The Collection Tree Protocol for the Castalia Wireless Sensor Networks Simulator

Ugo Colesanti
Dipartimento di Informatica e Sistemistica
Sapienza Università di Roma
colesanti@dis.uniroma1.it

Silvia Santini
Institute for Pervasive Computing
ETH Zurich
santinis@inf.ethz.ch

Technical Report Nr. 729
Department of Computer Science
ETH Zurich
June, 2011

Abstract

The Collection Tree Protocol (CTP) is a well-know protocol that provides a reliable collection service for wireless sensor networks. In this report, we describe our implementation of CTP for the version 3.0 of the Castalia wireless sensor networks simulator. Besides being a reference for researchers interested in experimenting with CTP in Castalia, this report also provides a throughout description of the mechanisms of CTP. As the structure of our Castalia-based implementation mimics that of CTP's TinyOS 2.1 components, the report also offers several insights into the details of such components. A former implementation of CTP for Castalia 1.3, which we had described in previous work [8], is no longer supported. The module implementing CTP for Castalia 3.0 is publicly available at `http://code.google.com/p/ctp-castalia`.

Contents

1 Introduction

Within the last decade, a plethora of algorithms and protocols to perform data collection in wireless sensor networks (WSNs) have been developed [29, 6, 13, 16]. Among these, the Collection Tree Protocol (CTP) is widely regarded as the reference protocol for data collection [11, 13, 14]. CTP provides for "*best-effort anycast datagram communication to one of the collection roots in a network*" [11] and its specification is provided in TinyOS[1] Enhancement Proposal 123 [11]. Gnawali *et al.* [13, 14] also report a throughout performance evaluation of CTP in realistic settings, demonstrating the ability of the protocol to reliably and efficiently report data to one or more collectors. The current distribution of TinyOS, also includes an implementation of CTP. Writing TinyOS applications that make use of CTP and run on either actual prototyping platforms or on the TOSSIM simulator is thus straightforward.

When developing new algorithms or applications for WSNs, however, it is often useful to experiment with different simulators. This allows to gain a better understanding of the problem at hand as well as to ensure that obtained experimental results are consistent across different platforms and scenarios. Furthermore, different simulators may offer different levels of details. The Castalia simulator for WSNs, for instance, provides a generic platform to perform "*first order validation of an algorithm before moving to an implementation on a specific platform*" [1]. It is based on the well-known OMNeT++ simulation environment, which mainly provides for Castalia's modularity. Castalia allows to integrate several different channel models and MAC protocols in the simulations, it provides tools to quickly inspect and plot simulation results and it is straightforward to install and use. In general, Castalia represents an excellent choice for WSN developers that aim at studying their algorithms in a well-designed simulation environment before moving to a more complex, and often more cumbersome, TinyOS-based implementation.

The current standard distribution of Castalia, however, does not include an implementation of CTP. We therefore implemented a corresponding CTP module whose detailed description is provided in the following sections. Besides being a reference for researchers interested in experimenting with CTP in Castalia, this report also provides a throughout description of the mechanisms of CTP. As the structure of our Castalia-based implementation closely resembles that of CTP's TinyOS 2.1 components, the report also offers several insights into the details of such components. A former implementation of CTP for Castalia 1.3, which we had described in previous work [8], is no longer supported. The module implementing CTP for Castalia 3.0 (and its subsequent updates) is publicly available at `http://code.google.com/p/ctp-castalia/`.

In the remainder of this report we will first provide some background information about Castalia and CTP in section 2. We will then focus on the description of our implementation in section 3 and outline relevant MAC-layer issues in section 4. Finally, section 5 concludes the report.

2 Background

As stated in TinyOS TEP 119, data collection is one of the fundamental primitives for implementing WSN applications [12]. A typical collection protocol provides for the construction and maintenance of one or more routing trees having each a so-called *sink node* as their root. A

[1] TinyOS is a widely used operating system for WSNs. For more information please refer to the TinyOS project website: `http://www.tinyos.net/`.

sink can store the received packets or forward them to an external network, typically through a reliable and possibly wired communication link. Within the network, nodes forward packets through the routing tree up to (at least) one of the sinks. To this end, each node selects one of its neighboring nodes as its *parent*. Nodes acting as parents are responsible of handling the packets they receive from their *children* and further forwarding them towards the sink. To construct and maintain a routing tree a collection protocol must thus first of all define a metric each node can use to select a parent. The distance in hops to the sink or the quality of the local communication link (or a function thereof) can for instance be used as metrics for parent selection. In either cases, nodes need to collect information about their neighboring nodes in order to compute the parent selection metric. To this end, nodes regularly exchange corresponding messages, usually called *beacons*, that contain information about, e.g., the (estimated) distance in hops of the node to the sink or its residual energy.

Collection protocols mainly differ in the definition of the parent selection metric and the way they handle critical situations like the detection and repairing of routing loops. On this regard, TinyOS TEP 119 specifies the requirements a collection protocol for WSNs must be able to comply with. First of all, it should be able to properly estimate the (1-hop) link quality. Second, it must have a mechanism to detect (and repair) routing loops. Last but not least, it should be able to detect and suppress duplicate packets, which can be generated as a consequence of lost acknowledgments.

Although these requirements may sound simple to fulfill, collection protocols providing for high data delivery ratios are hard to design and develop. The main factor hampering the performance of such protocols is the instability of wireless links. In particular, as pointed out in [13], the quality of a link may vary significantly, and quickly, over time. Also, the estimation of the link quality is often based on correctly received packets only; clearly, this introduce a bias in the estimation since information about dropped packets is lost. CTP directly addresses these problems and can reach excellent delivery performance, as shown in [14, 8]. CTP, which is described in detail below, became quickly popular within the WSNs research community [11, 17, 18]. Nonetheless, a CTP module supporting several WSN hardware platforms (MicaZ, Telosb/TmoteSky, TinyNode) is available for the TinyOS 2.1 distribution.

2.1 The Collection Tree Protocol (CTP)

CTP uses routing messages (also called *beacons*) for tree construction and maintenance, and *data* messages to report application data to the sink. The standard implementation of CTP described in [11] and evaluated in [13, 14] consists of three main logical software components: the *Routing Engine* (RE), the *Forwarding Engine* (FE), and the *Link Estimator* (LE). In the following, we will focus on the main role taken over by these three components, while in section 3 we will provide amore in-depth descriptions of their features.

Routing Engine. The Routing Engine, an instance of which runs on each node, takes care of sending and receiving beacons as well as creating and updating the *routing table*. This table holds a list of neighbors from which the node can select its parent in the routing tree. The table is filled using the information extracted from the beacons. Along with the identifier of the neighboring nodes, the routing table holds further information, like a metric indicating the "quality" of a node as a potential parent.

In the case of CTP, this metric is the ETX (Expected Transmissions), which is communicated by a node to its neighbors through beacons exchange. A node having an ETX equal to

n is expected to be able to deliver a data packet to the sink with a total of n transmissions, on average. The ETX of a node is defined as the "*ETX of its parent plus the ETX of its link to its parent*" [11]. More precisely, a node first computes, for each of its neighbors, the link quality of the current node-neighbor link. This metric, to which we refer to as the *1-hop ETX*, or ETX_{1hop}, is computed by the LE. For each of its neighbors the node then sums up the 1-hop ETX with the ETX the corresponding neighbors had declared in their routing beacons. The result of this sum is the metric which we call the *multi-hop ETX*, or ETX_{mhop}. Since the ETX_{mhop} of a neighbor quantifies the expected number of transmissions required to deliver a packet to a sink using that neighbor as a relay, the node clearly selects the neighbor corresponding to the lowest ETX_{mhop} as its parent. The value of this ETX_{mhop} is then included by the node in its own beacons so as to enable lower level nodes to compute their own ETX_{mhop}. Clearly, the ETX_{mhop} of a sink node is always 0. For the interested reader, section 3.2 provides an in-depth description of the procedure that computes the ETX.

The frequency at which CTP beacons are sent is set by the *Trickle* algorithm [20]. Using Trickle, each node progressively reduces the sending rate of the beacons so as to save energy and bandwidth. The occurrence of specific events such as route discovery requests may however trigger a reset of the sending rate. Such resets are necessary in order to make CTP able to quickly react to topology or environmental changes, as we will also detail in section 3.3.

Forwarding Engine. The Forwarding Engine, as the name says, takes care of forwarding data packets which may either come from the application layer of the same node or from neighboring nodes. As we will detail in section 3.4, the FE is also responsible of detecting and repairing routing loops as well as suppressing duplicate packets. As mentioned above, the ability of detecting and repairing routing loops and the handling of duplicate packets are two of the tree features TinyOS TEP 119 requires to be part of a collection protocol [12]. The third one, i.e., a mean to estimate the 1-hop link quality, is handled in CTP by the Link Estimator.

Link Estimator. The Link Estimator takes care of determining the inbound and outbound quality of 1-hop communication links. As mentioned before, we refer to the metric that expresses the quality of such links as the 1-hop ETX. The LE computes the 1-hop ETX by collecting statistics over the number of beacons received and the number of successfully transmitted data packets. From these statistics, the LE computes the inbound metric as the ratio between the total number of beacons sent by the neighbor over the fraction of received beacons. Similarly, the outbound metric represents the expected number of transmission attempts required by the node to successfully deliver a data packet to its neighbor.

To gather the necessary statistics and compute the 1-hop ETX, the LE adds a 2 byte header and a variable length footer to outgoing routing beacons. To this end, as shown in figure 1, routing beacons are passed over by the RE to the LE before transmission. The fine-grained structure of LE's header and footer will be described in section 3.2.

Upon reception of a beacon, the LE extracts the information from both the header and footer and includes it in the so-called *link estimator table* [13]. This table holds a list of identifiers of the neighbors of a node, along with related information, like their 1-hop ETX or the amount of time elapsed since the ETX of a specific neighbor has been updated. In contrast to the routing table, which is maintained by the RE, the link estimator table is created and updated by the LE. These tables are however tightly coupled. For instance, the RE can force

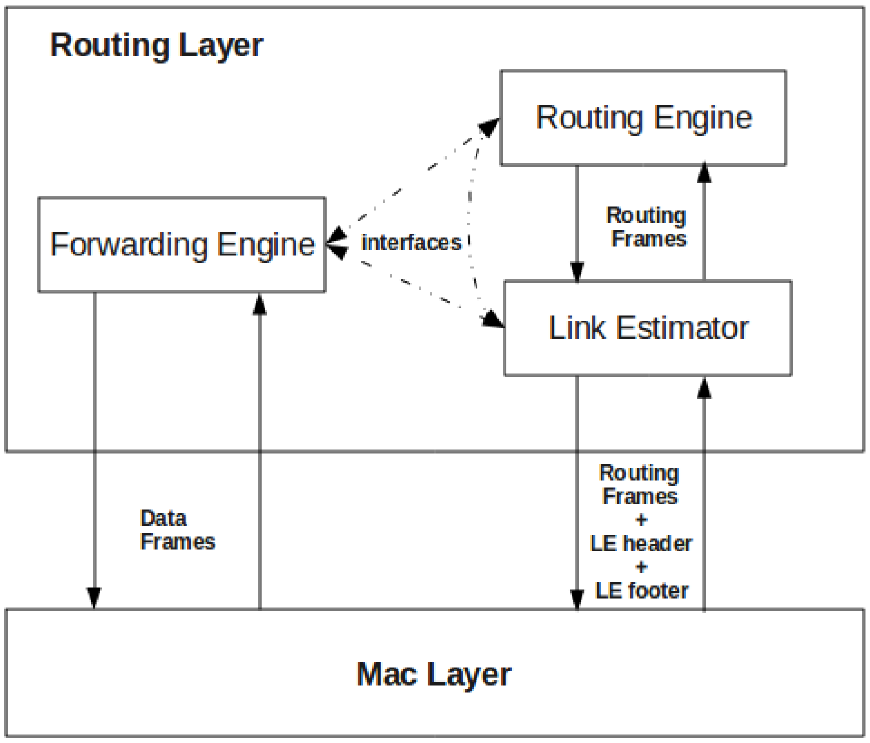

Figure 1: Message flow and modules interactions.

the LE to add an entry to the link estimator table or to block a removal. The latter case occurs when one of the neighbors is the sink node. Similarly, the LE can signal the eviction of a specific node from the link estimator table to the RE, which in turn accordingly removes the entry corresponding to the same node in the routing table. The information available from the link estimator table is used to fill the footer of outgoing beacons, so that nodes can efficiently share neighborhood information. However, the space available in the footer may not be sufficient to include all the entries of the neighborhood table in one single beacon. Therefore, the entries to send are selected following a round robin procedure over the link estimator table.

Interfaces. As we will also detail in the following section 3, the three components RE, FE, and LE do not work independently but interact through a set of well-defined interfaces. For instance, the RE needs to pull the 1-hop ETX metric from the LE to compute the multi-hop ETX. On the other side, the FE must obtain the identifier of the current parent from the RE and check the congestion status of the neighbors with the RE. As schematically depicted in figure 1, these interactions are managed by specific interfaces. In the following section 3, we will explicitly mention these interfaces whenever necessary or appropriate.

2.2 Castalia and OMNeT^{++}

There exist a plethora of different frameworks that provide comfortable simulation environments for WSNs applications. Their survey is beyond the scope of this report and the interested reader is referred to [15, 10]. In the context of our work, we are interested in simulation environments that provide for modularity, realistic radio and channel models, and, at the same time, comfortable programming. To the best of our knowledge, among the currently available frameworks the Castalia WSNs simulator emerges for its quality and completeness [21, 17, 22, 27]. Castalia provides a generic platform to perform "*first order validation of an algorithm before moving to an implementation on a specific platform*" [1]. It is based on the well-known OMNeT^{++} simulation environment, which mainly provides for Castalia's modularity.

OMNeT^{++} is a discrete event simulation environment that thanks to its excellent modularity is particularly suited to support frameworks for specialized research fields. For instance, it supports the Mobility Framework (MF) to simulate mobile networks, or the INET framework that models several internet protocols. OMNeT^{++} is written in C++, is well documented and features a graphical environment that eases development and debugging. Additionally, a wide community of contributors supports OMNeT^{++} by continuously providing updates and new frameworks. The comfortable initial training, the modularity, the possibility of programming in an object-oriented language (C^{+}+), are among the reasons that led us to prefer the OMNeT^{++} platform, and thus Castalia, over other available network simulators like the well-known ns2 and the related extensions for WSNs (e.g., SensorSim [24]). Nonetheless, in the last years Castalia has been continuously improved [4, 25] and there is an increasing number of researchers using Castalia to support their investigations [5, 28, 3, 18, 2].

In the context of this work, we make use of version 3.0 of Castalia, which builds upon version 4.1 of OMNeT^{++}. In this version, Castalia features advanced channel and radio models, a MAC protocol with large number of tunable parameters and a highly flexible model for simulating physical processes. Castalia offers exhaustive models for simulating both the radio channel and the physical layer of the radio module. In particular, it provides bundled support for the $CC2420$ radio controller, which is the transceiver of choice for the TelosB/TmoteSky platform [9, 26]. This is particularly relevant to us since the Tmote Sky is our reference hardware platform.

3 Implementation of CTP

Our CTP module for Castalia mimics TinyOS 2.1's reference implementation of CTP [11, 13]. Figure 2 shows the basic architecture of the *CtpNoe* compound module, which includes the modules *Ctp*, *DualBuffer*, *LinkEstimator*, *CtpRoutingEngine*, and *CtpForwardingEngine*. The latter three modules clearly provide the same functionalities of the correspondent components described in section 2.1. The module *Ctp* provides marshaller functionalities by managing incoming and outgoing messages between the *CtpNoe* compound module and its internal components. Thus, only the *Ctp* module is connected to the input and output gates of the *CtpNoe* compound module. The *LinkEstimator*, *CtpRoutingEngine*, and *CtpForwardingEngine* must therefore interact with the *CtpNoe* module to dispatch their messages to other (internal or external) modules. The dotted lines in figure 2 represent direct method calls.[2] Indeed, the internal modules of a compound module may use a common set of functions. These function can be implemented in only one of the modules but may be used by the others through the above mentioned direct calls. The role of the *DualBuffer* module will be clarified in the following section 3.5.

[2]Please refer to the OMNeT^{++} user manual for a formal definition of *module*, *compound module*, *direct method call* and their usage [23].

Figure 2 also shows that our *CtpNoe* module interacts with the application and physical layers through the *Application* and *MAC* modules, respectively. In particular, the module *CtpNoe* implements the same connections to both the *Application* and the *MAC* modules as Castalia's default routing module. Embedding our implementation of CTP within the standard distribution of Castalia is thus straightforward, since it can transparently replace the default routing module. However, CTP poses some constraints on the underlying MAC layer and we thus needed to implement a CTP-compliant MAC module, as detailed in section 4.

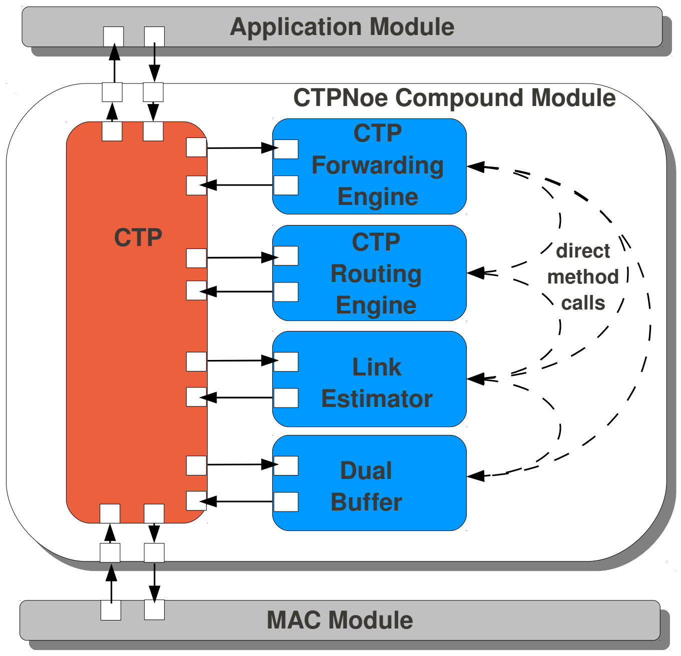

Figure 2: Architecture of the CtpNoe compound module.

The values of the relevant parameters of our Castalia-based implementation of CTP are set as default values in the network definition (NED) files of the correspondent modules. We would like to point out that our Castalia-based implementation of CTP has been developed for the versions 4.1 and 3.0 of OMNeT++ and Castalia, respectively.

In the remainder of this section we will describe our implementation of the *LinkEstimator*, *CtpRoutingEngine*, *CtpForwardingEngine*, and *DualBuffer* modules. We will particularly focus on the most tricky implementation issues and thus assume the reader has some familiarity with the topic at hand. Before going into further details, however, we first describe the structure of both the routing and data messages handled by CTP. Within each of the following subsections we organized the content in paragraphs so as to improve the readability of the text.

3.1 CTP routing and data packets

As mentioned in section 2.1, CTP relies on the exchange of *routing* messages for tree construction and maintenance and on *data* messages to report the application payload to the sink. In the TinyOS 2.1 implementation, routing and data packets are structured as schematically reported in figure 3. Clearly, we defined the same structure for the packets used by our

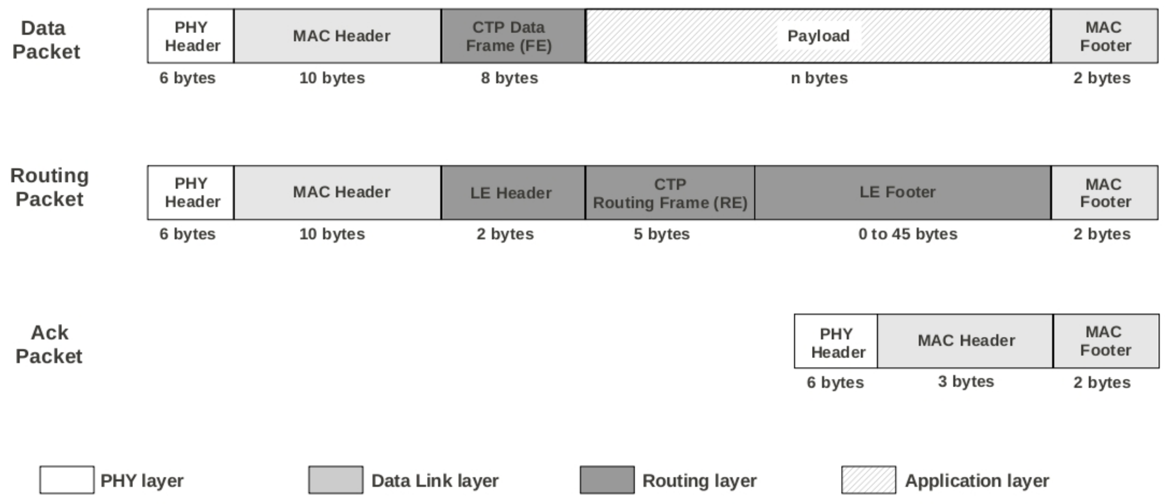

Figure 3: Structure of CTP's routing, data, and acknowledgments packets.

Castalia-based implementation of CTP.

PHY and MAC overhead. Figure 3 shows that both routing and data packet carry a total of 18 bytes of information added by the physical (PHY) and data link layer (MAC). These include the 6 bytes of the PHY header, the 10 bytes of the MAC header and the 2 bytes of the MAC footer, which we will describe in more detail in section 4. These 18 bytes constitute a fixed overhead that is attached to any routing or data packet that is sent through the wireless transceiver. Within Castalia, the MAC modules take care of setting the values of these bytes and adding them to transiting packets. In TinyOS 2.1 the same role is taken over by the controller of the radio.

CTP data frame. Before being passed to the radio data packets are handled by the Forwarding Engine, which schedules their transmission at the routing layer. The FE adds 8 bytes of control information to transiting data packets, namely the CTP data frame shown in figure 3. Figure 4b reports the structure of this 8 bytes long frame added by the FE. The first two bits of the first byte include the *P* (Pull) and *C* (Congestion) flags. The former is used to trigger the sending of beacon frames from neighbors for topology update, while the latter allows a node to signal that it is congested. As we will detail in section 3.4, if a node receives a routing beacon from a neighbor with the C flag set, it will stop sending packets to this neighbor and look for alternatives routes, so as to release the congested node. The last 6 bits of the first byte are reserved for future use (e.g., additional flags). The second byte reports the *THL* (Time Has Lived) metric. The THL is a counter that is incremented by one at each packet forwarding and thus indicates the number of hops a packet has effectively traveled before reaching the current node. The third and fourth bytes are reserved for the (multi-hop) ETX metric, which we introduced in section 2, while the fifth and sixth constitute the *Origin* field, which includes the identifier of the node that originally sent the packet. The originating node also sets the 1-byte long *SeqNo* field, which specifies the sequence number of the packet. Further, the *CollectId* is an identifier specifying which instance of a collection service is in-

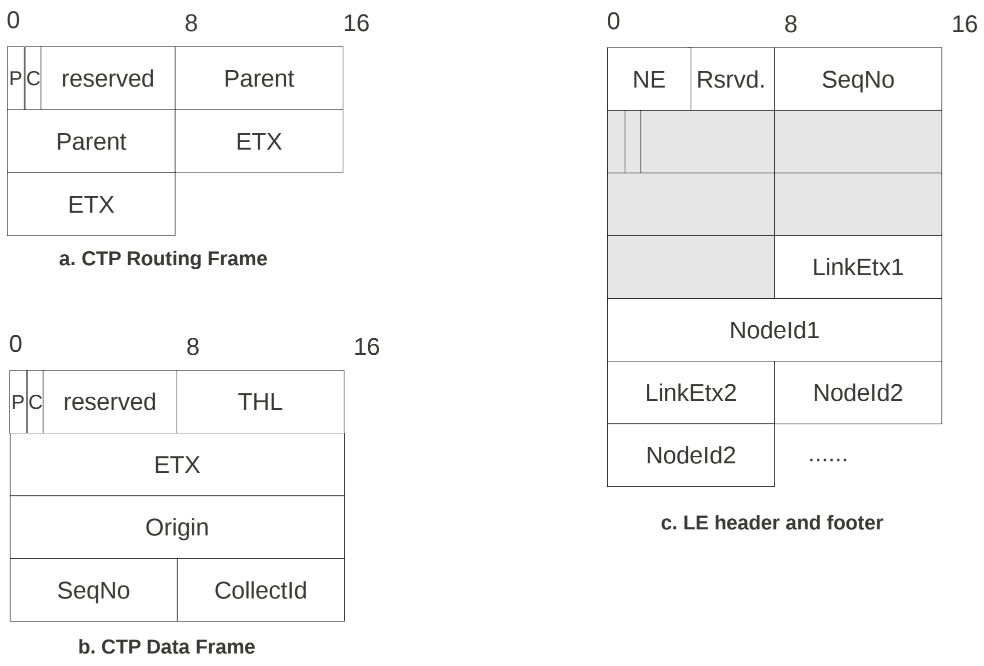

Figure 4: Structure of CTP's routing (a) and data (b) frames, along with the header and footer added by the LE (c).

tended to handle the packet. Indeed, in TinyOS 2.1 CTP can manage multiple application level components. Each of these components is assigned a unique CollectId identifier, which allows differentiating one application flow from the other [11]. Figure 3 finally shows that data packets carry a payload of n bytes, whereby the actual length n of the payload clearly depends on the application logic.

CTP routing frame. As mentioned above, CTP relies on information shared by neighboring nodes through the sending of routing packets in order to build and maintain the routing tree. While data packets are handled by the FE only, routing packets are generated and processed by both the RE and LE. As shown in figure 3 routing packets carry, in addition to the PHY and MAC overhead, a 2 bytes header and a variable length footer which are set by the LE and will be described in the following section 3.2. The actual CTP routing frame is 5 bytes long and its fine-grained structure is reported in figure 4a. As for data frames, the first byte of a CTP routing frame includes the P and C flags and 6 unused bits. The second and third bytes host the *Parent* field, which specifies the identifier of the parent of the node sending the beacon. The fourth and fifth byte finally include the (multi-hop) ETX metric. Please note that while the multi-hop path ETX is stored over 2 bytes, the 1-hop ETX only needs 1 byte.

Acknowledgments. For the sake of completeness figure 3 also shows the structure of acknowledgment (Ack) packets, which are used to notify the successful reception of a data packet to the sender of the packet. Since CTP makes use of link-layer acknowledgments, Ack packets only include PHY and MAC layer information for a total of 11 bytes.

3.2 *LinkEstimator* module

As discussed in section 2.1 and reported in [13], the Link Estimator is mainly responsible for determining the quality of the communication link. In TinyOS 2.1, the LE is implemented by the `LinkEstimatorP.nc` component, located in in the folder `/tos/lib/net/4bitle`.[3] In our Castalia-based implementation, the function of the LE is clearly implemented by the *LinkEstimator* module, to which we will simply refer to as the LE (or the LE module), for simplicity.

LE header and footer. Figure 4c shows the structure of the header and footer added by the LE to routing packets before they are passed over to the radio for transmission. The first byte of the header, namely the NE (Number of Entries) field, encodes the length of the footer. This value is actually encoded in 4 bits only while the remaining 4 are reserved for future use. The second byte includes the *SeqNo* field, which represents a sequence number that is incremented by one at every beacon transmission. By counting the number of beacons actually received from each neighbor and comparing this number with the corresponding *SeqNo*, the LE can estimate the number of missing beacons over the total number of beacons sent by a specific neighbor.

While the header has a fixed length of 2 bytes, the footer has a variable length which is upper bounded by the residual space available on the beacon. The footer carries a variable number of *<etx,address>* couples, each of 3 bytes in length, including the 1-hop ETX (1 byte) and the address (2 bytes) of neighboring nodes. Please recall that the 1-hop ETX requires only 1 byte to be stored while the multi-hop ETX requires 2 bytes. As mentioned in section 2.1, the entries included in LE's footer are selected following a round robin procedure over the link estimator table. Since the maximal length of the footer is of 45 bytes, the total size of a routing packet from 25 to 70 bytes, as also shown in figure 3.

Although the use of the footer is foreseen in CTP's original design [13, 14], the TinyOS 2.1 implementation of CTP does not make use of it and so neither does our Castalia-based implementation.[4]

Computation of the 1-hop ETX. For each neighbor, the LE determines the 1-hop ETX considering the quality of both the ingoing and outgoing links. The quality of the outgoing link is computed as follows. Let n_u be the number of unicast packets sent by the node (including retransmissions) and n_a the corresponding number of received acknowledgments. The quality of the outgoing link is then simply computed as the ratio:

$$Q_u = \frac{n_u}{n_a}. \tag{1}$$

If $n_a = 0$ then Q_u is set equal to "*the number of failed deliveries since the last successful delivery*" [13]. Following the link estimation method proposed in [30], the LE computes the first value of Q_u after a number w_u of unicast packets has been sent. The values of n_u and n_a are then reset and, after w_u transmissions, a new value of Q_u is computed. This windowing method basically allows to "sample" the value of Q_u at a frequency specified by w_u. We

[3]For the sake of simplicity, we assume that for all the TinyOS paths listed in this report the root / refers to the root of the TinyOS 2.1 distribution (e.g., *tinyos-2.x/*).

[4]Another implementation of the LE, described in revision 1.8 of TEP 123 and available in `/tos/lib/net/le` does use the footer.

should recall at this point that the value of Q_u is computed for the link to a specific neighbor. Therefore, the values of n_u and n_a clearly refer only to the packets and acknowledgment exchanged with that specific neighbor. Furthermore, we would like to outline that in order to count the number of successfully acknowledged packets, the LE must retrieve (from the link layer) the information stored on the *Ack* bit of a packet. More specifically, in our Castalia-based implementation the MAC module pushes this information to the *CtpForwardingEngine* module immediately upon reception of an acknowledgment, as described in section 4. The *CtpForwardingEngine* module, in turn, signals this reception to the LE.

As soon as a new value of Q_u is available, it is passed to the function that computes the overall 1-hop ETX. Beside the quality of the outgoing link, however, this function takes into account also the quality of the ingoing one. If n_b is the number of beacons received by a node from a specific neighbor and N_b is the total number of beacons broadcasted by the same neighbor, then the quality of the corresponding (ingoing) link is given by the ratio:

$$Q_b = \frac{n_b}{N_b}. \tag{2}$$

As for Q_u, the values of Q_b are computed over a window of length w_b. This means that every w_b receptions of a beacon from a given neighbor a new value of Q_b is computed. Before being forwarded to the function that eventually determines the 1-hop ETX, however, the value of Q_b is passed through an exponential smoothing filter. This filter averages the new value of Q_b and that of previous samples but weighting the latter according to an exponentially decaying function. In mathematical notation, the kth sample of Q_b, indicated as $Q_b[k]$, is computed as:

$$Q_b[k] = \alpha_b \frac{n_b}{N_b} + (1 - \alpha_b) Q_b[k-1]. \tag{3}$$

In the equation above, α_b is a smoothing constant that can take values between 0 and 1, and the values of n_b and N_b are computed over a window of length w_b, as explained above. For the TinyOS 2.1 implementation of CTP, the values of α_b, w_b, and w_u are specified in the file `/tos/lib/4bitle/LinkEstimatorP.nc` as the ALPHA, BLQ_PKT_WINDOW, and DLQ_PKT_WINDOW constants, respectively. Our Castalia-based implementation uses the same variable names and values, which are reported, for the sake of completeness, in table 1. To avoid the use of floating point operations the computation of the 1-hop ETX is done using integer values only. Consequently, a scaling of the involved variables is necessary in order to avoid a significant loss of precision due to truncation. This is why the value of the constant ALPHA in `/tos/lib/4bitle/LinkEstimatorP.nc` is set to 9 instead of to 0.9. Clearly, also the other quantities involved in the computation must be accordingly scaled. Eventually, this causes the value of both the 1-hop and multi-hop ETX to actually represent the tenfold of the expected number of transmissions. The parent selection procedure requires finding the neighbor with the lowest ETX, irrespectively of the absolute values, and it is thus not affected by the above mentioned scaling. For further details on this issue we refer the reader to the well-documented `/tos/lib/4bitle/LinkEstimatorP.nc` file.

Each time a new value of Q_u or Q_b for a specific neighbor is available, the 1-hop ETX metric relative to the same neighbor is accordingly updated. As for Q_b, the update procedure uses an exponential smoothing filter. Let Q be the new value of either Q_u or Q_b and ETX_{1hop}^{old} the previously computed value of the 1-hop ETX. The updated value of the ETX_{1hop} is then computed as follows:

$$ETX_{1hop} = \alpha_{ETX} Q + (1 - \alpha_{ETX}) ETX_{1hop}^{old}. \tag{4}$$

Link Estimator		
Parameter	**Value**	**Unit**
α_{ETX}	0.9	-
α_b	0.9	-
w_b	3	packets
w_u	5	packets
Size of link estimator table	10	entries
Header length	2	bytes
Footer length	< 45	bytes

Table 1: Values of relevant parameters of the Link Estimator module.

In the standard implementation of the LE the smoothing constant α_{ETX} is set to 0.9. As mentioned above, however, the value of α_{ETX} used within the `/tos/lib/4bitle/LinkEstimatorP.nc` file is the tenfold of the "theoretical" one, thus 9 instead of 0.9.

After each update, the value of the 1-hop ETX is stored in the link estimator table along with the corresponding neighbor identifier.

Insertion/eviction procedure of the link estimator table. When a beacon from a neighbor that is not (yet) included in the link estimator table is received, the LE first checks whether there is a free slot in the table to allocate the newly discovered neighbor. If the link estimator table is not filled, the LE simply inserts the new entry in one of the free available slots. In the TinyOS 2.1 implementation of CTP the maximal number of neighbors that can be included in the link estimator table is 10. A different value can be used by accordingly setting the variable NEIGHBOR_TABLE_SIZE in the file `/tos/lib/net/4bitle/LinkEstimator.h`.

If the link estimator table is filled but it includes at least one entry that is not *valid*, then the LE replaces the first found non valid entry with the new neighbor. An entry of the link estimator table becomes *valid* as soon as it is included in the table and turns not valid if it is not updated within a fixed timeout. For instance, the entry corresponding to a neighbor that (even if only temporarily) loses connectivity may become not valid. Beside the valid flag the LE also sets, for each entry of the link estimator table, a *mature* and *pinned* flag. The former is set to 1 when the first estimation of the 1-hop ETX of a new entry of the neighborhood table becomes available. This happens after at least one value of Q_b or Q_u can be computed. Instead, the pinned flag, or *pin* bit, is set to 1 if the multi-hop ETX of a node is 0, and thus the node is the root of a routing tree, or if a neighbor is the currently selected parent. The pin bit is set at the routing layer and its value is propagated to the LE and its link estimator table.

If the link estimator table is filled and all of its entries are valid, then the LE verifies if there exist (valid, mature, and non pinned) entries whose 1-hop ETX is higher than a given threshold. This threshold is set to a high value[5] and it thus allows individuating unreliable communication partners. Among these nodes, the one with the worst (i.e., highest) 1-hop ETX is evicted from the table and the new neighbor is inserted in place of it. The value of the 1-hop ETX of the new neighbor is irrelevant at this point since it is not yet available. Indeed,

[5] 55 in the TinyOS 2.1 implementation of CTP. See also the EVICT_EETX_THRESHOLD constant in `/tos/lib/net/4bitle/LinkEstimatorP.nc`.

the computation of the 1-hop ETX can only start after a node has been inserted in the link estimator table and the values of Q_u and Q_b start being computed.

If none of the entries of the link estimator table is eligible for eviction as described above, then the LE determines whether to insert the new neighbor anyway or discard it. The LE forces an insertion of the new neighbor in two cases. First, if it is the root of a routing tree and thus the multi-hop ETX declared in the received beacon is set to 0. Second, if the multi-hop ETX of the new neighbor is lower (i.e., better) than at least one of the (valid, mature, and non pinned) entries of the routing table. This latter step clearly requires contacting the Routing Engine in order to retrieve the information related to the multi-hop ETX. To this end, the LE requests the RE to return the value of the so-called *compare* bit. If the compare bit is set to 1, then the new neighbor must be inserted, if it set to 0 the new neighbor is discarded.

If the LE determines the new neighbor should be inserted, then it randomly chooses one the existing (non pinned) entries and replaced it with the new neighbor. The flow chart corresponding to the above described eviction/insertion procedure is reported in figure 5.

The white bit. In the original design of CTP [13] not all beacons received from neighbors that are not included in the link estimator table are considered for insertion. In particular, upon reception of such a beacon the LE first checks if the corresponding probability of decoding error (averaged over all symbols) is higher than a given quality threshold. Better said, the LE requires the physical layer to return the value of 1 bit of information, the so-called *white* bit. If the bit is set to 1, then the packet is considered for insertion otherwise it is immediately discarded. The white bit thus "*provides a fast and inexpensive way to avoid borderline or marginal links*" [13]. In the TinyOS 2.1 implementation of the LE, however, this information is not considered during the neighbor eviction process and so neither does our Castalia-based implementation. We refer to the `CompareBit.shouldInsert` function in `/tos/lib/net/ctp/RoutingEngine.nc` for further details on this issue.

4-bits Link Estimator. As the above description makes clear the LE has two main tasks: populating the link estimator table with the "best possible" neighbors and computing their 1-hop ETX. To comply with both tasks, the LE retrieves information from the MAC layer as well as the FE and RE. In particular, the LE needs to retrieve the values of the ack, white, pin, and compare bits, which have been described above. Since it makes use of these 4 state bits, the LE is also called 4-*bit LE*.

Differences between actual implementation and original design. The actual default implementation of the Link Estimator in TinyOS 2.1 can be found in `/tos/lib/net/4bitle` and is described in revision 1.15 of TEP 123 [11]. As noted above, this implementation (and thus also our Castalia-based one) differs from CTP's original design in two points. First, the white bit is not considered during the LE's table insertion/eviction procedure. Second, the entries included in the footer of routing beacons are not used for populating the link estimator and routing tables.

3.3 *CtpRoutingEngine* module

In CTP, the main tasks of the Routing Engine consist in sending beacons, filling the routing table, keeping it up to date, and selecting a parent in the routing tree towards which data frames must be routed. We recall at this point that CTP's routing frames are 5 bytes long,

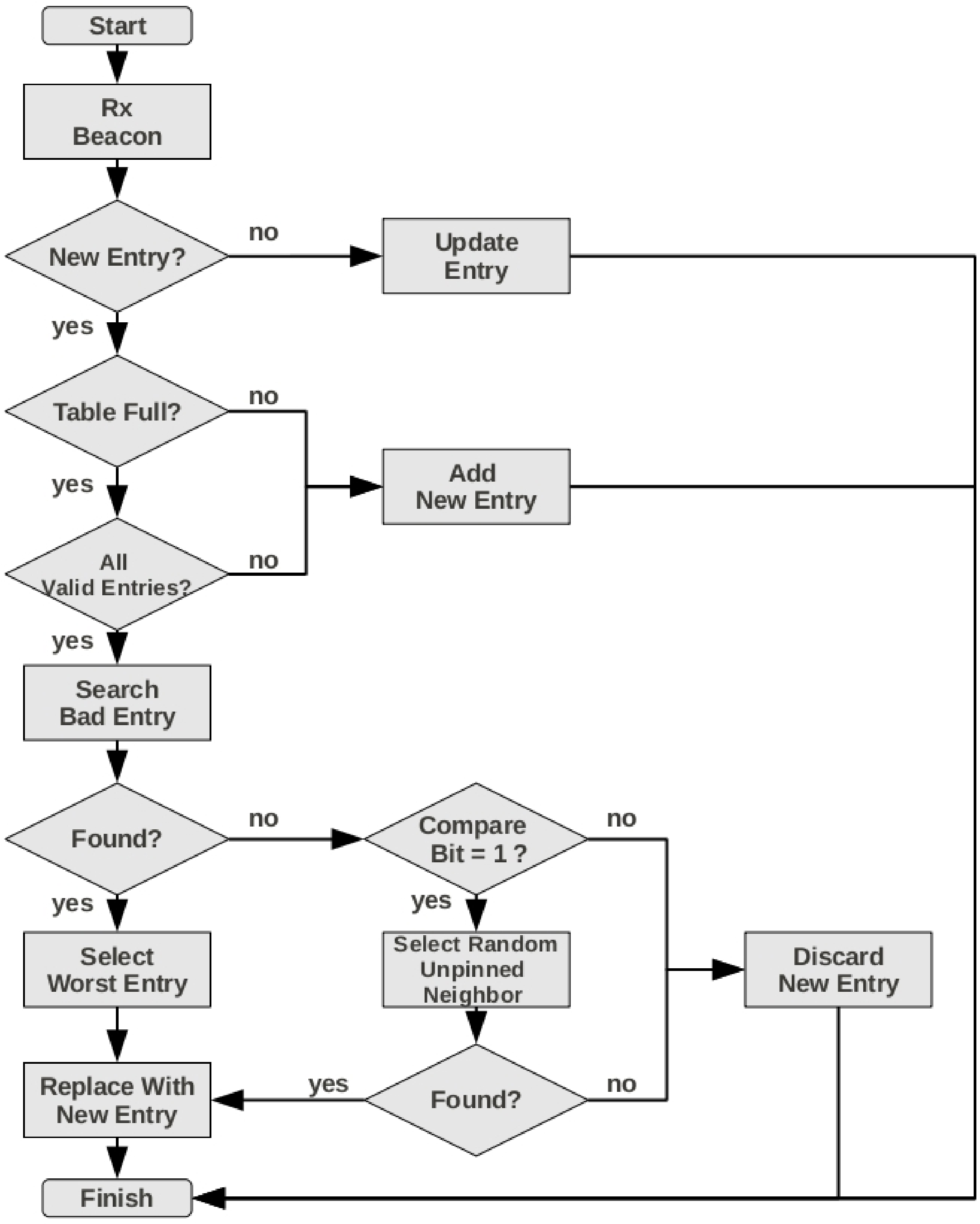

Figure 5: Insertion policy for LE's link estimator table.

as shown in figure 4a. To derive the total size of a beacon, however, the overhead (headers and footers) added by the LE and the physical and link layers must also be considered, as we discussed in section 3.1.

Frequency of beacon sending. The frequency at which beacons are sent is controlled according to the Trickle algorithm [20]. Trickle chooses the time instant for sending a beacon as a random value within the interval $[I_b/2, I_b]$. The value of I_b is doubled after each transmission so that the frequency at which beacons are sent is progressively reduced. The minimal value of I_b, called I_b^{min}, is set a priori. Also, in order to avoid a too long absence of beacon transmissions, a maximal value for I_b, called I_b^{max}, is fixed a priori. Both the values of I_b^{min} and I_b^{max} must be specified when the *CtpRoutingEngine* module is initialized[6]. As usual, our implementation of the RE uses the same values of its TinyOS 2.1 counterpart, as also summarized in table 2.

The occurrence of specific events can cause the value of I_b to be reset (through a call of the `resetInterval` command in the TinyOS 2.1 implementation). These events include the detection of a routing loop and the reception of a packet with the pull (P) flag set to 1. As we will detail below, the reception of a data frame whose multi-hop ETX is lower than that of the receiver may signal the existence of a routing loop. If this is the case, the FE triggers a topology update (through the `triggerRouteUpdate` interface in the TinyOS implementation), which is in turn implemented in the RE. When a node needs a topology update (e.g., since it has no route to the sink), it sets the pull flag of its outgoing packets to 1. Nodes receiving a beacon or data frame with the pull flag set reset their beacon sending interval.

Routing table updates. The RE further takes care of updating the routing table by retrieving information about available neighbors, their selected parent, congestion status and multi-hop ETX. The routing table is updated asynchronously upon reception of routing frames and, in the TinyOS 2.1 implementation, it contains up to 10 entries (see also the TREE_ROUTING_TABLE_SIZE constant in `/tos/lib/ctp/CtpP.nc`). The eviction procedure described in the previous section allows keeping only the most reliable neighbors in both the link estimator and the routing tables.

Parent selection procedure. The parent selection procedure is repeated periodically[7] or called asynchronously when one of the following events occurs: a beacon is sent; a neighbor is unreachable (thus its entry in the link estimator table is not valid); a neighbor is no longer congested; the currently selected parent gets congested; the node has no route to the sink.[8]

Among those included in the routing table only neighbors having a valid path to the sink, that are not congested and are not children of the current node are eligible to be selected as the parent node. Among eligible neighbors, the node with the lowest multi-hop ETX is selected as the parent node. A new parent is selected if one of the two following conditions is fulfilled.

First, if the current parent is congested and there exists a neighbor whose multi-hop ETX is strictly lower than the multi-hop ETX of the current parent plus a threshold value ETX_{switch}^{con},

[6]In the TinyOS 2.1 implementation of CTP, this is done at line 134 of the file `/tos/lib/net/ctp/CtpP.nc`

[7]In the TinyOS 2.1 implementation of CTP, the length of this period is set to 8 seconds (see also the BEACON_INTERVAL parameter in `/tos/lib/net/ctp/TreeRouting.h`).

[8]For detailed information about how CTP actually handles congestions please refer also to section 3.4.

Routing Engine		
Parameter	**Value**	**Unit**
I_b^{min}	125	ms
I_b^{max}	500	s
Size of the routing table	10	entries
Parent switch threshold	15	-
Parent refresh period	8	s
Beacon Packet size	5	bytes

Table 2: Values of relevant parameters of the Routing Engine module.

then this neighbor is selected as the new parent. The threshold ETX_{switch}^{con} is necessary to avoid selecting as the new parent a neighbor which is actually a child of the current node. By definition, a child is 1-hop away from the parent and thus has a multi-hop ETX which is at least the parent's multi-hop ETX + 1. The "natural" value for the threshold ETX_{switch}^{con} is therefore 1. However, since the the ETX actually represents ten times the expected number of transmissions, as explained in 3.2, the ETX_{switch}^{con} threshold is equal to 10.

Second, if the current parent is not congested, a parent switch may still take place. To this end, the multi-hop ETX of a neighbor increased by a threshold ETX_{switch}^{nocon} must be strictly lower than the multi-hop ETX of the current parent. If this condition is fulfilled, then the corresponding neighbor is selected as the new parent. In the TinyOS 2.1 implementation of CTP the value of ETX_{switch}^{nocon} is 1.5. However, due to the usual scaling, the value of the PARENT_SWITCH_THRESHOLD constant in `/tos/lib/net/ctp/TreeRouting.h` is set to 15.

3.4 *CtpForwardingEngine* module

In CTP, the main task of the Forwarding Engine consist in forwarding data packets received from neighboring nodes as well as sending packets generated by the application module of the node. Additionally, the FE is responsible for recognizing the occurrence of duplicate packets and routing loops. Last but not least, the FE also works as a snooper and listens to data packets addressed to other nodes in order to timely detect a topology update request or a congestion status. In our Castalia-based implementation of CTP, the *CtpForwardingEngine* module takes over the tasks of the FE.

Packet queue and retransmissions. The FE stores the data packets to send in a FIFO queue whose length is set to 12 in the TinyOS 2.1 implementation (see the FORWARD_COUNT constant in `/tos/lib/net/ctp/CtpP.nc`). This queue, however, also hosts packets coming from the application layer of the node. In the TinyOS 2.1 implementation, the length of the queue is increased by one for every application-level client. In our Catsalia-based implementation we assume the presence of a single client and we thus set the length of the queue to 13. To forward a packet, the FE first retrieves the identifier of the current parent from the RE. If the parent is not congested the FE calls a *send* procedure thereby appending to the packet the 8 bytes long CTP data frame shown in figure 4b. Otherwise, if the parent is congested, it waits until the congestion status of the parent changes or a new parent is selected.

After sending a packet the FE awaits for a corresponding acknowledgment before removing it from the head of the queue. If the acknowledgment is not received within a given time interval, the FE will try and retransmit it until a pre-specified maximum of retransmission attempts have been performed. After the maximal number of retransmission attempts have been reached, the packet is discarded. In the TinyOS 2.1 implementation the maximal number of retransmissions is set to 30 (see also the MAX_RETRIES parameter in `/tos/lib/net/ctp/CtpForwardingEngine.h`), and so it is in our Castalia-based version.

Congestion flag. If a node is chosen as parent from several neighbors, or if it must perform many retransmissions attempts, the queue of its FE may quickly fill up with unsent packets. As this may eventually cause the node to drop further incoming packets, the FE notifies this congestion status by setting the C flag of outgoing data frames to 1. In particular, the FE declares the node as congested as soon as half of its packet queue is full. Additionally, the FE also notifies the congestion status to the RE, which takes care of setting also the C flag of routing frames to 1. This simple mechanism allows the protocol to (re-)distribute the communication load over of the network since a congested node is unlikely to be selected as parent (see also section 3.3).

Optionally, the FE can force the RE to reset the beacon sending interval as soon as a congestion state is detected (by calling the command `setClientCongested` in the TinyOS implementation). This option, however, is not used in the TinyOS implementation of CTP, so nor does our Castalia-based one. Nonetheless, CTP can react quickly when congestions occur. Indeed, when a node gets congested its packet queue is at least half full, and, thus, it is likely that a data packet will be sent soon after the congestion flag is set. This flag is carried by both data and routing packets and as nearby nodes snoop on data traffic (see also the corresponding paragraph below), they will be timely informed about the congestion status of their neighbor. In this way, data packets take care of carrying the notification of congestion and make it possible to avoid resetting the beaconing interval.

Congestions are signalled by the FE as described above. On the other side, both the FE and the RE can detect congestions by reading the correspondent flag of received unicast packets or routing beacons. In the TinyOS implementation of CTP, however, both the FE and the RE may be inhibited from reporting this information in the routing table. This can be achieved by setting to *true* the value of the variable *ECNOff* in TinyOS's `CtpRoutingEngineP.nc` file. Indeed, this is CTP's default behavior and it implies that nodes do not delay their transmissions nor immediately change their parent even when packets from their current parent have the C flag set to 1. Similarly, a call to the method `isNeighborCongested`, also implemented in the file `CtpRoutingEngineP.nc`, will immediately return false when *ECNOff* is set to true.

Duplicate packets. In order to detect duplicate packets, the FE evaluates the tuple <Origin, CollectId, SeqNo, THL> for each incoming data packet. As described in section 2.1, the Origin parameter specifies the identifier of the node which originally sent the packet; SeqNo is the sequence number of the current frame; and the THL is the hop count of the packet. The CollectId field identifies a specific instance of CTP. If only a single instance of CTP is active, the CollectId field is not relevant and can be ignored during duplicate detection. Thus, the tuple <Origin, CollectId, SeqNo, THL> represents a unique packet instance. By comparing the value of the tuple of an incoming packet with that of the packets stored in the forwarding queue, the FE can detect duplicates. The FE also maintains an additional cache to store the last 4 successfully transmitted packets. Using this cache, duplicates detection can be per-

Forwarding Engine		
Parameter	**Value**	**Unit**
Forwarding queue size	13	packets
LRU Cache size	4	packets
Max tx retries	30	-
TX_OK backoff	15.6 - 30.3	ms
TX_NOACK backoff	15.6 - 30.3	ms
CONGESTION backoff	15.6 - 30.3	ms
LOOP backoff	62.5 - 124	ms
Header size	8	bytes

Table 3: Values of relevant parameters of the Forwarding Engine module.

formed also over recently transmitted packets that are no more available in the forwarding queue.

Routing loops. The FE also features a mechanism to detect the occurrence of routing loops. To this end, the FE compares the (multi-hop) ETX of each incoming packet with the (multi-hop) ETX of the current node. In particular, since the ETX is an additive metric over the whole routing path and the current node is selected as a parent by the sender of the packet, then the ETX of the sender must be strictly higher than the ETX of the receiver (i.e., the current node). If this is not the case, the FE executes a loop management procedure that first of all starts the so-called LOOP backoff timer, resets the beacon sending interval and sets the pull flag to 1 in order to force a topology update. The FE does not forward any packets until the expiration of the LOOP timer so that the radio is used to propagate routing beacons and repair the routing loop. The LOOP timer is thus usually significantly higher than the beacon sending interval. It is important to note that if a data packet is forwarded during the execution of the loop management procedure, then the packet continues being forwarded over the loop until a new path to the sink is established. Since the THL field is increase at each retransmission, the packet cannot be erroneously recognized as a duplicate.

FE's snooping mechanism. A further interesting feature of the FE consists in its ability to overhear unicast data packets addressed to other nodes. This behavior, referred to as *snooping*, allows CTP to quickly react to congestion notifications or topology update requests carried by the C and P flags of the snooped data frame. In TinyOS 2.1, the `Snoop` interface used by the FE is implemented within the `CC2420ActiveMessageP` component, which is located in the `/tos/chips/cc2420/` folder. In our Castalia-based implementation, we assume that the information about the actual intended receiver of a packet is made available by the link-layer. In particular, we implement this functionality in a CTP-compliant MAC module, as detailed in section 4.

Backoff timers. Last but not least, the FE also manages the backoff timers, i.e., the timers regulating collision avoidance mechanisms. In particular, the FE sets the TX_OK, TX_NOACK, CONGESTION, and LOOP backoff timers. The first timer is started after each successful packet transmission to balance channel reservation between nodes. The second has the same function but is activated if the intended receiver of a packet fails to return the corresponding acknowledgment. The CONGESTION backoff timer is started when a congestion status of the selected parent is detected. When this timer expires, the status of the parent is evaluated again. Finally, the LOOP timer starts when a loop is detected, as described above. Table 3 lists the value of these timers used in both TinyOS 2.1 and our Castalia-based implementation of CTP.

3.5 *DualBuffer* module

In the TinyOS 2.1 implementation of CTP the RE and the FE both make use of the `AMSend` TinyOS interface for sending radio packets. More precisely, packets sent by the RE are first passed to the LE that attaches its headers and footers and then takes care of actually forwarding the packet to the radio. The LE and the FE, however, use two different instances of the `AMSend` interface. Each of these instances holds its own packet queue (with a maximum length of one packet each) and the radio component alternatively selects packets from either queues. The *DualBuffer* module takes care of reproducing in Castalia the same behavior of the TinyOS `AMSend` interface. In particular, the module intercepts packets from the LE and FE and manages two separate buffers for handling outgoing packets.

4 Writing CTP-compliant MAC modules in Castalia

As mentioned several times in the previous section, using CTP in Castalia requires defining a medium access control (MAC) module that complies with a given set of requirements. In particular, CTP relies on link-layer acknowledgments. Also, it needs to use information that is available in the components implementing the medium access control and it thus requires a way of retrieving this information from the link-layer. Furthermore, as our Castalia-based code mimics CTP's implementation for TinyOS, it is also necessary to emulate TinyOS's split-phase behavior within Castalia. When using our CTP module it is thus necessary that the underlying MAC module supports the following features:

1. Link-layer acknowledgements;
2. Propagation of link-level information;
3. Reception of packets whose destination is not the current node (packet snooping);
4. Split-phase mechanism of TinyOS.

In general, these features can be added to any MAC module within Castalia. For the sake of completeness, however, we already implemented a CTP-compliant MAC module, which is publicly available on the repository of the CTP-Castalia project (`http://code.google.com/p/ctp-castalia/`). In the following, we first briefly summarize the characteristics of this module and then discuss each of the four features listed above. Finally, we also provide some considerations on the overhead induced by the MAC layer on packets length.

CC2420Mac module. This Castalia module, called CC2420Mac, implements the MAC protocol for the ChipCon 2420 radio transceiver. Several well-known WSN prototyping platforms like the TelosB and MicaZ motes are equipped with this radio chip. The MAC protocol for the ChipCon 2420 is a CSMA/CA protocol that, upon waking up the radio, starts a timer

TunableMAC		
Parameter	**Value**	**Unit**
Initial Backoff range	0.3 - 10	ms
Congestion Backoff range	0.3 - 2.4	ms
PHY overhead	6	bytes
MAC overhead	12	bytes
Ack Packet size	11	bytes
Ack timeout	7.8	ms
Ack turnaround time	21	μs

Table 4: Values of relevant parameters of the TunableMAC module.

(the so-called *initialBackoff*) whose value is chosen randomly within a pre-specified interval. After this timer expires the module verifies if the channel is clear for at least a given time interval (whose length is determined by the so-called *ack turnaround* time). In the affirmative case, it sends the next packet scheduled for sending. If the channel is busy, a new transmission attempt is scheduled and started after a randomly chosen *congestionBackoff* timer expires.

Our Castalia-based implementation of the CC2420Mac modules closely follows the corresponding TinyOS component `/tos/chips/cc2420/transmit/CC2420TransmitP.nc`. At the same time, the CC2420Mac module complies with Castalia's design requirements. In particular, the module itself extends the *VirtualMac* module defined in the Castalia framework; data packets defined in CC2420Mac extend *MacPacket* message; and control messages accordingly extend *MacControlMessage* messages. Both *MacPacket* and *MacControlMessage* are standard Castalia message formats.

Table 4 illustrates the default values used for the main parameters of the CC2420Mac module. The values of the initial and congestion backoff timers, which we mentioned above, are set within the range $0.3-10$ ms and $0.3-2.4$ ms, respectively. In the TinyOS code, the actual values of these timers are computed by the component `/tos/chips/cc2420/csma/CC2420CsmaP.nc`, which in turn refers to the values *CC2420_ MIN_ BACKOFF* and *CC2420_ BACKOFF_ PERIOD* defined in the file `/tos/chips/cc2420/CC2420.h`. The maximal time interval a node waits for an acknowledgment before declaring the transmission attempt failed is also defined in the same file through the constant *CC2420_ ACK_ WAIT_ DELAY*. Note that in the TinyOS files these values are expressed as multiples of 32Khz clock ticks while in table 4 we reported the corresponding values in milliseconds.

Link-layer acknowledgements. When a node receives a CTP unicast packet, it is required to confirm its reception at the *link-layer*. This implies that before CTP starts processing the packet, the MAC module must send an acknowledgment message over the air. This message is a broadcast packet that contains only few bytes of information, as shown in figure 3. If the sender of the packet does not receive an acknowledgment for a previously sent packet, its MAC protocol accordingly notifies CTP through an *ack failure* message. This, in turn, schedules a retransmission attempt (unless the maximal number of allowed retransmissions has already been reached), as already discussed in section 3.4.

The maximal extension of the time interval the sender waits for such an acknowledgment is called the *ack timeout*. As indicated in table 4, the default value of this timeout is 7.8ms. In TinyOS, this value is specified by the constant *CC2420_ ACK_ WAIT_ DELAY* that is in turn

```
typedef nx_struct cc2420_header_t {
  nxle_uint8_t length;
  nxle_uint16_t fcf;
  nxle_uint8_t dsn;
  nxle_uint16_t destpan;
  nxle_uint16_t dest;
  nxle_uint16_t src;
  nxle_uint8_t type;
} cc2420_header_t;
```

Figure 6: The *cc2420_header_t* file.

defined in the file `/tos/chips/cc2420/CC2420.h`. When an acknowledgement is successfully received then the ack timeout timer is stopped and the transmission phase concluded. When computing the 1-hop ETX CTP makes use of information about the number of successfully received acknowledgments, as described in section 3.2.

Propagation of link-level information. In order to work properly CTP uses some pieces of information that are available at the link-layer and must thus somehow be propagated to the upper layers. These include the address of the node sending a packet (the source `src`) and that of its intended receiver (the destination `dest`). These values must thus be stored by the MAC module and passed to CTP, as they would otherwise be discarded together with the rest of the MAC-level header and footer as the packet moves towards upper layers. To this end, we make use of the *RoutingInteractionControl* structure that is included in Castalia's default *RoutingPacket* (see file `RoutingPacket.msg` in Castalia's `routing` folder). Although this structure should actually be used to handle information at the routing level, we "misuse" it as a carrier for link-level information.[9] Similarly, other fields like the LQI or RSSI of the received packet (which are inserted in the footer of the received packet when it traverses the physical layer) may also be used by CTP and should thus be propagated to the upper layers as described above. These fields just need to be copied from one structure to the other, as they are both included by default in the *MacInteractionControl* structure of *MacPacket* and the *RoutingInteractionControl* of *RoutingPacket*.

Packet snooping. The above described mechanism to propagate link-level information to CTP also enables a straightforward implementation of a snooping mechanism. When snooping is not required, unicast packets having an intended destination that does not coincide with the nodes actually receiving it are discarded at the MAC level and never reach the upper layers. Instead, CTP uses these packets to improve its responsiveness upon specific events (e.g., congestion or topology update requests). Thus, the MAC layer must forward all received unicast packets to the routing layer but it also need to keep a mechanism to differentiate between packets actually intended for the nodes and packets that are just being snooped. To this end, CTP can use the destination address information that is propagated to the upper layers instead of being discarded at the link-layer, as described above.

[9]In particular, instead of calling Castalia's *decapsulatePacket* function, we explicitly set the fields of the *RoutingInteractionControl* structure when passing the packet from the MAC to the routing layer. See also the `CC2420Mac.cc` file in the CC2420Mac module.

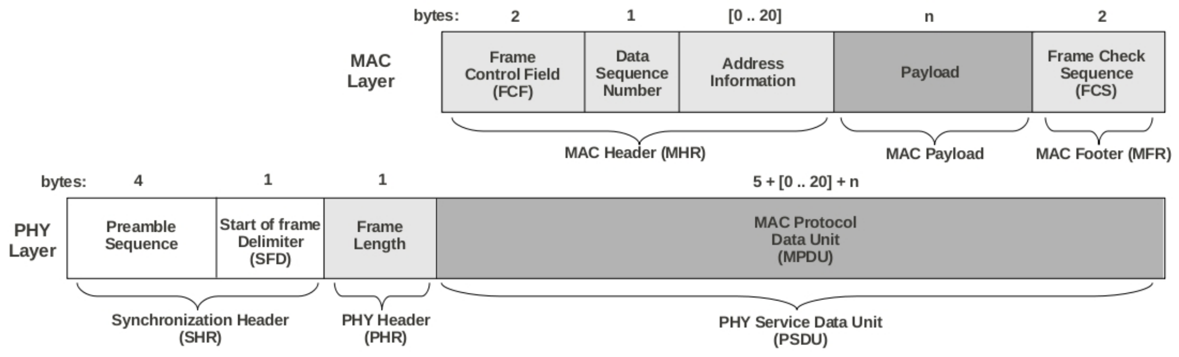

Figure 7: Schematic view of the IEEE 802.15.4 packet format [7].

Split-phase mechanism of TinyOS. The CC2420Mac module emulates the TinyOs split-phase mechanism by sending a message to the CTP module that corresponds to a TinyOS signal. To this end, we defined the *CC2420ControlMessage* that extends Castalia's *MacControlMessage* and we used it to emulate TinyOS's signalling mechanism. For instance, we emulate TinyOS's *Send.sendDone(message_ t,error_ t)* signal by generating a CC2420ControlMessage message that carries information related to the corresponding TinyOs interface (i.e., Send), the type (i.e., sendDone) and other useful information (e.g., result, acknowledgement, etc. . .).

MAC overhead. Last but not least, we would like to discuss the protocol overhead of the MAC layer. The overhead is derived from the packet format of a IEEE802.15.4 frame, which is the standard used by the CC2420 (figure 7). This figure shows the control fields added at both the physical (PHY) and data link (MAC) layers. In particular, the PHY layer adds 3 pieces of information for a total of 6 bytes: 4 for the preamble sequence, 1 for frame delimiter, and 1 for the frame length. The MAC layer, in turn, adds further 5 fields, two of which, the address information and the payload, of variable length. The actual MAC overhead is determined by analyzing the cc2420_header_t struct defined in the /tos/chips/cc2420/CC2420.h file (figure 6). This figure shows that the MAC header contains 7 fields for a total of of 11 bytes. The first field (1 byte) specifies the total length of the packet. The `fcf` and `dsn` fields occupy 3 bytes and define the values of the *Frame Control Field* (FCF) and of the *Data Sequence Number* (DSN). Further, the three field `destpan`, `dest` and `src` contains the addresses of the sender and intended receiver(s) of the packet. Finally, the field `type` represents the payload of the MAC layer and indicates the AM (Active Message) identifier of a TinyOS packet [19, Sect. 6]. Comparing figures 6 and 7 we can see that there are several differences. First of all, the length field is assigned to the PHY layer in figure 6 but appears in the MAC overhead defined by the `cc2420_header_t` struct. Second, the frame check sequence (FCS) field, which is assigned to the MAC layer in figure 7 does not appear in the `cc2420_header_t` struct. To be coherent with the IEEE 802.15.4 standard we thus removed the 1 byte of the length field from the computation of the MAC overhead and added the 2 bytes of the FCS field. We therefore consider a MAC overhead of 12 bytes, 10 for the header and 2 for the footer, as depicted in figure 3. Also according to figure 3 (and figure 7), we consider a PHY overhead of 6 bytes.

The length of packets used for acknowledging the reception of a data message must be considered separately. Indeed, acknowledgments are generated within the CC2420 controller and the corresponding packet format is described in [7, p.22]. According to this format, an acknowledgment packet has a total size of 11 bytes, 5 of which represent MAC layer overhead, while the other 6 are assigned by the physical layer, as shown in figure 3 and summarized in table 4.

5 Conclusions

This reports provides a detailed description of the implementation of the Collection Tree Protocol for the Castalia wireless sensor networks simulator. The report focuses on particularly tricky implementation issues and represents an handy reference for other researchers interested in working with CTP or re-implementing it for other platforms. Software artifacts, documentation, and updates concerning this project are available at `http://code.google.com/p/ctp-castalia/`.

References

[1] Athanassios Boulis et al. Castalia: A Simulator for Wireless Sensor Networks. http://castalia.npc.nicta.com.au/.

[2] Manohar Bathula, Mehrdad Ramezanali, Ishu Pradhan, Nilesh Patel, Joe Gotschall, and Nigamanth Sridhar. A Sensor Network System for Measuring Traffic in Short-Term Construction Work Zones. In *Proceedings of the 5th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2009)*, pages 216–230, Marina del Rey, CA, USA, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Lorenzo Bergamini, Carlo Crociani, and Andrea Vitaletti. Simulation vs Real Testbeds: A Validation of WSN Simulators. Technical Report 3, Sapienza Università di Roma, Dipartimento di Informatica e Sistemistica Antonio Ruberti, 2009.

[4] Athanassios Boulis. Castalia: Revealing Pitfalls in Designing Distributed Algorithms in WSN. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys 2007)*, pages 407–408, Sydney, Australia, 6-7 November 2007. Demonstration session.

[5] Athanassios Boulis, Ansgar Fehnker, Matthias Fruth, and Annabelle McIver. CaVi – Simulation and Model Checking forWireless Sensor Networks. In *Proceedings of the Fifth International Conference on Quantitative Evaluation of Systems (QEST 2008)*, pages 37–38, Saint Malo, France, September 14-17 2008.

[6] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN 2007)*, Cambridge, MA, USA, April 2007.

[7] ChipCon 2420 Datasheet. http://focus.ti.com/lit/ds/symlink/cc2420.pdf.

[8] Ugo Colesanti and Silvia Santini. A performance evaluation of the collection tree protocol based on its implementation for the castalia wireless sensor networks simulator. Technical Report 681, Department of Computer Science, ETH Zurich, Zurich, Switzerland, August 2010.

[9] Crossbow Technology Inc. www.xbow.com.

[10] J. E. Egea-López, A. Vales-Alonso, P. S. Martínez-Sala, J. Pavón-Mariï£¡o, and García-Haro. Simulation Tools for Wireless Sensor Networks. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*, Philadelphia, PA, USA, July 2005.

[11] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo. TinyOS Enhancement Proposal (TEP) 123: The Collection Tree Protocol (CTP). www.tinyos.net/tinyos-2.x/doc/pdf/tep123.pdf.

[12] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis. TinyOS Enhancement Proposal (TEP) 119: Collection. www.tinyos.net/tinyos-2.x/doc/pdf/tep119.pdf.

[13] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, and Philip Levis. CTP: Robust and Efficient Collection through Control and Data Plane Integration. Technical report, The Stanford Information Networks Group (SING), 2008. `http://sing.stanford.edu/pubs/sing-08-02.pdf`.

[14] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys 2009)*, Berkeley, CA, USA, November 2009.

[15] Miloš Jevtić, Nikola Zogović, and Goran Dimić. Evaluation of Wireless Sensor Network Simulators. In *Proceedings of the 17th Telecommunications Forum (TELFOR 2009)*, Belgrade, Serbia, November 2009.

[16] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley and Sons, 2005.

[17] Alireza Khadivi and Martin Hasler. Fire Detection and Localization Using Wireless Sensor Networks. *Sensor Applications Experimentation and Logistics*, 29:16–26, 2010.

[18] JeongGil Ko, Tia Gao, and Andreas Terzis. Empirical Study of a Medical Sensor Application in an Urban Emergency Department. In *Proceedings of the 4th International Conference on Body Area Networks (BodyNets 2009)*, Los Angeles, CA, USA, April 2009.

[19] Philip Levis. TinyOS Enhancement Proposal (TEP) 116: Packet Protocols. `www.tinyos.net/tinyos-2.x/doc/pdf/tep116.pdf`.

[20] Philip Levis, Neil Patel, David Culler, and Scott Shenker. A Self-Regulating Algorithm for Code Maintenance and Propagation in Wireless Sensor Networks. In *Proceedings of the 1st USENIX Conference on Networked Systems Design and Implementation (NSDI 2004)*, San Francisco, CA, USA, March 2004.

[21] Andreas Meier, Mehul Motani, Hu Siquan, and Simon Künzli. DiMo: Distributed Node Monitoring in Wireless Sensor Networks. In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2008)*, pages 117–121, Vancouver, Canada, New York, NY, USA, October 2008. ACM.

[22] Andreas Meier, Matthias Woehrle, Mischa Weise, Jan Beutel, and Lothar Thiele. NoSE: Efficient Maintenance and Initialization of Wireless Sensor Networks. In *Proceedings of the 6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON 2009*, pages 395–403, Rome, Italy, Piscataway, NJ, USA, 2009. IEEE Press.

[23] OMNeT++ User Manual (Version 3.2). www.omnetpp.org/doc/omnetpp33/manual/usman.html.

[24] Sung Park, Andreas Savvides, and Mani B. Srivastava. SensorSim: a Simulation Framework for Sensor Networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2000)*, pages 104–111, Boston, MA, USA, August 2000.

[25] Hai N. Pham, Dimosthenis Pediaditakis, and Athanassios Boulis. From Simulation to Real Deployments in WSN and Back. In *Proceedings of the IEEE International Symposium*

on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2007), pages 1–6, Helsinki, Finland, June 18-21 2007.

[26] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS 2005)*, pages 364–369, Los Angeles, CA, USA, April 2005.

[27] Thomas Schmid, Zainul Charbiwala, Roy Shea, and Mani Srivastava. Temperature Compensated Time Synchronization. *IEEE Embedded Systems Letters (ESL)*, 1(2):37–41, August 2009.

[28] Simon Tschirner, Liang Xuedong, and Wang Yi. Model-based Validation of QoS Properties of Biomedical Sensor Networks. In *Proceedings of the 8th ACM International Conference On Embedded Software*, pages 69–78, Atlanta, GA, USA, October 2008.

[29] Tijs van Dam and Koen Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the First International Conference on Embedded Networked Sensing Systems (SenSys 2003)*, pages 171 – 180, New York, NY, USA, 2003.

[30] Alec Woo, Terence Tong, and David Culler. Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks. In *Proceedings of the 1st ACM International Conference on Embedded Networked Sensor Systems (SenSys 2003)*, pages 14–27, Los Angeles, CA, USA, November 2003.