Chapter 8
Time Synchronization
Goals of this chapter

- Understand the importance of time synchronization in WSNs

- Understand typical strategies for time synchronization and how they are applied in WSNs
The Time Synchronization Problem
The role of time in WSNs

- Time synchronization is needed for WSN applications and protocols:
  - Applications: beamforming
  - Protocols: TDMA, protocols with coordinated wakeup, ...
  - Distributed debugging: timestamping of distributed events is needed to figure out their correct order of appearance

- WSNs have a direct coupling to the physical world, hence their notion of time should be related to **physical time**:
  - physical time = wall clock time, real-time, *i.e.* one second of a WSN clock should be close to one second of real time
  - Commonly agreed time scale for real time is UTC, generated from atomic clocks and modified by insertion of leap seconds to keep in synch with astronomical timescales (one rotation of earth)
  - Other concept: logical time, where only the relative ordering of events counts but not their relation to real time
Clocks in WSN nodes

- Often, a **hardware clock** is present:
  - An oscillator generates pulses at a fixed nominal frequency
  - A counter register is incremented after a fixed number of pulses
    - Only register content is available to software
    - Register change rate gives achievable time resolution
  - Node $i$’s register value at real time $t$ is $H_i(t)$
    - Convention: small letters (like $t$, $t'$) denote real physical times, capital letters denote timestamps or anything else visible to nodes

- A (node-local) software clock is usually derived as follows:
  - $L_i(t) = \theta_i H_i(t) + \phi_i$ (not considering overruns of the counter-register)
  - $\theta_i$ is the drift rate, $\phi_i$ is the phase shift
  - Time synchronization algorithms modify $\theta_i$ and $\phi_i$, but not the counter register
Synchronization accuracy / agreement

- **External synchronization:**
  - Synchronization with external real time scale like UTC
  - Nodes $i=1, \ldots, n$ are **accurate** at time $t$ within bound $\delta$ when $|L_i(t) - t| < \delta$ for all $i$
    - Hence, at least one node must have access to the external time scale

- **Internal synchronization**
  - No external timescale, nodes must agree on common time
  - Nodes $i=1, \ldots, n$ **agree** on time within bound $\delta$ when $|L_i(t) - L_j(t)| < \delta$ for all $i$ and $j$
Sources of inaccuracies

- Nodes are switched on at random times, initial phases $\phi_i$ hence can be random.
- Actual oscillators have random deviations from nominal frequency (drift, clock skew).
  - Drifts are specified in ppm (parts per million), the ppm value counts the additional or missing oscillations over the time of one million oscillations at nominal rate.
  - The cheaper the oscillators, the larger the average deviation
    - For sensor nodes, values between 1ppm (1s error every 11.6 days) and 100ppm (1s error every 2.78 hours) are assumed. Berkeley motes have an average drift of 40ppm.
- Oscillator frequency depends on time (oscillator aging) and environment (temperature, pressure, supply voltage, ...).
  - Especially the time-dependent drift rates call for frequent re-synchronization, as one-time synchronization is not sufficient.
  - However, stability over tens of minutes is often a reasonable assumption.
General properties of time synchronization algorithms

1. Physical time vs. logical time
2. External vs. internal synchronization
3. Global vs. local algorithms
   - Keep all nodes of a WSN synchronized or only a local neighborhood?
4. Absolute vs. relative time
5. Hardware vs. software-based mechanisms
   - A GPS receiver would be a hardware solution, but often too heavyweight/costly/energy-consuming in WSN nodes, and in addition a line-of-sight to at least four satellites is required
6. A-priori vs. a-posteriori (post-facto) synchronization
   - Is time synchronization achieved before or after an interesting event?
7. Deterministic vs. stochastic precision bounds
8. Local clock update discipline
   - Avoid backward jumps
   - Avoid sudden jumps
Performance metrics and fundamental structure

- **Metrics:**
  - Precision: maximum synchronization error for deterministic algorithms, error mean / std dev / quantiles for stochastic ones
  - Energy costs: # of exchanged packets, computational costs, resynchronization frequency
  - Memory requirements: history of previous time synchronization packets needed for drift rates estimation
  - Fault tolerance: failing nodes, error-prone links, network partition, mobility

- **Fundamental building blocks:**
  - Resynchronization event detection block: when to trigger a time synchronization round? Periodically? After external event?
  - Remote clock estimation block: figuring out the other nodes clocks with the help of exchanging packets
  - Clock correction block: compute adjustments for own local clock based on estimated clocks of other nodes
  - Synchronization mesh setup block: figure out which node synchronizes with which other nodes
Constraints for Time Synchronization in WSNs

- Should scale to large networks of unreliable nodes
- Quite diverse precision requirements, from microseconds to tens of seconds
- Use of extra hardware (like GPS receivers) is mostly not an option
- Low mobility and a node can reach its neighbors at any time
- Often there are no fixed upper bounds on packet delivery times (due to MAC delays, buffering, ...)
- Negligible propagation delay between neighboring nodes
- Manual node configuration is not an option
Protocols Based on Sender/Receiver Synchronization
Protocols based on sender/receiver synchronization

- A receiver synchronizes to the clock of a sender.

- Two steps:
  1. Pair-wise synchronization: how does a single receiver synchronize to a single sender?
  2. Network-wide synchronization: how to figure out who synchronizes with whom to keep the whole network (parts of it) synchronized?

- The classical NTP protocol belongs to this class.
LTS – Lightweight Time Synchronization

- Overall goal: synchronize the clocks of all sensor nodes / a subset of nodes to one reference clock (equipped with GPS receivers)
- It can synchronize the whole network (parts of it) and support post-facto synchronization
- It considers only phase shifts and does not try to correct different drift rates
- Two components:
  - Pair-wise synchronization: based on sender/receiver technique
  - Network-wide synchronization: minimum spanning tree construction with reference node as root
LTS – Pairwise Synchronization

1. Trigger resynchronization
2. Format synch packet
3. Timestamp packet with $L_i(t_1)$
4. Hand over packet for transmission
5. Operating system, channel access
6. Start packet transmission
7. Propagation delay
8. Packet transmission time
9. Packet reception interrupt
10. Timestamp with $L_j(t_5)$
11. Format synch answer packet
12. Timestamp with $L_j(t_6)$
13. Hand over packet for transmission
14. OS, Channel access
15. Start packet transmission
16. Packet reception interrupt
17. Timestamp with $L_i(t_8)$
LTS – Pairwise Synchronization

- Node $i$ wants to synchronize its clock to node $j$’s clock
- Timeline:
  - Node $i$ triggers resynchronization at time $t_0$, formats packet, timestamps it at $t_1$ with $L_i(t_1)$ and hands it over to transmission (with $L_i(t_1)$ as payload)
  - At $t_2$ the first bit appears on the channel, at $t_3$ the receiver receives last bit, packet reception is signaled at $t_4$, and at $t_5$ node $j$ timestamps it with $L_j(t_5)$
  - Node $j$ formats answer packet, timestamps it at time $t_6$ with $L_j(t_6)$ and hands it over for transmission – as payload the timestamps $L_i(t_1), L_j(t_5)$ and $L_j(t_6)$ are included
  - The arrival of the answer packet is signaled at time $t_7$ to node $i$, and $i$ timestamps it afterwards with $L_i(t_8)$
  - After time $t_8$, node $i$ has four values: $L_i(t_1), L_j(t_5), L_j(t_6)$ and $L_i(t_8)$, and wants to estimate its clock offset to node $j$. 
LTS – Pairwise Synchronization

**Assumptions:**

- Node \( i \)'s original aim is to estimate the true offset \( O = \Delta(t_1) = L_i(t_1) - L_j(t_1) \)
- During the whole process the drift is negligible
  \(\Rightarrow\) the algorithm in fact estimates \( \Delta(t_5) \) and assumes \( \Delta(t_5) = \Delta(t_1) \)
- Propagation delay \( \tau \) is the same in both directions, request and answer packets have duration \( t_p \), both parameters are known to \( i \)

**Approach:**

- Node \( i \) estimates \( \Delta(t_5) = L_i(t_5) - L_j(t_5) \) and therefore needs to estimate \( L_i(t_5) \), which is generated “somewhere” between \( t_1 \) and \( t_8 \)
- When \( t_8 - t_1 \) is very small, we might be willing to approximate \( O \) as \( \frac{1}{4} L_i(t_1) - L_j(t_5) \) or as \( \frac{1}{4} L_i(t_8) - L_j(t_5) \)

**But we can reduce uncertainty:**

- after \( t_1 \) we have at least one propagation delay and packet transmission time (for the request packet)
- before \( t_8 \) we have another propagation delay and packet transmission time (for the response packet)
- There also passes time between \( L_j(t_5) \) and \( L_j(t_6) \), and \( L_i(t_5) \) must be before this interval
LTS – Pairwise Synchronization

- Under the assumption that the remaining uncertainty is allocated equally to both $i$ and $j$, node $i$ can estimate $L_j(t_5)$ as

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

- This means:

$$0 = \Delta(t_5) = L_i(t_5) - L_j(t_5) = \frac{L_i(t_8) + L_i(t_1) - L_j(t_6) - L_j(t_5)}{2}$$
LTS – Pairwise Synchronization -- Discussion

- Node $i$ can figure out the offset to node $j$ based on the known values $L_i(t_1), L_j(t_5), L_j(t_6), L_i(t_8)$

- This exchange takes two packets – if node $j$ should also learn about the offset, a third packet is needed from $i$ to $j$ carrying $O$

- The uncertainty is in the interval 
  
  \[ I = [L_i(t_1) + \tau + t_p, L_i(t_8) - \tau - t_p - (L_j(t_6) - L_j(t_5))] \]

  and by picking the mid-point of the interval as $L_i(t_5)$, the maximum uncertainty is $|I|/2$
LTS – Pairwise Synchronization -- Discussion

- **Sources of inaccuracies:**
  - MAC delay
  - interrupt latencies upon receiving packets
  - Delays between packet interrupts and timestamping operation
  - Delay in operating system and protocol stack

- **Improvements:**
  - Let $i$ timestamp its packet after the MAC delay, immediately before transmitting the first bit
    - This would remove the uncertainty due to $i$’s operating system / protocol stack and the MAC delay!
  - Let $j$ timestamp receive packets as early as possible, e.g. in the interrupt routine
    - this removes the delay between packet interrupts and timestamping from the uncertainty, and leaves only interrupt latencies.
LTS – Networkwide Synchronization

- Given one reference node \( R \), to which all other nodes (a subset of nodes) want to synchronize
  - \( R \)'s direct neighbors (level-1 neighbors) synchronize with \( R \)
  - Two-hop (level-2) neighbors synchronize with level-1 neighbors
  - ....

- This way a spanning tree is created

- Should not allow arbitrary spanning trees:
  - Consider a node \( i \) with hop distance \( h_i \) to the root node
  - Assume that:
    - all synchronization errors are independent
    - all synch errors are identically normally distributed with zero mean and variance \( 4\sigma^2 \)
  - Then node \( i \)'s synchronization error is a zero-mean normal with variance \( h_i4 \sigma^2 \)
  - Hence, LTS aims to construct a spanning tree of minimum height and only node pairs along the edges of the tree are synchronized.
LTS – Centralized Multihop LTS

- Reference node $R$ triggers construction of a spanning tree, it first synchronizes its neighbors, then the level-1 neighbors synchronize level-2 neighbors and so on.

- Different distributed algorithms for construction of spanning tree can be used, e.g. DDFS, Echo algorithm.

- Communication costs:
  - Costs for construction of spanning tree
  - Synchronizing two nodes costs 3 packets, synchronizing $n$ nodes costs on the order of $3n$ packets.
LTS – Distributed Multihop LTS

- No explicit construction of spanning tree, but each node knows identity of reference node(s) and routes to them
- When node 1 wants to synchronize with R, an appropriate request travels to R – following this, 4 synchronizes to R, 3 synchronizes to 4, 2 synchronizes to 3, 1 synchronizes to 2
  - By-product: nodes 2, 3, and 4 are synchronized with R
- Minimum spanning tree constructed implicitly: node 1 should know shortest route to the closest reference node
When node 5 wants to synchronize with $R$, it can:

- issue its own synchronization request using route over 3, 4 and put additional synchronization burden on them
- ask in its local neighborhood whether someone is synchronized or has an ongoing synchronization request and benefit from that later on
- Enforce usage of path over 7, 8 (path diversification) to also synchronize these nodes

Discussion:

- Simulation shows that distributed multihop LTS needs more packets (between 40% and 100%) when all nodes have to be synchronized, even with optimizations
- Distributed multihop LTS allows to synchronize only the minimally required set of nodes $\rightarrow$ post-facto synchronization
Other Sender/Receiver-based Protocols

- These protocols work similar to LTS, with some differences in:
  - Method of spanning tree construction
  - How and when to take timestamps
  - How to achieve post-facto synchronization
- One variant: TPSN (Timing-Sync Protocol for Sensor Nets)
  - Pairwise-protocol similar to LTS, but timestamping at node \( i \) happens immediately before first bit appears on the medium, and timestamping at node \( j \) happens in interrupt routine
  - Networkwise=protocol constructs spanning tree based on level-discovery protocol:
    - root issues \( \text{level\_discovery} \) packet with level 0
    - neighbors assign themselves level \( 1+\text{level} \) value from \( \text{level\_discovery} \)
    - neighbors wait for some random time before they issue \( \text{level\_discovery} \) packets indicating their own level
    - Nodes missing \( \text{level\_discovery} \) packets for long time ask their neighborhood
Protocols Based on Receiver/Receiver Synchronization
Protocols based on receiver/receiver synchronization

• In this class of schemes the receivers of packets synchronize among each other, not with the transmitter of the packet

• **RBS = Reference Broadcast Synchronization**
  1. Synchronize receivers within a single broadcast domain
  2. A scheme for relating timestamps between nodes in different domains

• RBS does not modify the local clocks of nodes, but computes a table of conversion parameters for each peer in a broadcast domain

• RBS allows for post-facto synchronization
RBS – Synchronization in a Broadcast Domain

Packet reception interrupt

Timestamp with $L_i(t_{3,i})$

Send ($L_i(t_{3,i}), R, s$)

Packet reception interrupt

Timestamp with $L_j(t_{3,j})$

Send ($L_j(t_{3,j}), R, s$)

Packet $R, s$

Receiver uncertainty

$t_0$

$t_1,i$

$t_2,i$

$t_3,i$

$t_1,j$

$t_2,j$

$t_3,j$
RBS – Synchronization in a Broadcast Domain

- The goal is to synchronize i’s and j’s clocks to each other
- Timeline:
  - Reference node R broadcasts at time $t_0$ some synchronization packet carrying its ID $R$ and a sequence number $s$
  - Receiver i receives the last bit at time $t_{1,i}$, gets the packet interrupt at time $t_{2,i}$ and timestamps it at time $t_{3,i}$
  - Receiver j is doing the same
  - At some later time node i transmits its observation $(L_i(t_{3,i}), R, s)$ to node j
  - At some later time node j transmits its observation $(L_j(t_{3,j}), R, s)$ to node i
  - The whole procedure is repeated periodically, the reference node transmits its synchronization packets with increasing sequence numbers
    - $R$ could also use ordinary data packets as long as they have sequence numbers ...
  - Under the assumption $t_{3,i} = t_{3,j}$ node j can figure out the offset $O_{i,j} = L_j(t_{3,j}) - L_i(t_{3,i})$ after receiving node i’s final packet – of course, node i can do the same
Synchronization error:
- Difference between $t_{3,i}$ and $t_{3,j}$
- Drift between $t_{3,i}$ and the time where node $i$ transmits its observations to $j$

However:
- In small broadcast domains and when received packets are timestamped as early as possible, the difference between $t_{3,i}$ and $t_{3,j}$ is very small
  - As compared to sender/receiver based schemes the MAC delay and operating system delays experienced by the reference node play no role!!
- Drift can be neglected when observations are exchanged quickly after reference packets
- Drift can be estimated jointly with Offset $O$ when a number of periodic observations of $O_{i,j}$ have been collected
  - This amounts to a standard least-squares line regression problem
RBS – Network Synchronization
RBS – Network Synchronization

- Suppose that:
  - node 1 has detected an event at time $L_1(t)$
  - the sink is connected to a GPS receiver and has UTC timescale
  - node 1 wants to inform the sink about the event such that the sink receives a timestamp in UTC timescale
  - Broadcast domains are indicated by “circles”

- Timestamp conversion approach:
  - Idea: do not synchronize all nodes to UTC time, but convert timestamps as packet is forwarded from node 1 to the sink $\Rightarrow$ avoids global synch!!!
  - Node 1 picks node 3 as forwarder – as they are both in the same broadcast domain, node 1 can convert the timestamp $L_1(t)$ into $L_3(t)$
  - Node 3 picks node 5 in the same way
  - Node 5 is member in two broadcast domains and knows also the conversion parameters for the next forwarder 9
  - And so on ...
  - Result: the sink receives a timestamp in UTC timescale!
  - Nodes 5, 8 and 9 are gateway nodes!
Reading Assignment


Questions:

1. Show how to derive the offsets (O) of RBS and TPSN. Also show their synchronization error bounds.
2. Compare RBS with TPSN.