_{j,i}$ to node i. Under the assumption that i and j receive the **sync_begin** at the same time, we have $t_2 = t_2'$ and $L_i(t_2) = L_j(t_2') + O_{j,i}$. Upon receiving the clock value $L_i(t_2)$ from the root node's final broadcast, node j can compute $O_{j,i}$ directly as $O_{j,i} = L_i(t_2) - L_j(t_2')$. On the other hand, since

$$L_i(t_2) = L_R(t_2) + O_{R,i}$$
$$L_j(t_2) = L_r(t_2) + O_{R,j}$$

we obtain

$$L_R(t_2) + O_{R,i} = L_j(t_2') + O_{j,i}$$

which gives

$$\begin{aligned} O_{R,j} &= L_j(t_2) - L_R(t_2) \\ &\approx L_j(t_2') - L_R(t_2) \\ &= L_R(t_2) + O_{R,i} - O_{j,i} - L_R(t_2) \\ &= O_{R,i} - O_{j,i} \\ &= O_{R,i} - (L_i(t_2) - L_j(t_2')). \end{aligned}$$

Therefore, node j can compute the phase offset to node R directly from its own observation $L_j(t_2')$ and the values from the final broadcast, *without exchanging any packets with other nodes.*

The important property of this scheme is that three packets suffice to synchronize *all* of R's neighbors to R's clock, no matter what their number is and whether they are in mutual range or not. In contrast to RBS, the receivers of the **sync_begin** packet do not have to exchange their observations in a pair-wise fashion and the protocol is therefore insensitive to the network density.

Dai and Han [189] have shown experimentally that with respect to a single broadcast domain as well as over multiple hops the HRTS protocol and RBS have approximately the same distribution of synchronization errors when taken over a large number of repetitions. Similar to the other protocols, the receive and transmit uncertainties can be reduced by timestamping outgoing packets as lately as possible and timestamping incoming packets as early as possible.

Although not necessary for this approach to work, Dai and Han [189] propose to run this protocol over a separate MAC channel. The goal is to separate synchronization-related traffic from user data traffic and thus to reduce the probability of collision and the medium access delays when the selected node i answers to the root node's **sync_begin** packet. Specifically, the root node sends the **sync_begin** on a dedicated **control channel** and includes the specification of another dedicated

channel, the **clock channel**. Both the root node and the selected node i switch to the clock channel and i transmits its answer packet. Finally, the root node broadcasts the packet containing $L_i(t_2)$ and O, again on the control channel. All other nodes can continue forwarding data packets as R and i exchange their packets.

Network synchronization over multiple hops

Let us assume that the whole network is static and connected and that there is at least one **reference node** having access to an external time source and, thus, to UTC time; there can be multiple reference nodes. These reference nodes are assigned level 0 and become root nodes for their one-hop neighborhoods. They trigger resynchronization periodically, according to the protocol described in the previous section. The level is included in the **sync_begin** packet. All nodes behave according to a simple rule. They maintain a local level variable, initialized with a sufficiently large value. If a node receives a **sync_begin** packet with a level value being truly smaller than its own level variable, the node accepts the packet, sets its own level to the received level value plus one, and becomes a root node of its own, starting synchronization with its neighbors after the triggering synchronization round has finished. In the other case, if the received level is larger than or equal to the own level variable, the **sync_begin** packet is simply dropped. The process continues in a recursive fashion until the fringe of the network is reached.

This way, nodes always synchronize over multiple hops with the closest reference node, and their level variable indicates the hop distance. If an already-synchronized node learns later about a closer reference node, it assigns itself a new level and starts resynchronization of its children.

The overhead (expressed as number of exchanged packets) of HRTS and RBS are compared by DAI and HAN [189] for a scenario with 200 nodes placed in an area of 400×400 m^2 square meters when varying the average node degree. RBS creates much higher overhead than HRTS, however, the overhead of HRTS increases when the node density increases. One explanation for this might be the observation that the broadcast domains of higher-level nodes are smaller than those of reference nodes because of overlapping transmission ranges of neighboring children. This way they have relatively fewer children.

The protocol has a further mechanism to restrict the depth of a synchronization tree rooted in a reference node. In addition to the level parameter and the identity of the answering node, each **sync_begin** packet carries also a depth parameter, which is decremented in every level of the tree. The tree construction stops upon reaching a depth of zero.

8.4 Further reading

- The TinySync and MiniSync protocols are presented by SICHITIU and VEERARITTIPHAN [764]. They belong to the class of sender/receiver protocols and consist of schemes for pair-wise synchronization and synchronizing a network. Similar to RBS, the nodes do not adjust their local clocks but compute conversion parameters for their respective phase shifts and drift rates. Let us assume that node i wants to synchronize with node j. At time $t_{0,1}$ (and local time $L_i(t_{0,1})$), node i sends a packet to node j, which timestamps it as $L_j(t_{1,1})$. Node j answers immediately and i receives the answer at $L_i(t_{2,1})$. Under the assumption of constant drift rate, we need to find coefficients $a_{i,j}$ and $b_{i,j}$ such that $L_j(t) = a_{i,j} \cdot L_i(t) + b_{i,j}$ holds for all t. From only a single three-way handshake, one can constrain the parameters $a_{i,j}$ and $b_{i,j}$ by considering that node j must have received the first packet *after* it has been sent by i. By the same token, node j must have sent the answer *before* node i has received it. To summarize, it must hold that

$$L_j(t_{1,1}) > a_{i,j} \cdot L_i(t_{0,1}) + b_{i,j}$$
$$L_j(t_{1,1}) < a_{i,j} \cdot L_i(t_{2,1}) + b_{i,j}.$$

Objectives of this Chapter

This chapter gives an overview of the methods to determine the symbolic location – "in the living room" – and the numeric position – "at coordinates (23.54, 11.87)" – of a wireless sensor node. The properties of such methods and the principal possibilities for a node to determine information about its whereabouts are discussed. The mathematical basics for positioning are introduced and the single-hop and multihop positioning case are described using several example systems.

At the end of the chapter, the reader will understand the principal design trade-offs for positioning and gain an appreciation for the overhead involved in obtaining this information.

This chapter introduces various techniques of how sensor nodes can learn their location automatically, either fully autonomically by relying on means of the WSN itself or by using some assistance from external infrastructure.

9.1 Properties of localization and positioning procedures

The simple intuition of "providing location information to a node" has a number of facets that should be classified to make the options for a location procedure clear. The most important properties are (following the survey papers [349, 350]):

Physical position versus symbolic location Does the system provide data about the *physical position* of a node (in some numeric coordinate system) or does a node learn about a *symbolic location* – for example, "living room", "office 123 in building 4"? Is it, in addition, possible to match physical position with a symbolic location name (out of possibly several applicable ones)?

While these two concepts are different, there is no consistent nomenclature in the literature – position and location are often used interchangeably. The tendency is to use "location" as the more general term. We have to rely on context to distinguish between these two contexts.

Absolute versus relative coordinates An absolute coordinate system is valid for all objects and embedded in some general frame of reference. For example, positions in the Universal Transverse Mercator (UTM) coordinates form an absolute coordinate system for any place on earth. Relative coordinates, on the other hand, can differ for any located object or set of objects – a WSN where nodes have coordinates that are correct with respect to each other but have no relationship to absolute coordinates is an example.

To provide absolute coordinates, a few **anchors** are necessary (at least three for a two-dimensional system). These anchors are nodes that know their own position in the absolute coordinate system. Anchors can rotate, translate, and possibly scale a relative coordinate system so that it coincides with the absolute coordinate system. These anchors are also commonly called *"beacons" or "landmarks"* in the literature.

Localized versus centralized computation Are any required computations performed locally, by the participants, on the basis of some locally available measurements or are measurements reported to a central station that computes positions or locations and distributes them back to the participants? Apart from scaling and efficiency considerations (both with respect to computational and communication overhead), privacy issues are important here as it might not be desirable for a participant to reveal its position to a central entity.

Accuracy and precision The two most important figures of merit for a localization system are the accuracy and the precision of its results. Positioning accuracy is the largest distance between the estimated and the true position of an entity (high accuracy indicates a small maximal mismatch). Precision is the ratio with which a given accuracy is reached, averaged over many repeated attempts to determine a position. For example, a system could claim to provide a 20-cm accuracy with at least 95 % precision. Evidently, accuracy and precision values only make sense when considered together, forming the accuracy/precision characteristic of a system.

Scale A system can be intended for different scales, for example – in indoor deployment – the size of a room or a building or – in outdoor deployment – a parking lot or even worldwide operation. Two important metrics here are the area the system can cover per unit of infrastructure and the number of locatable objects per unit of infrastructure per time interval.

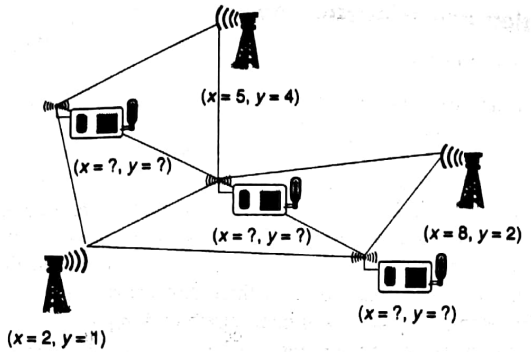


Figure 9.1 Determining the position of sensor nodes with the assistance from some anchor points; not all nodes are necessarily in contact with all anchors

Limitations For some positioning techniques, there are inherent deployment limitations. GPS, for example, does not work indoors; other systems have only limited ranges over which they operate.

Costs Positioning systems cause costs in time (infrastructure installation, administration), space (device size, space for infrastructure), energy (during operation), and capital (price of a node, infrastructure installation).

Figure 9.1 illustrates the positioning problem. The figures in this chapter use the "access point" icon to indicate anchors for easy distinction. It should be pointed out, however, that normal sensor nodes can just as well be used as anchors, as long as they have are aware of their position.

In addition, a positioning or localization system can be used to provide the recognition or classification of objects; this property is less important in the WSN context or, if used, usually not considered a part of the localization system.

9.2 Possible approaches

Three main approaches exist to determine a node's position: Using information about a node's neighborhood (proximity-based approaches), exploiting geometric properties of a given scenario (triangulation and trilateration), and trying to analyze characteristic properties of the position of a node in comparison with premeasured properties (scene analysis). The overview given here again considerably follows reference [350].

9.2.1 Proximity

The simplest technique is to exploit the finite range of wireless communication. It can be used to decide whether a node that wants to determine its position or location is in the proximity of an anchor. While this only provides coarse-grain information, it can be perfectly sufficient. One example is the natural restriction of infrared communication by walls, which can be used to provide a node with simple location information about the room it is in.

Proximity-based systems can be quite sophisticated and can even be used for approximate positioning when a node can analyze proximity information of several overlapping anchors (e.g. [106]). They can also be relatively robust to the uncertainties of the wireless channel – deciding whether a node is in the proximity of another node is tantamount to deciding connectivity, which can happen on relatively long time scales, averaging out short-term fluctuations.

9.2.2 Trilateration and triangulation

Lateration versus angulation

In addition to mere connectivity/proximity information, the communication between two nodes often allows to extract information about their geometric relationship. For example, the distance between two nodes or the angle in a triangle can be estimated – how this is done is discussed in the following two subsections. Using elementary geometry, this information can be used to derive information about node positions. When distances between entities are used, the approach is called **lateration**; when angles between nodes are used, one talks about **angulation**.

For lateration in a plane, the simplest case is for a node to have precise distance measurements to three noncolinear anchors. The extension to a three-dimensional space is trivial (four anchors are needed); all the following discussion will concentrate on the planar case for simplicity. Using distances and anchor positions, the node's position has to be at the intersection of three circles around the anchors (Figure 9.2).

The problem here is that, in reality, distance measurements are never perfect and the intersection of these three circles will, in general, not result in a single point. To overcome these imperfections, distance measurements form more that three anchors can be used, resulting in a **multilateration** problem. Multilateration is a core solution technique, used and reused in many concrete systems described below. Its mathematical details are treated in Section 9.3.

Angulation exploits the fact that in a triangle once the length of two sides and two angles are known the position of the third point is known as the intersection of the two remaining sides of the triangle. The problem of imprecise measurements arises here as well and can also be solved using multiple measurements.

Determining distances

To use (multi-)lateration, estimates of distances to anchor nodes are required. This **ranging** process[1] ideally leverages the facilities already present on a wireless node, in particular, the radio communication device. The characteristics of wireless communication are partially determined by the distance between sender and receiver, and if these characteristics can be measured at the receiver.

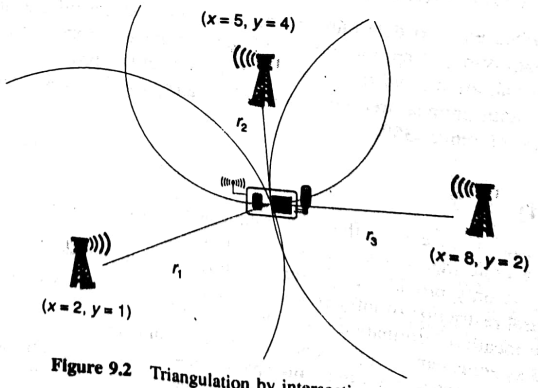


Figure 9.2 Triangulation by intersecting three circles

[1] Because of this name, proximity-based approaches are sometimes also called "range-free" approaches.

they can serve as an estimator of distance. The most important characteristics are Received Signal Strength Indicator (RSSI), Time of Arrival (ToA), and Time Difference of Arrival (TDoA).

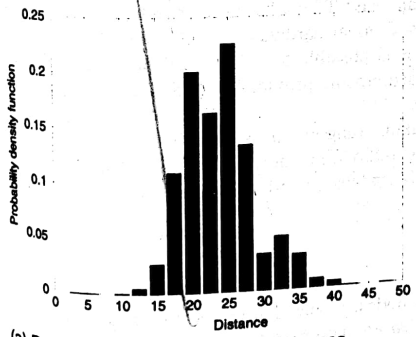
Received signal strength indicator

Assuming that the transmission power P_{tx}, the path loss model, and the path loss coefficient α are known, the receiver can use the received signal strength P_{rcvd} to solve for the distance d in a path loss equation like

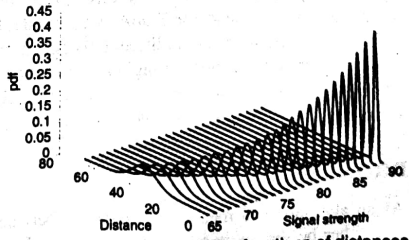
$$P_{rcvd} = c\frac{P_{tx}}{d^{\alpha}} \Leftrightarrow d = \sqrt[\alpha]{\frac{cP_{tx}}{P_{rcvd}}}.$$

This is appealing since no additional hardware is necessary and distance estimates can even be derived without additional overhead from communication that is taking place anyway. The disadvantage, however, is that RSSI values are not constant but can heavily oscillate, even when sender and receiver do not move. This is caused by effects like fast fading and mobility of the environment – ranging errors of ±50 % are reported, for example, by SAVARESE et al. [724]. To some degree, this effect can be counteracted by repeated measurements and filtering out incorrect values by statistical techniques [864]. In addition, simple, cheap radio transceivers are often not calibrated and the same actual signal strength can result in different RSSI values on different devices (reference [873] considers the calibration problem in detail); similarly, the actual transmission power of such a transceiver shows discrepancies from the intended power [725]. A third problem is the presence of obstacles in combination with multipath fading [104]. Here, the signal attenuation along an indirect path, which is higher than along a direct path, can lead to incorrectly assuming a longer distance than what is actually the case. As this is a structural problem, it cannot be combated by repeated measurements.

A more detailed consideration shows that mapping RSSI values to distances is actually a random process. RAMADURAI and SICHITIU [674], for example, collected, for several distances, repeated samples of RSSI values in an open field setup. Then, they counted how many times each distance resulted in a given RSSI value and computed the density of this random variable – Figure 9.3(a) shows this probability density function for a single given value of RSSI, Figure 9.3(b) for several. The information provided in particular by small RSSI values, indicating longer distances, is quite limited as the density is widely spread.



(a) Probability density function of distances resulting in a given RSSI value



(b) Several probability density functions of distances for various given RSSI values

Figure 9.3 Random nature of mapping RSSI values to distances [674]. Reproduced by permission of V. Ramadurai and M. L. Sichitiu

Hence, when using RSSI as a ranging technique, it is necessary to accept and deal with considerable ranging errors or to treat the outcome of the ranging process as a stochastic result to begin with.

Time of arrival

Time of Arrival (ToA) (also sometimes called *"time of flight"*) exploits the relationship between distance and transmission time when the propagation speed is known. Assuming both sender and receiver know the time when a transmission – for example, a short ultrasound pulse – starts, the time of arrival of this transmission at the receiver can be used to compute propagation time and, thus, distance. To relieve the receiver of this duty, it can return any received "measurement pulse" in a deterministic time; the original sender then only has to measure the round trip time assuming symmetric paths.

Depending on the transmission medium that is used, time of arrival requires very high resolution clocks to produce results of acceptable accuracy. For sound waves, these resolution requirements are modest; they are very hard for radio wave propagation.

One disadvantage of sound is that its propagation speed depends on external factors such as temperature or humidity – careful calibration is necessary but not obvious.

Time difference of arrival

To overcome the need for explicit synchronization, the Time Difference of Arrival (TDoA) method utilizes implicit synchronization by directly providing the start of transmission information to the receiver. This can be done if two transmission mediums of very different propagation speeds are used – for example, radio waves propagating at the speed of light and ultrasound, with a different in speed of about six orders of magnitude.[2] Hence, when a sender starts an ultrasound and a radio transmission simultaneously, the receiver can use the arrival of the radio transmission to start measuring the time until arrival of the ultrasound transmission, safely ignoring the propagation time of the radio communication.[3]

The obvious disadvantage of this approach is the need for two types of senders and receivers on each node. The advantage, on the other hand, is a considerably better accuracy compared to RSSI-based approaches. This concept and variations of it have been used in various research efforts [659, 725] and accuracies of down to 2 cm have been reported [725].

Discussion

There is a clear trade-off between ranging error and complexity/cost. RSSI-based approaches are simpler and work with hardware that is required anyway; TDoA-based approaches provide superior ranging results but need more complex and additional hardware, which also adds to energy consumption. The open question is thus whether it is possible to solve the actual positioning or localization problem based on the error-prone measurements provided by RSSI-based approaches or whether the overhead for TDoA is unavoidable.

There is a number of additional work on making ranging measurements more reliable, for example, by using acoustic ranging as opposed to radio frequency attenuation [291] or by considering the effects introduced by quantizing RSSI values (in effect, proximity systems can be regarded as yes/no quantizations of RSSI) [624].

Determining angles

As an alternative to measuring distances between nodes, angles can be measured. Such an angle can either be the angle of a connecting line between an anchor and a position-unaware node to a

[2] Speed of light in vacuum: 299,792,458 m/s, ultrasound in air about 344 m/s at 21 °C.

[3] In fact, this is precisely what happens when one estimates the distance to a thunderstorm by counting the seconds between lightning and thunder.

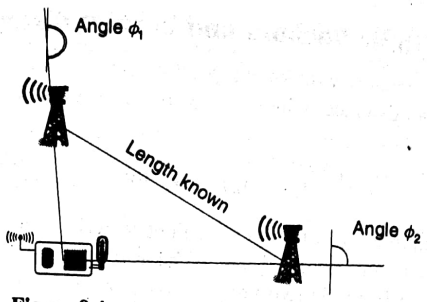


Figure 9.4 Angulation based on two anchors

given reference direction ("0° north"). It can also be the angle between two such connecting lines if no reference direction is commonly known to all nodes (Figure 9.4).

A traditional approach to measuring angles is to use directional antennas (antennas that only send to/receive from a given direction), rotating on their axis, similar to a radar station or a conventional lighthouse. This makes angle measurements conceptually simple, but such devices are quite inappropriate for sensors nodes; they can be useful for supporting infrastructure anchors.

Another technique is to exploit the finite propagation speed of all waveforms. With multiple antennas mounted on a device at known separation and measuring the time difference between a signal's arrival at the different antennas, the direction from which a wavefront arrived at the device can be computed. The smaller the antenna separation, the higher the precision of the time differences has to be, which results in strenuous timing requirements given the desirable small size of sensor nodes.

Overall, angulation is a less frequently discussed technique compared to lateration; Section 9.4 discusses some examples.

9.2.3 Scene analysis

A quite different technique is **scene analysis**. The most evident form of it is to analyze pictures taken by a camera and to try to derive the position from this picture. This requires substantial computational effort and is hardly appropriate for sensor nodes.

But apart from visual pictures, other measurable characteristic "fingerprints" of a given location can be used for scene analysis, for example, radio wave propagation patterns. One option is to use signal strength measurements of (one or more anchors) transmitting a known signal strength and compare the actually measured values with those stored in a database of previously off-line measured values for each location – the RADAR system [35] is one example that uses this approach to determine positions in a building. Using other physical characteristics such as multipath behavior is also conceivable.

While scene analysis is interesting for systems that have a dedicated deployment phase and where off-line measurements are acceptable, this is not always the case for WSNs. Hence, this approach is not the main focus of attention.

9.3 Mathematical basics for the lateration problem

Since (multi)lateration is one of the most popular techniques for positioning applied in WSNs and serves as a primitive building block for some of the approaches discussed later, it is worthwhile to have a closer look at the mathematics behind it.