A Performance Evaluation of the Collection Tree Protocol Based on its Implementation for the Castalia Wireless Sensor Networks Simulator

Report

Author(s):
Colesanti, Ugo; Santini, Silvia

Publication date:
2010-08

Permanent link:
https://doi.org/10.3929/ethz-a-006869841

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technical Report / ETH Zurich, Department of Computer Science 681

A Performance Evaluation Of The Collection Tree Protocol Based On Its Implementation 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. 681
Department of Computer Science
ETH Zurich
August 31, 2010

Abstract

Many wireless sensor network applications rely on the availability of a collection service to route data packets towards a sink node. The service is typically accessed through well-defined interfaces so as to hide the details of its implementation. Providing for efficient network operation, however, often requires investigating the interplay between specific collection services and application-level algorithms. To enable a smooth evaluation of these mutual dependencies, we implemented a reference collection protocol, known as CTP, as a module for the Castalia wireless sensor networks simulator. Castalia is a well-known and widely used simulator but its standard distribution only provides for a basic collection module. By implementing a more advanced protocol like CTP we extend and improve the application scope of Castalia. In this report, we describe our implementation and present a study of the performance of CTP. All the software modules developed in the context of this work are available upon request from the authors.

1 Introduction

A wireless sensor network (WSN) is a collection of tiny, autonomously powered devices – commonly called sensor nodes – that are endowed with sensing, communication, and processing capabilities [14, 1]. Typical application scenarios for WSNs envision a large number of nodes being distributed at various locations over a region. Once deployed, sensor nodes can capture data about some physical quantity, like temperature, atmospheric pressure or a pollutant concentration [34, 9, 3, 35]. Sensor readings are then usually reported to a central server, also called the *sink* node, where they are further processed according to the application requirements. To report their readings to one or more data collectors, sensor nodes communicate through their integrated radio-transceivers and collaboratively build an ad-hoc, possibly multi-hop relay network.

*Corresponding author.

Within the last decade, the WSN research community proposed a plethora of algorithms and protocol aiming at guaranteeing efficient and reliable data collection. These include several power-aware medium access protocols and reliable routing schemes [37, 10, 17, 20]. In particular, the Collection Tree Protocol (CTP) provides for "*best-effort anycast datagram communication to one of the collection roots in a network*" [15, 17, 18]. CTP is widely regarded as a reference protocol for performing data collection in WSNs and its specification is provided in TinyOS[1] Enhancement Proposal 123 [15]. Gnawali *et al.* also report a throughout description and performance evaluation of CTP in realistic settings, demonstrating the ability of the protocol to reliably and efficiently report data to a central collector [17, 18]. A TinyOS implementation of CTP is available within the TinyOS 2.1 distribution and therefore directly usable for implementing WSNs applications. In particular, application-level modules can call a generic collection service which is in turn implemented through CTP.

This level of abstraction is usually highly desirable, since the actual protocol implementing the collection service can (theoretically) be changed without affecting the functioning of the related calling and called modules. However, when developing WSN applications it is often crucial to work with actual implementations of generic services, like CTP as a collection primitive, so as to investigate possible pitfalls and potential for cross-layer optimizations. This in turn often requires to resort to simulation as an investigation tool, especially as the number of nodes grows, due to the well-known burdens connected with the deployment of WSNs. Additionally, simulation results offer a benchmark towards which experimental data can then be compared.

In the context of our work, we make use of the Castalia WSN simulator. The standard Castalia distribution, however, does not yet include an implementation of CTP. We therefore implemented a corresponding CTP module, so as to have it available for our research on application-level algorithms. In this report, we provide a detailed description of our implementation of CTP for the Castalia simulator and report a corresponding performance analysis of the protocol. We believe this report to constitute a very useful reference for researchers interested in working with CTP. Furthermore, all the software artifacts developed in the context of this work are available from the authors upon request.

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 CTP's implementation for the Castalia simulator in section 3. In section 5, we will report an analysis of the performance of CTP based on a simulation study whose setup is described in section 4. Finally, 6 concludes the report.

2 Background

This section provides background information about data collection in WSNs, CTP, and the Castalia simulator. The reader familiar with these topics can easily skip this section and proceed to the description of CTP's Castalia-based implementation reported in section 3.

[1]TinyOS is a well-known operating system and programming environment for wireless sensor networks. For more information see also the TinyOS project's website: `www.tinyos.net`.

2.1 Data collection in wireless sensor networks

As stated in TinyOS TEP 119, data collection is one of the fundamental primitives for implementing WSN applications [16]. 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 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 its 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 occurrence 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 rare. The main factor hampering the performance of such protocols is the instability of wireless links. In particular, as pointed out in [17], 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. The Collection Tree Protocol (CTP) by Gnawali et al. directly addresses these problems and can reach excellent delivery performance, as we also show in section 5. CTP, which is described in detail below, became quickly popular within the WSNs research community [15, 21, 22]. Nonetheless, a CTP module supporting several WSN hardware platforms (MicaZ, Telosb/TmoteSky, TinyNode) is available for the TinyOS 2.1 distribution.

2.2 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 [15] and evaluated in [17, 18] 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 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 communicate 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. The ETX of a node is defined as the "*ETX of its parent plus the ETX of its link to its parent*" [15]. 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.

The frequency at which CTP beacons are sent is set by the *Trickle* algorithm [24]. 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 [16]. 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 expected number of transmission attempts required by the neighbor to successfully deliver a beacon. 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

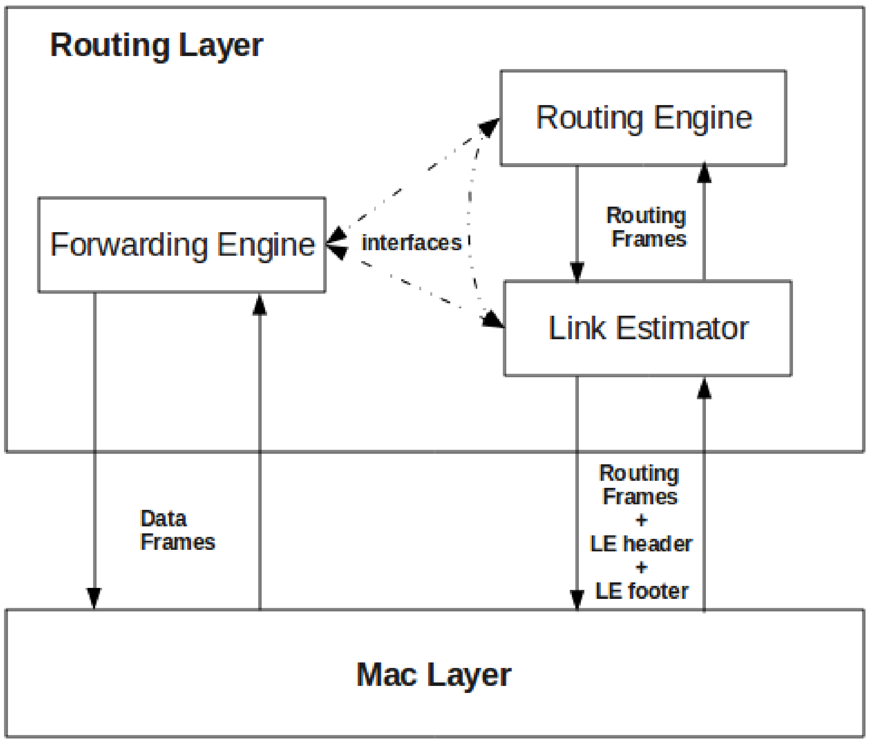


Figure 1: Message flow and modules interactions.

footer and includes it in the so-called *link estimator table* [17]. 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 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.

2.3 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 [19, 13]. 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 [26, 21, 27, 33]. Castalia provides a generic platform to perform "*first order validation of an algorithm before moving to an implementation on a specific platform*" [2]. 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 [30]). Nonetheless, in the last years Castalia has been continuously improved [7, 31] and there is an increasing number of researchers using Castalia to support their investigations [8, 36, 5, 22, 4].

In the context of this work, we make use of version 1.3 of Castalia, which builds upon version 3.3 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. In contrast to other frameworks for wireless sensor networks, Castalia offers exhaustive models for simulating both the radio channel and the physical layer of the radio module. In particular, Castalia provides bundled support for the $CC2420$ radio controller, which is the transceiver of choice for the TelosB/TmoteSky platform [12, 32]. This is particularly relevant to us since the Tmote Sky is our reference hardware platform.

3 Implementation of CTP

Being a simulator originally developed mainly for MAC protocols testing, Castalia offers only a basic hop distance routing layer in its original distribution. Thanks to the excellent modularity inherited from OMNeT++, however, Castalia can be easily extended and adapted to include new components. Hence, we have implemented a CTP routing module for Castalia that mimics, as far as possible, TinyOS 2.1 reference implementation of CTP [15, 17]. The hop distance routing protocol available in the standard Castalia distribution can be transparently replaced by our CTP module.

Figure 2 shows the basic architecture of our implementation. The three modules Link Estimator, Routing Engine, and Forwarding Engine clearly provide the same functionalities of the correspondent components described in section 2.2. The full implementation is included in a compound module, which we dubbed *CTPRouting*, and that includes also the CTPProtocol module. CTPProtocol provides marshaller functionalities by managing incoming and outgoing messages between the CTPRouting compound module and its internal components. Thus, only the CTPProtocol module is connected to the input and output gates of the CTPRouting compound module. The LE, RE, and FE must therefore interact with the CTPProtocol 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 may be implemented in only one of the modules and used by the others through the above mentioned direct calls.

Figure 2 also shows that our CTPRouting module interacts with the application and physical layers through the *Application* and *TunableMAC* modules, respectively. In particular, our CTPRouting module implements the same connections to both the Application and the TunableMAC 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, since CTP poses some constraints on the underlying MAC layer, a few changes to the TunableMAC module have been necessary in order to make our CTPRouting module work. We describe these changes in section 3.5.

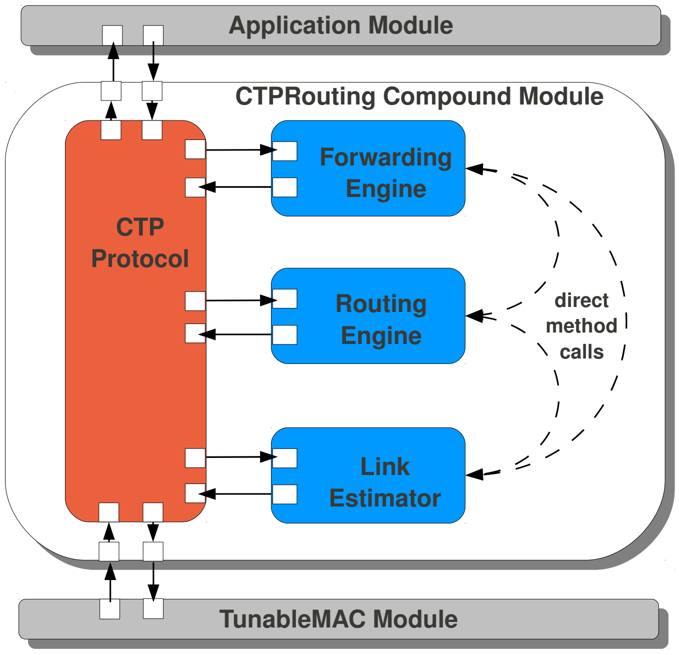


Figure 2: Architecture of the CTPRouting compound module.

The values of the relevant parameters of our Castalia-based implementation of CTP are listed in the corresponding `ctpRouting.ini` configuration file. We would like to point out that our Castalia-based implementation of CTP has been developed on top of the versions 3.3 and 1.3 of OMNeT++ and Castalia, respectively. Although these distribution are by now out-of-date[3], the considerations reported below still hold and the developed software modules would need minor adjustments only in order to run on the newer versions of the platforms.

In the remainder of this section we will describe our implementation of the LE, RE, and

[2]Please refer to the OMNeT++ user manual for a formal definition of *module*, *compound module*, *direct method call* and their usage [29].
[3]As of August 2010, versions 4.1 and 3 of OMNeT++ and Castalia are available.

FE and then detail about the changes we had to carry out on the TunableMAC module. 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.

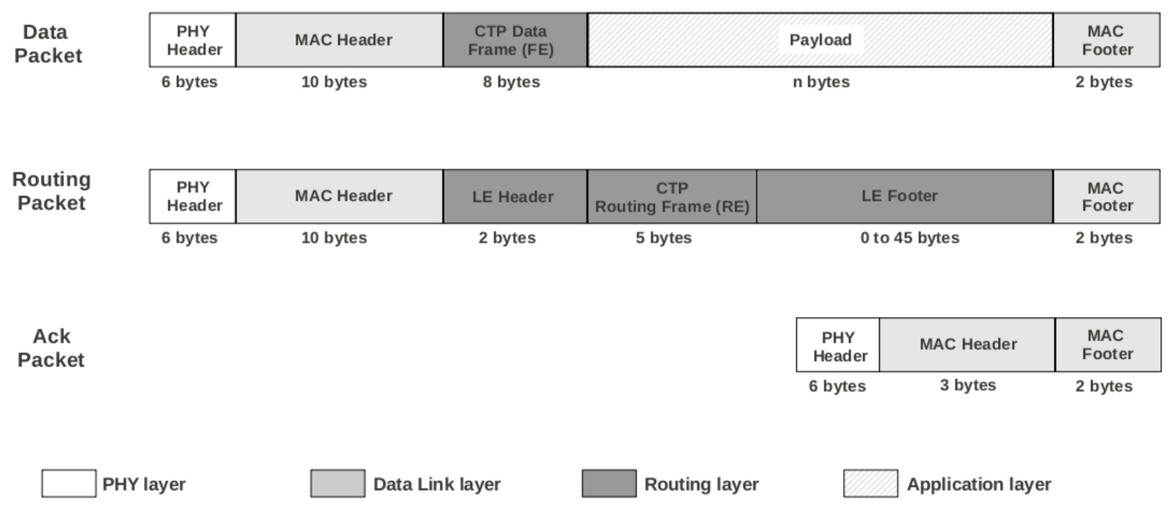


Figure 3: Structure of CTP's routing, data, and acknowledgments packets.

3.1 CTP routing and data packets

As mentioned in section 2.2, 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 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 3.5. 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 TunableMAC module takes 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 FE, 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

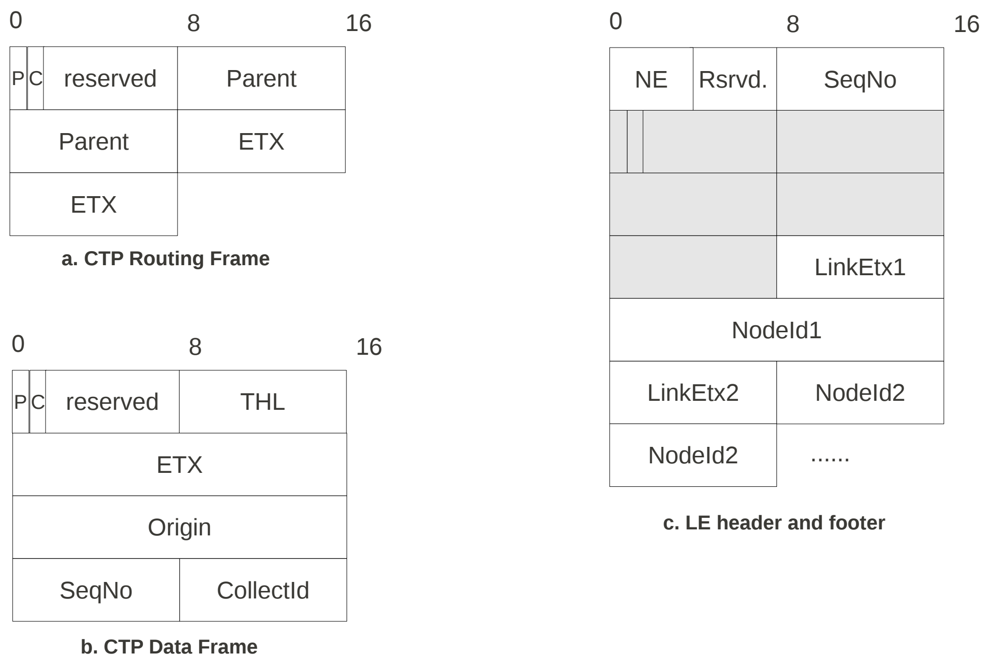


Figure 4: Structure of CTP's routing (a) and data (b) frames, along with the header and footer added by the LE (c).

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 intended 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 [15]. 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 Link Estimator module

As discussed in section 2.2 and reported in [17], 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`.[4]

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 2.2, 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 [17, 18], the TinyOS 2.1 implementation of CTP does not make use of it and so neither does our Castalia-based implementation[5].

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

[4]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/*).

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

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*" [17]. Following the link estimation method proposed in [38], 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 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 TunableMAC module pushes this information to the FE immediately upon reception of an acknowledgment. The FE, 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 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 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

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.

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:

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

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[6] 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, 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 [17] 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*" [17]. 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 TunableMAC, FE, and RE modules. 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`

[6]55 in the TinyOS 2.1 implementation of CTP. See also the EVICT_EETX_THRESHOLD constant in `/tos/lib/net/4bitle/LinkEstimatorP.nc`).

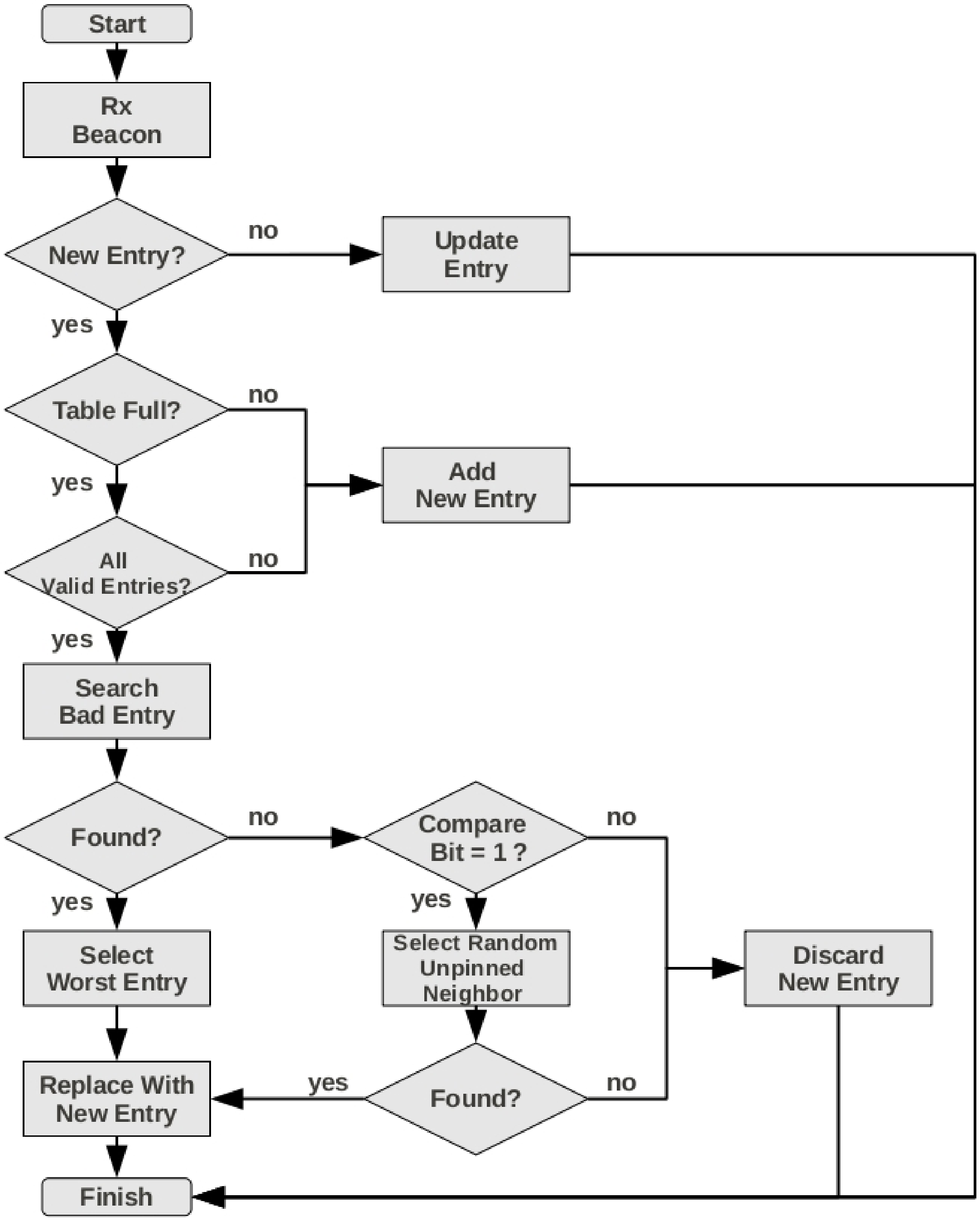


Figure 5: Insertion policy for LE's link estimator table.

and is described in revision 1.15 of TEP 123 [15]. 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 Routing Engine module

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, 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 beacons sending. The frequency at which beacons are sent is controlled according to the Trickle algorithm [24]. Using Trickle a beacon is sent at a random instant within a given time interval, whose minimal length I_b^{min} is set a priori. The length I_b of the interval is doubled after each transmission so that the frequency at which beacons are sent is progressively reduced. In order to avoid a too long absence of beacon transmissions, however, a maximal length of the sending interval, which we refer to as I_b^{max}, is fixed a priori. Both the values of I_b^{min} and I_b^{max} must be specified when the Routing Engine is initialized (in the TinyOS 2.1 implementation, this is done at line 134 of the file `/tos/lib/net/ctp/CtpP.nc`). 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; a congested node; a node 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, 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. Finally, if a node gets congested, i.e., its forwarding queue is half full, it sets the correspondent flag of its packet to 1. Again, nodes receiving beacons or data frames with the congestion flag set must reset the value of I_b. The actual TinyOS 2.1 implementation of the RE, however, does not reset the value of I_b if a congestion flag is set. Our Castalia-based implementation, on the contrary, does provide this functionality and thus the ability of quickly handling congestions. If the command `resetInterval` is called too frequently, however, the timer whose expiration triggers the sending of a beacon is continuously reset and thus never fires. This may happen if routing loops or congested nodes are often detected and has the nasty consequence of blocking beacon transmissions. To avoid such situations, our Castalia-based implementation of CTP uses a *fading* flag that inhibits a timer reset if there is a beacon sending event already scheduled and the value of the sending interval is already equal to I_b^{min}.

Routing Engine		
Parameter	**Value**	**Unit**
Minimal length of the beacon window	63	ms
Maximal length of the beacon window	250	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.

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.

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}, 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.

[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`).

3.4 Forwarding Engine module

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.

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`). 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 the *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. Since 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 call the command `setClientCongested` to force the RE to reset the beacon sending interval as soon as a congestion state is detected. Despite increasing the number of transmitted beacons, this option allows nodes to timely notify their congestion status, thus improving the reactiveness of the network. Our simulation study showed that by activating this option we often can significantly decrease the number of packets dropped due to congested nodes. For this reason the `setClientCongested` option is always enabled in our Castalia-based implementation of CTP.

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 2.2, 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

Forwarding Engine		
Parameter	**Value**	**Unit**
Forwarding queue size	12	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.

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 performed 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, the TunableMAC module notifies the FE when such snooped packets are available. To this end, we created a new `MAC_2_NETWORK_SNOOP_MSG` message within the TunableMAC module, as we detail in the following section 3.5.

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 TunableMAC module

TunableMAC is the name of the module that implements MAC functionalities in Castalia. Unfortunately, TunableMAC does not implement all the features that CTP requires to be available at the MAC layer and we thus accordingly modified it and added the required features.

RE and FE's packet queues. For instance, in the TinyOS 2.1 implementation of CTP the RE and the FE both make use of the `AMSend` TinyOS interface, which offers primitives for sending radio packets. In particular, the RE and the FE use two different instances of this 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. Instead, the TunableMAC component relies on a single message queue with a freely configurable maximum length.[8] To reproduce in Castalia the same behavior of the TinyOS `AMSend` interface we thus added a second message queue in TunableMAC and configured the length of both queues to be of 1 packet. Messages coming from the RE or FE are then appropriately added to the corresponding queue. Routing and data frames can be easily distinguished since the former are broadcast messages while the latter unicast packets.

FE's snooping mechanism. As mentioned in section 3.4, the FE implements a snooping mechanism to timely detect congestion notifications and topology update requests. Since the TunableMAC module is actually designed to discard unicast packets that are not addressed to the current node, a little modification has been necessary to enable FE's snooping mechanism. In particular, we have disabled the packet deletion procedure and implemented a new mechanism that inserts a snooped packet in a `MAC_2_NETWORK_SNOOP_MSG` message, which is in turn passed to the CTPRouting compound module and finally to the FE module.

Link-layer acknowledgments. An additional feature we needed to add to the TunableMAC module concerns data link-layer acknowledgments. In particular, CTP requires the receiver of a unicast packet to explicitly acknowledge a successful reception (at the data link-layer) to the sender of the packet. However, TunableMAC does not provide for data link-layer acknowledgements. In order to enhance TunableMAC with this additional feature we first investigated how data link-layer acknowledgments are handled by the Chipcon $CC2420$, the radio chip embedded on the Telosb/Tmote Sky platform. In particular, upon receiving a unicast packet

[8]The length of the queue can be set in the `omnetpp.ini` configuration file. Please refer to the OMNet++ user manual [29] for further information.

at the MAC level, the $CC2420$ controller automatically generates an acknowledgment message and sends it to the sender of the packet. To this end, the corresponding option must be enabled on the radio chip [11]. On the other side, the sender keeps the packet in its queue (at the MAC level) and awaits for an acknowledgment. Upon reception of the acknowledgment the sender sets the *isAck* flag on the packet to 1 and triggers the `sendDone` event that is then handled at the routing layer. The packet is passed as a parameter along with the `sendDone` event and then removed from the queue so that the next sending operation can be executed. The *isAck* flag signals the successful reception of the packet to the routing layer of the sender. For further details about this split-phase mechanism, typical of the TinyOS operating system, as well as for more information about the handling of acknowledgments within the $CC2420$ please refer to [25] and [28, Sect. 5], respectively.

To emulate the behavior of the $CC2420$ within the TunableMAC module, we first made it immediately send an acknowledgment message back to the transmitter as soon as an unicast packet is received (at the MAC level). Additionally, we defined a new OMNet++ message[9], called MAC_2_NETWORK_SEND_DONE, which is sent by the TunableMAC to the CTPRouting module upon reception of an acknowledgment.

Backoff timers. Regarding (MAC level) backoff timers, we would like to point out that the TinyOS 2.1 distribution the backoff timers of the CC2420 are managed by the component `/tos/chips/cc2420/csma/CC2420CsmaP.nc`. In particular, the values of the initial and congestion backoff timers are set by calling the interfaces `initialBackoff` and `congestionBackoff` of the BMAC protocol. The former timer determines the delay of the first transmission attempt while the second is used in the case a busy channel is detected by the radio while sensing the carrier. Both values are chosen uniformly at random within a pre-specified time intervals, as detailed in table 4. TinyOS 2.1's `/tos/chips/cc2420/CC2420.h` file reports the values of these intervals expressed as number of ticks of a 32kHz clock while table 4 reports these same values in milliseconds.

The last parameter specified in table 4 is the acknowledgment timeout. This timeout sets the maximal time interval a sender waits for a packet to be acknowledged. If this timeout expires before an acknowledgment has been received, then the packet is passed back to the routing layer along with a message signalling the missing acknowledgment. Clearly, lowering the acknowledgment timeout may increase the throughput of the radio but at the same time cause unnecessary packet retransmissions. Within TinyOS 2.1 the acknowledgment timeout (`CC2420_ACK_WAIT_DELAY`) is set in the `/tos/cc2420/cc2420.h` file and expressed as the number of ticks of a 32kHz clock.

Packet overhead. Last but not least, we would like to quantify the overhead in terms of bytes that are attached at the MAC layer to both routing and data frames in order to make CTP work. To this end, we focus on the structure of the packets handled by the CC2420 radio chip, which is used on several WSN prototyping platforms, like the Tmote Sky. Within TinyOS 2.1, the structure of the MAC overhead is defined in the `cc2420_header_t` struct in the `/tos/chips/cc2420/CC2420.h` file, also reported in figure 6. This figure shows that the

[9]Please refer to the OMNet++ manual for messages definition and handling [29].

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

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

```
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.

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 [23, Sect. 6].

For the sake of completeness, figure 7 reports the generic packet format of a IEEE 802.15.4 frame, which is the standard used by the CC2420. 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. 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 [11, p.22]. According to this format, an acknowledgment packet has a total size of 11 bytes, 5 of which represent MAC layer overhead,

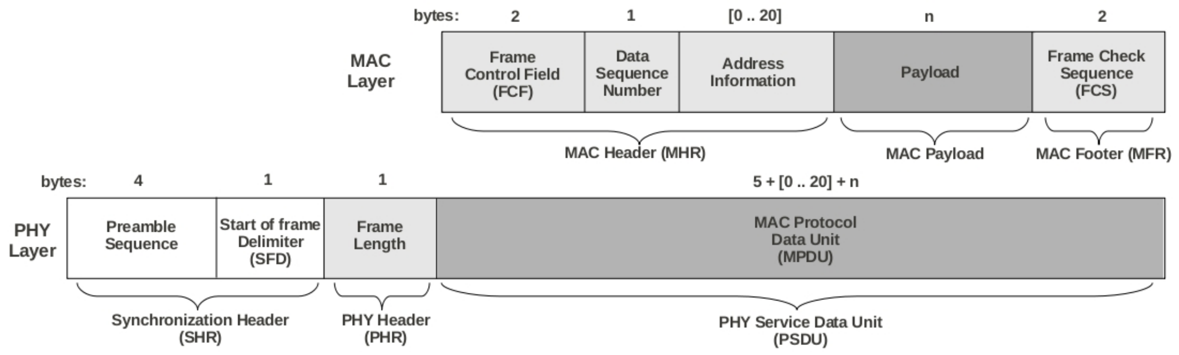


Figure 7: Schematic view of the IEEE 802.15.4 packet format [11].

while the other 6 are assigned by the physical layer, as shown in figure 3 and summarized in table 4.

4 Performance evaluation: setup and metrics

To test our implementation of the CTP protocol and investigate its performance under different conditions, we performed a simulation study. Before going into the analysis of experimental results in section 5, we describe here our simulation setup and the metrics we defined to observe and evaluate the behavior of CTP.

4.1 Simulation setup

Network topology. To perform our tests, we consider a rectangular region of side lengths L_x and L_y and area $A = L_x \cdot L_y$ over which we deploy a total number of N_{nodes} nodes. The nodes are assumed to be deployed uniformly at random, i.e., the x and y coordinates of the sensor nodes within the area A are the realizations of independent random variables uniformly distributed between 0 and L_x and between 0 and L_y, respectively. This physical topology is a common choice in sensor network literature and it often allows to benchmark protocols' performance. The network is assumed to have a single data sink at position (x_s, y_s). To understand the influence of the physical topology on the performance of CTP, we run our simulations on a high number N_{net} of different random network configurations. Table 5 summarizes the typical values we used for the parameters introduced above (and those defined below). Unless specified differently, we refer to these values for all the experiments presented in the next section 5.

Radio model. We assume all nodes in the network to have antennas with a omnidirectional radiation pattern and the wireless channel to be described by the *log shadowing* model. With this model, the reduction in power density of an electromagnetic radiation, called the path loss (PL), is a function of the distance d from the transmitter. In particular, the PL can be expressed as:

$$PL(d) = PL(d_0) + \eta \cdot 10Log(\frac{d}{d_0}) + X_\sigma. \tag{5}$$

Parameter	**Value**	**Unit**
Topology		
L_x	250	meters
L_y	250	meters
N_{nodes}	100	-
(x_s, y_s)	$(0, 0)$	meters
Radio model		
d_0	1	meters
$PL(d_0)$	54.2247	dB
η	2.4	-
X_σ	0	-
R_{TX}	50	meters
Data Traffic		
$T_{s(time)}$	5	seconds
I_p	$[0.5, 1]$ with step 0.05	-
$\lvert I_p \rvert$	11	-
Physical Process		
$N_{sources}$	1	-
K	1	-
a	2	-
$V_1(0)$	10	-
$s_i(0) = (x_i(0), y_i(0))$	$(125, 125)$	meters
General		
N_{net}	50	-
N_{rounds}	50	-

Table 5: Relevant parameters of our experimental setup and their correspondent values.

In equation 5 $PL(d0)$ represents the (known) path loss at a reference distance $d0$, η is the path loss exponent, and X_σ is a gaussian zero-mean random variable with standard deviation σ. Table 5 summarizes the values we assigned to these parameters in the context of our simulation study. With this set of parameters and assuming the absence of any interference, the communication range of the radio is 50 meters. For a detailed description about the derivation of this value please refer to Castalia 1.3's user manual [6]. To determine whether packets transmitted by nearby nodes collide or not we resort to the *additive interference model* [6]. Taking into account the possibility of collisions clearly makes the experimental results presented in section 5 more realistic.

Data traffic. To generate data traffic, we consider a typical data collection scenario. We assume that the nodes are required to provide "snapshots" of the values of a physical phenomenon at regular time intervals. The temporal sampling frequency $F_{s(time)}$ is fixed a priori and equal for all nodes, i.e., the network is programmed to wake up every $T_{s(time)} = F_{s(time)}^{-1}$ seconds, sample the physical phenomenon (e.g., gather a reading from the temperature sensor), send the data to a central sink and go to sleep again. To transport the collected data to the sink the network establish, after each wake-up, a data collection tree using the CTP protocol. We refer to the sequence *wake-up−sample−send−go to sleep again* as *round* or *epoch.*

For each of the generated network configuration we run the simulation for a number of rounds N_{rounds}. This is necessary to gather relevant statistics for the specific network configuration. Nonetheless, the actual simulation results are affected by the presence of several sources of randomness, like the amount and origin of the generated data traffic or communication failures due to collisions. As reported in table 5, we consider a value of N_{rounds} equal to 50 to be sufficient to gather statistically significant results.

To generate data traffic, we let selected subsets of the nodes transmit their data packets. In particular, we let the nodes decide with probability p whether they actually participate in the sensing task. At each round, each node chooses a number between 0 and 1 uniformly at random. If the value is lower than p, then the node will sense and transmit data, otherwise it will not. Thus, a value of p of 0.6 implies that a node will participate in the sensing task with a probability of 60%. In this preliminary study, the value of p is assumed to be known a priori and be the same for all nodes.To observe the behavior of the network as the number of nodes participating in the sensing task varies, we let p assume values in the set $I_p = 0.5 : 0.05 : 1$, thus from 0.5 to 1 with steps of 0.05 (so we consider a total of $|I_p| = 11$ different values of p).

In our experiments we thus consider, for each of the N_{net} network configurations, a total of N_{rounds} for each of the $|I_p|$ values of p. We therefore collect a total of $N_{net} \times N_{rounds} \times |I_p|$ datasets. Considering the values reported in table 5 this equals to 27500 datasets from which we can gather significant statistics.

It is here important to underline that although only a subset of the nodes actually gathers and transmits data, all $N_{nodes} + 1$ nodes in the network participate in the construction of the collection tree and thus, contribute to the data reporting.

Physical process. To simulate a physical process whose samples are collected and reported to the sink by the sensor nodes, we used the correspondent built-in primitive of the Castalia simulator. In particular, Castalia allows to generate a sensor field whose value at position s and time t is given by the superimposition of the effects of a number $N_{sources}$ sources. The value of the ith source at time t is indicated as $V_i(t)$ and must be given as input. The value of the sensor field at each position s and time t is then expressed by the following equation:

$$V(s,t) = sum_{i=1}^{N_{sources}} = \frac{V_i(s,t)}{(K \cdot d_i(t) + 1)^a} \tag{6}$$

where $d_i(t)$ is the distance of point s from the ith source at time t and K and a are parameters that control the way a source value spreads over space and time.

Since the actual values of the physical process are not critical for this study, we set K and a to the default values, as reported in table 5. The sources are supposed to be “static”, i.e., during the simulation their position s_i and the value they assume at that position does not change over time.

Timing. We assume the nodes in the network to be synchronized so that wake-up and sleep cycles can be easily scheduled. In particular, we let the nodes wake up every minute and remain active for 16 seconds before turning their radios and all other circuitries off again. Since the sleep phase lasts for 44 seconds the duty cycle of the complete data collection protocol is 26.6% ($100 \cdot \frac{16}{16+44}$). The active phase is divided in three successive intervals: the *startup*, the *CTP setup*, and the *data transmission* intervals. In the startup phase, which lasts in total just for few microseconds, nodes power up their circuitries and set their radios' duty cycle to be 100%. Immediately after startup nodes enter the CTP setup phase during which nodes exchange beacons and establish a routing tree having the sink as its root. We set the total duration of this phase to be 11 seconds. This value has been set empirically and could clearly be reduced if our CTP implementation is used in different scenarios. After completion of the CTP setup phase the actual data collection can start. During this phase, which lasts for 5 seconds, node send their data (assumed they have been selected for data sampling) and act as forwarder for other nodes' data packets. The sink node is assumed to have unlimited power supply and is thus always active.

4.2 Metrics

We evaluate the performance of CTP through a set of metrics that can outline the most significant features of the protocol. Table 6 summarizes the notation we will use in this section to define such metrics, which are in turn reported in table 7.

Data delivery ratio. The very first metric we are interested is the data delivery ratio (DDR). We define the DDR as the ratio between the number N_{sen} of data values collected and sent by the nodes and the number N_{rec} of data values received at the sink (without counting duplicates). We preferred to name this metric *data* delivery ratio, instead of *packet* delivery ratio, to stress the fact that it does only take into account the number of data values that the network can successfully deliver to the sink. Clearly, a DDR equal to 1 means indicates that the network can deliver all the data to the sink. In the worst case, none of the collected data values reaches the sink. This may happen if the sink is disconnected from the network and causes the DDR to be zero. If the unlikely case that no single data value is collected by the nodes, and thus $N_{sen} = N_{rec} = 0$, the value of the DDR is forced to be 1.

Parameter	**Description**
N_{sen}	Number of data values collected and sent by the nodes.
N_{rec}	Number of data packets received at the sink (without counting duplicates).
N_{beac}	Total number of beacons sent in the network (control traffic).
N_{data}	Total number of data packets sent by the network (data traffic) to deliver the N_{sen} data values to the sink
N_{dup}	Total number of duplicate data packets received at the sink

Table 6: Basic quantities used to defined the metrics listed in table 7.

Control overhead. A second interesting metric, which we named the control overhead (COV), allows quantifying the total amount of traffic that is generated in order to dispatch the N_{sen} data values to the sink. To this end, we define the total traffic due to data transmission as N_{data} and the control traffic necessary to setup and maintain the CTP routing structure as $N_{control}$. N_{data} is given by the sum of the total number of data packets sent by

Acronym	**Name**
DDR	Data delivery ratio
COV	Control overhead
AvgTHL	Average number of hops
MaxTHL	Maximal number of hops
N_{dup}	Number of duplicate packets N_{dup}

Table 7: Metrics used to evaluate the performance of CTP.

the application layer (N_{sen}) and the total number of data packets forwarded by each node, including retransmissions due to missing acknowledgments (which may result in the generation of duplicate packets). On the other hand, $N_{control}$ includes the total amount of beacons sent throughout the network to establish and maintain the routing tree. Since acknowledgments are managed at the radio level, we do not account for them in $N_{control}$.

Hop count. Further interesting properties of CTP's routing tree are the average and maximal number of hops packets travel before reaching the sink. Indeed, the routing tree generated by CTP changes depending on the physical topology of the network, thus on the network configuration at hand, and the specific condition of the wireless channel. This translates in possibly different numbers of hops traveled by a packet, on average and worst case, before reaching the sink. As detailed in section 2.2 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 sink. We will thus consider the average and maximal THL as additional metrics to describe the performance of CTP and refer to them as $AvgTHL$ and $MaxTHL$, respectively.

Duplicate packets. As described in section 3.4 CTP features a mechanism to detect duplicate packets. Nonetheless, a given number of duplicates may reach the sink thus inducing an unnecessary overhead. In order to quantify this overhead, we count the total number of duplicate packets N_{dup}.

5 Performance evaluation: analysis of experimental results

In this section we finally report an evaluation of the performance of CTP based on its implementation for the Castalia simulator. To this end, we report and comment experimental results related to the metrics introduced in the previous section, i.e., the data delivery ratio, the control overhead, the hop count, and the number of duplicate packets.

5.1 Data delivery ratio

To evaluate the ability of CTP to reliably report data to a central sink, we compute the DDR achieved in 50 different network configurations. As detailed in section 4.1 we generate the network configurations uniformly at random and, for each configuration and value of p, we run 50 rounds. At each round, nodes wake up, construct the routing tree and use it to report data to the sink. Each nodes generates a data packet with probability p. We repeated this experiment for several values of p, ranging from 0.5 to 1, thus only a (randomly selected) fraction of the nodes actually transmits data packets. However, all the nodes collaborate in relaying the packets by keeping their radio active. For each round and network, we then computed the DDR as the ratio between the number of packets delivered to the sink and the number of packets originally transmitted by the nodes. For further details about the simulation setup, please refer to section 4.1.

Figure 8 shows the number of sent and received packets for one of the 50 generated network configurations (network 2) and three different values of the probability of activation p. This plot shows that when the probability of activation p is 0.5 the number of received packets almost always coincides with the number of sent packets. Thus, the DDR of CTP for the corresponding network configuration is equal to 1 (or 100%). As the value of p and thus the total number of packets sent by nodes increases, the DDR decreases. This is due to the fact that increasing the number of transmissions causes more collisions and congestions, and thus packet losses, to occur.

The matrix plots reported in figure 9 show the DDR for all the 50 networks and 50 rounds and the two values of p 0.5 and 1. A pixel of this matrix represents the DDR measured for the network configuration indicated on the y-axis and the round specified on the x-axis. For instance, the second row (from the bottom) shows the DDR for network 2 over the 50 rounds. The color of each pixel codes the actual value of the DDR, according to the scale reported on the right side of the plot. Figure 9 shows that the DDR of networks 1 and 40 is constantly zero. This is due to the fact that for these configurations the sink is disconnected and thus no single data packets manages to reach the sink. For all other networks, when $p = 0.5$ the DDR is almost always above 95%. On the other side, especially for $p = 1$, there are several cases in which the DDR is as low as 80% or even lower. For instance, network 41 shows a very bad performance for $p = 1$. This is due to the particular topology of this network, depicted in figure 10. Network 41 has indeed three nodes close to the sink that can act as last-hop relays for data packets. But before reaching one of these three nodes the packets must first be routed through one of their neighbors. However, the distance between these latter nodes and the three relay nodes is $50m$, which coincides with the transmission range of the nodes. Therefore, this hop towards the sink is very unstable and several retransmissions may be necessary to successfully deliver a packet. This causes a general slowdown of the protocol since packets remain in the transmission queue until they have been acknowledged or they are dropped after a maximum number of retransmission attempts (30 in our simulations). As a consequence, the buffer may quickly fill up with packets waiting to be sent and thus become unable to accept new incoming packets, which must thus be dropped. Since the whole data traffic must be conveyed through the three “last mile” nodes, the number of packets dropped for full buffer may be high, causing the poor performance in terms of DDR. Additionally, in our simulation setup the network turns off after a certain timeout. Clearly, packets residing in the buffer as this timeout expires must be dropped too.

Unfavorable topologies like the one of network 41 may cause CTP to yield a very low DDR. However, the average DDR achieved by CTP on the large and diverse set of configurations we considered is highly satisfactory. This is also confirmed by figures 11 and 12. The first reports the overall average DDR (computed over all configurations and rounds) as the value of p increases. Each blue dot on this plot represents the average DDR of a single network configuration (computed over the 50 rounds). For each value of p, the set of blue dots give a measure of the standard deviation of the average DDR. The second reports the minimum, average, and maximum DDR (computed over the 50 rounds) for all network configurations and $p = 0.5$ and $p = 1$. Please note that we compute these statistics on the data corresponding to all networks but configurations 1 and 40.

5.2 Control Overhead

A further interesting metric to evaluate the performance of CTP is the control overhead. As specified in section 4.1, we define the control overhead as the ratio between the number of data packets sent (or forwarded) by the nodes and the number of routing beacons sent throughout the network to establish and maintain the routing tree. We refer to this quantities as the *data traffic* and *control traffic*, respectively.

Figure 13 shows the control and data traffic for network 2, over all rounds and for $p = 0.5$, $p = 0.75$, and $p = 1$. The number of both control and data packets oscillates around an average value, while their ratio, thus the control overhead, decreases as p increases. This is also outlined in figure 14 that visualizes the control overhead for network 2. As p increases, the number of data packets sent throughout the network increases and thus the control overhead decreases. This means that, as expected, the effort invested in building and maintaining the routing becomes less predominant as the number of data packets forwarded through the tree increases. This applies to all network configurations and rounds, as shown by figure 15. Figures 16 and 17 also allow to appreciate the variability of the control overhead over all networks, round and values of p. In particular, figure 16 shows that networks 14 and 25 have an average control overhead significantly higher than the overall average. This is due to the fact these configurations have small clusters of (at least two) nodes that are disconnected from the rest of the network but within the communication range of each other. The nodes in these clusters will thus exchange a high number of control packets in an attempt to establish a route to the sink. Also, since in this situation the pull flag is set to 1, the nodes will exchange control beacons at the maximal rate.

5.3 Hop count

Figures 18 and 19 show the average and maximal value of the Time Has Lived (THL) metric for all packets that reach the sink in every round and for all the 50 network configurations. These plots show that, for $p = 0.5$, both the average and maximal values of the THL for a specific network configuration are constant across the 50 rounds. On the contrary, when $p = 1$, both the average and maximum THL are more variable, since the network must deal with a higher number of packets and, thus, possibly congested routes.

Figures 20 and 21 also show the overall average and maximal THL for the different values of p considered in the experiment. The blue dots represents the average over 50 rounds of the average and maximal THL for each of the 50 network configurations. In three cases, not shown in figure 21, the (average) maximal number of hops is higher than 30. In particular, this happens for network 28, round 19 (MaxTHL is 35); network 16, round 42 (MaxTHL is 35); network 33, round 12 (MaxTHL is 49). For all other rounds and networks, the maximal THL is always lower than 30. Such high values of THL are usually due to the occurrence of routing loops. In the vast majority of the cases, however, the overall average THL is between 5 and 6 while the maximal THL is between 10 and 17, as shown in figures 20 and 21.

5.4 Duplicate packets

A further interesting metric to consider when evaluating the performance of CTP is the number of duplicate packets eventually reaching the sink. As detailed in section 3.4, duplicate packets are generated when acknowledgments get lost or are not received within a given time-out. Figure 22 shows the number of duplicate packets for all network configurations and rounds, while figure 23 shows the corresponding average for each network and over all networks as p increases. As expected, the number of duplicate packets increases as the value of p increases. The presence of duplicate packets clearly contributes to congest the network and thus negatively affects the performance of CTP.

6 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. The validity of the implementation has been tested through an extensive simulation study, which also confirmed the convincing performance of CTP in terms of data delivery ratio.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, March 2002.

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

[3] Aline Baggio. Wireless Sensor Networks in Precision Agriculture. In *Proceedings of the First Workshop on Real-World Wireless Sensor Networks (REALWSN 2005)*, Stockholm, Sweden, June 2005.

[4] 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.

[5] 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.

[6] Athanassios Boulis. Castalia User Manual (version 1.3). `http://castalia.npc.nicta.com.au/pdfs/Castalia%20-%20User%20Manual.pdf`.

[7] 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.

[8] 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.

[9] Phil Buonadonna, David Gay, Joseph M. Hellerstein, Wei Hong, and Samuel Madden. TASK: Sensor Network in a Box. In *Proceedings of the 2nd IEEE European Workshop on Wireless Sensor Networks and Applications (EWSN 2005)*, Istanbul, Turkey, February 2005.

[10] 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.

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

[12] Crossbow Technology Inc. `www.xbow.com`.

[13] 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.

[14] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 1999)*, pages 263 – 270, Seattle, WA, USA, 1999.

[15] 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`.

[16] 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`.

[17] 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`.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] Philipp Levis and David Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[26] 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.

[27] 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.

[28] David Moss, Jonathan Hui, Philip Levis, and Jung Il Choi. TinyOS Enhancement Proposal (TEP) 126: CC2420 Radio Stack. `www.tinyos.net/tinyos-2.x/doc/pdf/tep126.pdf`.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An Analysis of a Large Scale Habitat Monitoring Application. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, Baltimore, MD, USA, November 2004.

[35] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. PermaSense: Investigating Permafrost with a WSN in the Swiss Alps. In *Proceedings of the Fourth ACM Workshop on Embedded Networked Sensors (EmNets 2007)*, Cork, Ireland, June 2007.

[36] 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.

[37] 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.

[38] 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.

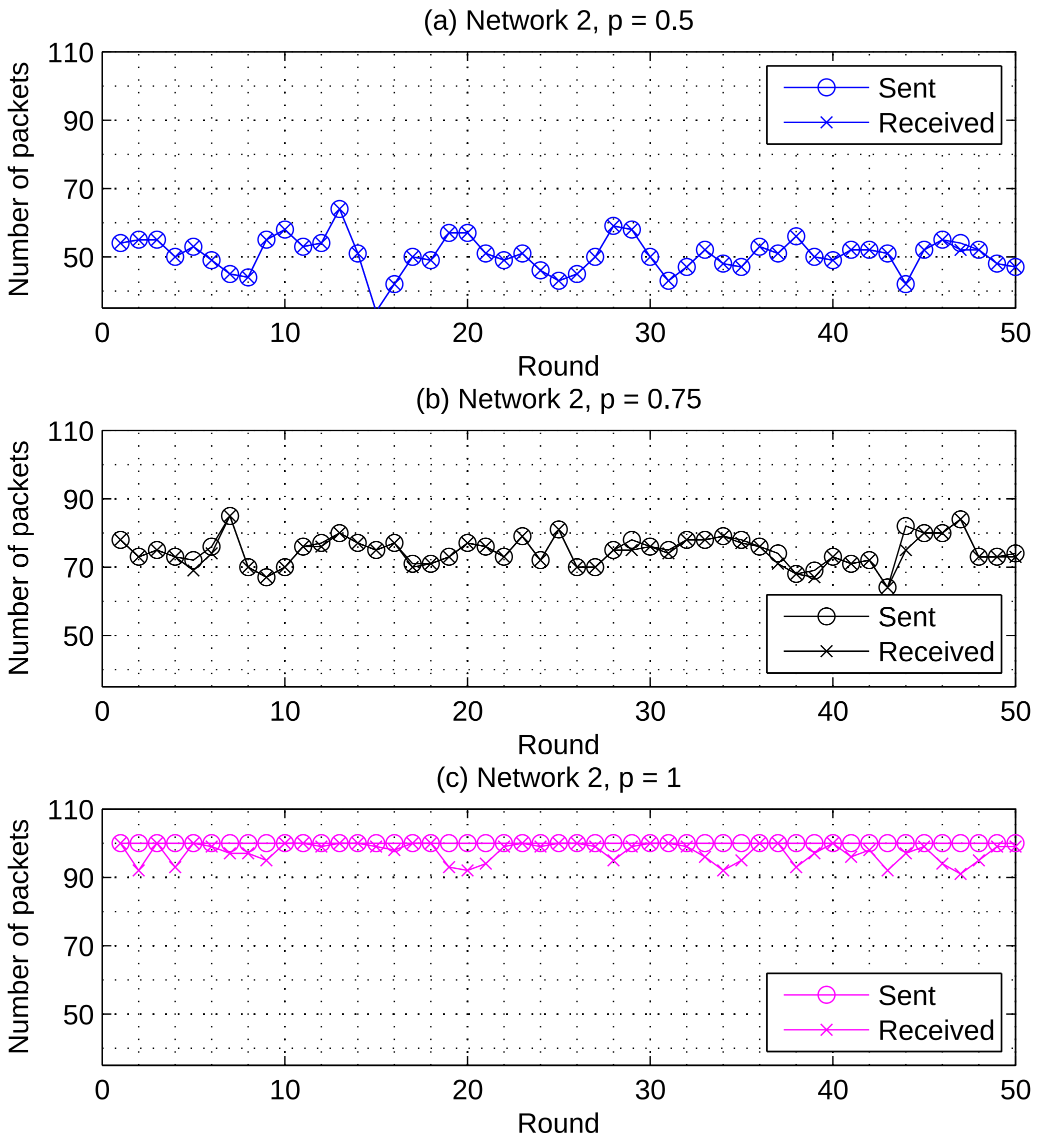
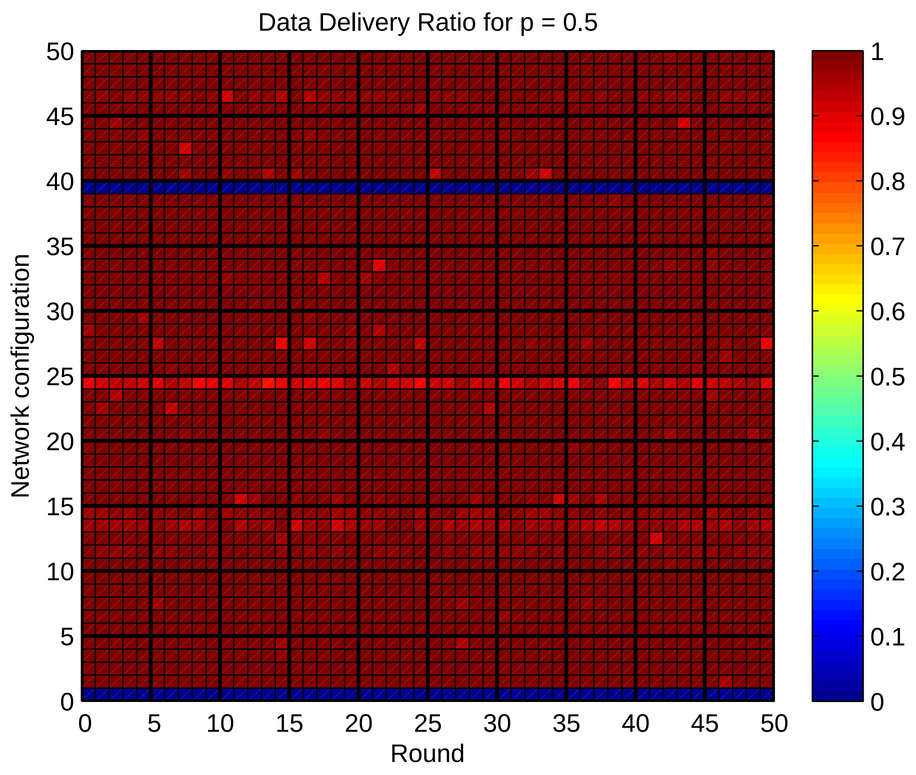


Figure 8: Number of sent and received packets for the network configuration 2 and $p = 0.5$, $p = 0.75$, and $p = 1$.



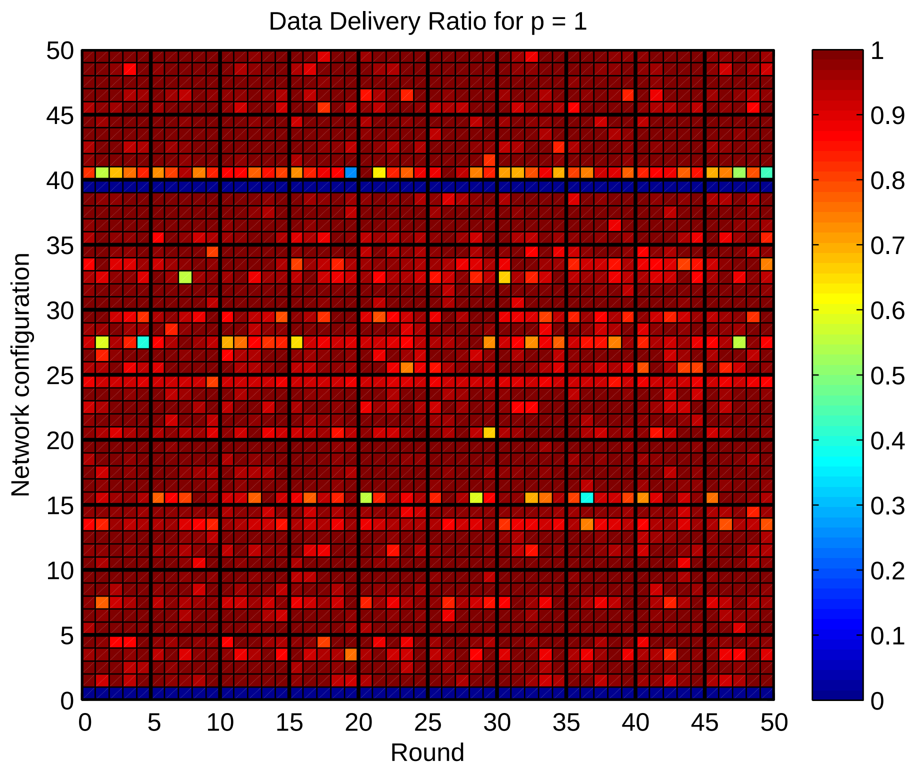


Figure 9: DDR for all the network configurations and rounds, for $p = 0.5$ and $p = 1$.

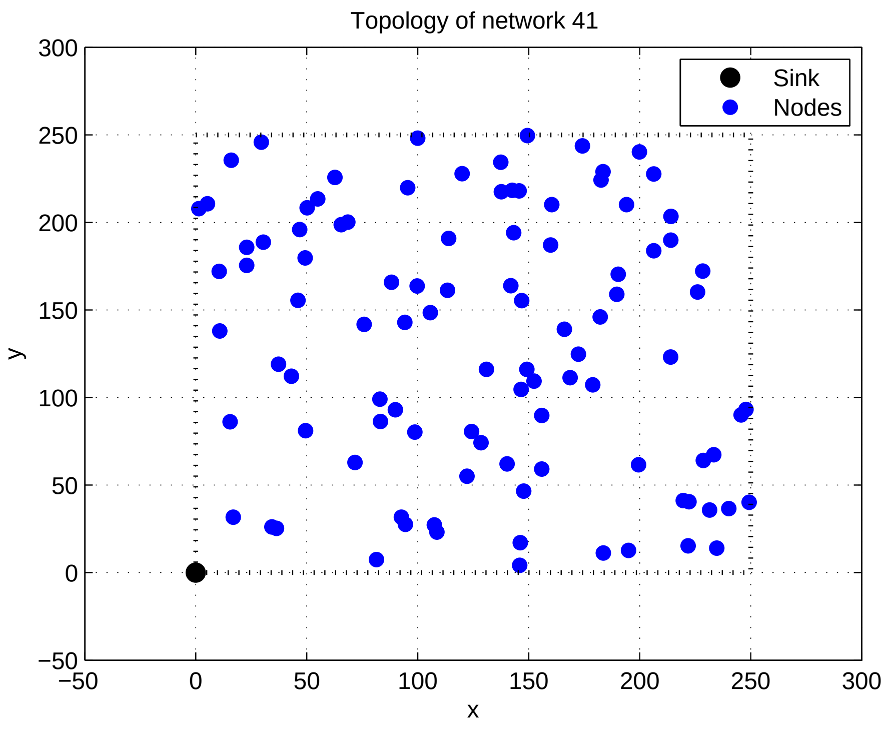


Figure 10: Physical topology of network configuration n^o 41.

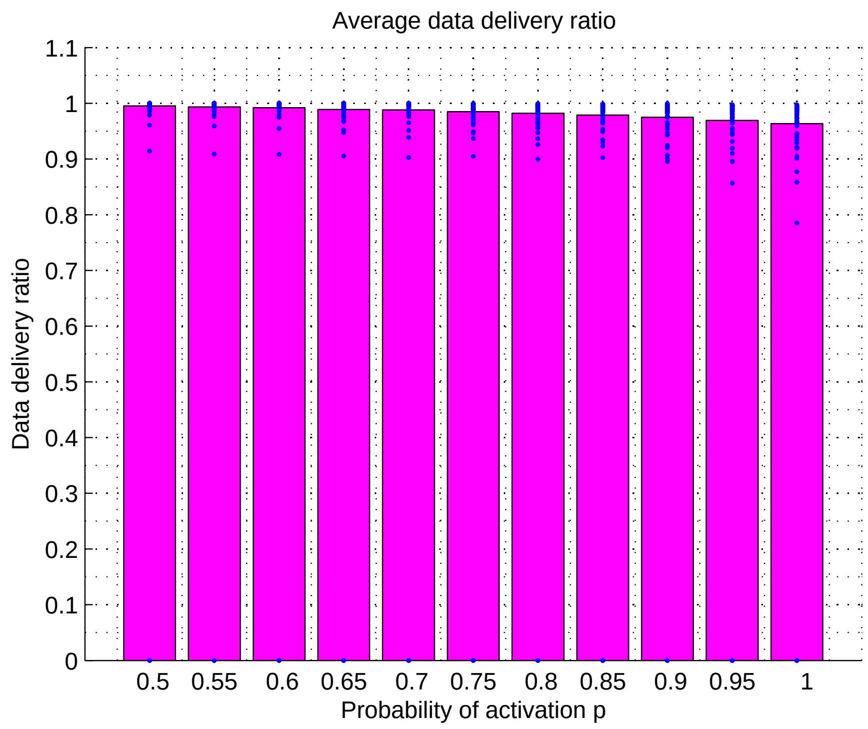


Figure 11: Average data delivery ratio for all networks and values of p.

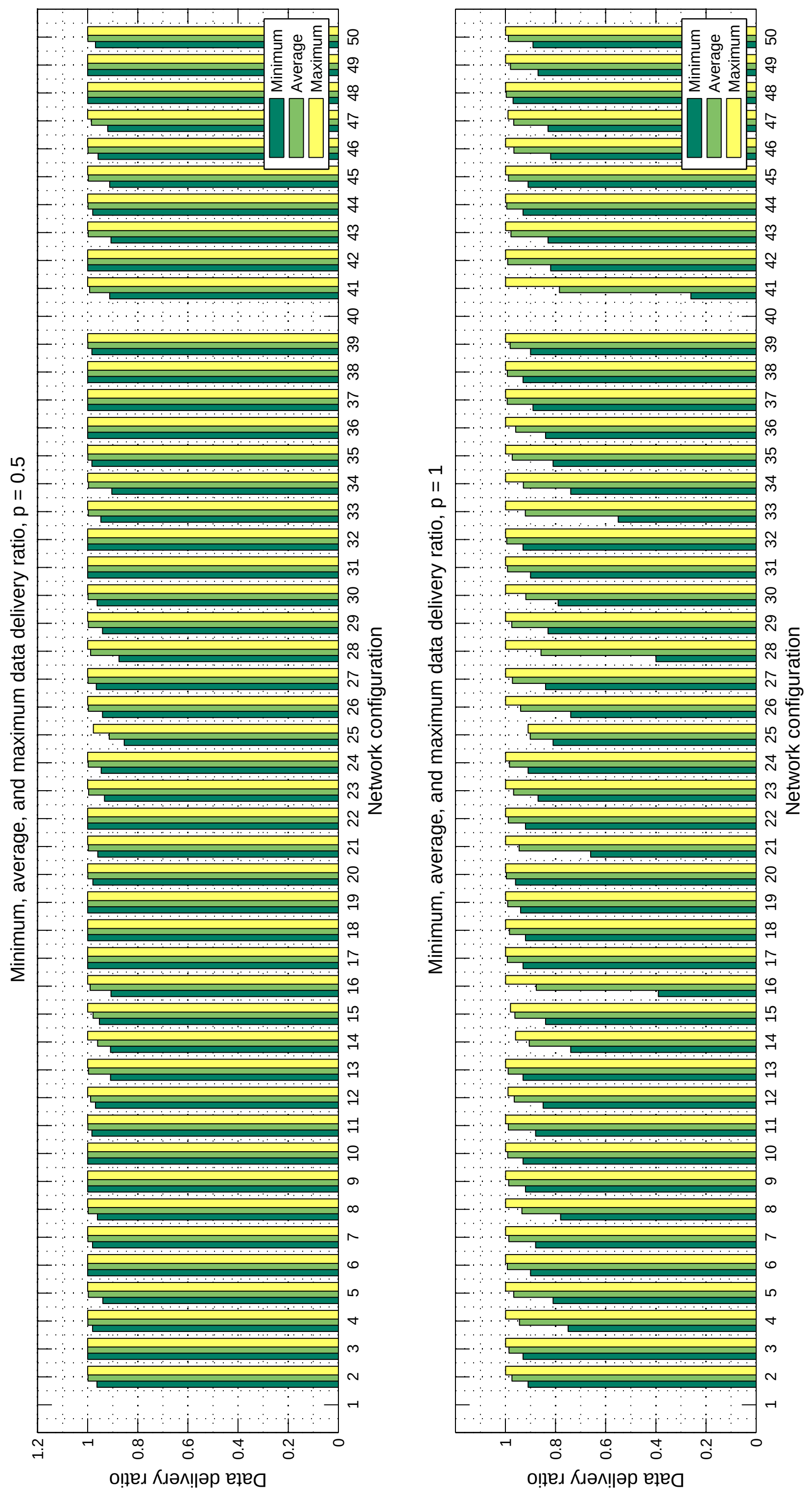


Figure 12: Minimum, average, and maximum data delivery ratio for all networks and $p = 0.5$ and $p = 1$.

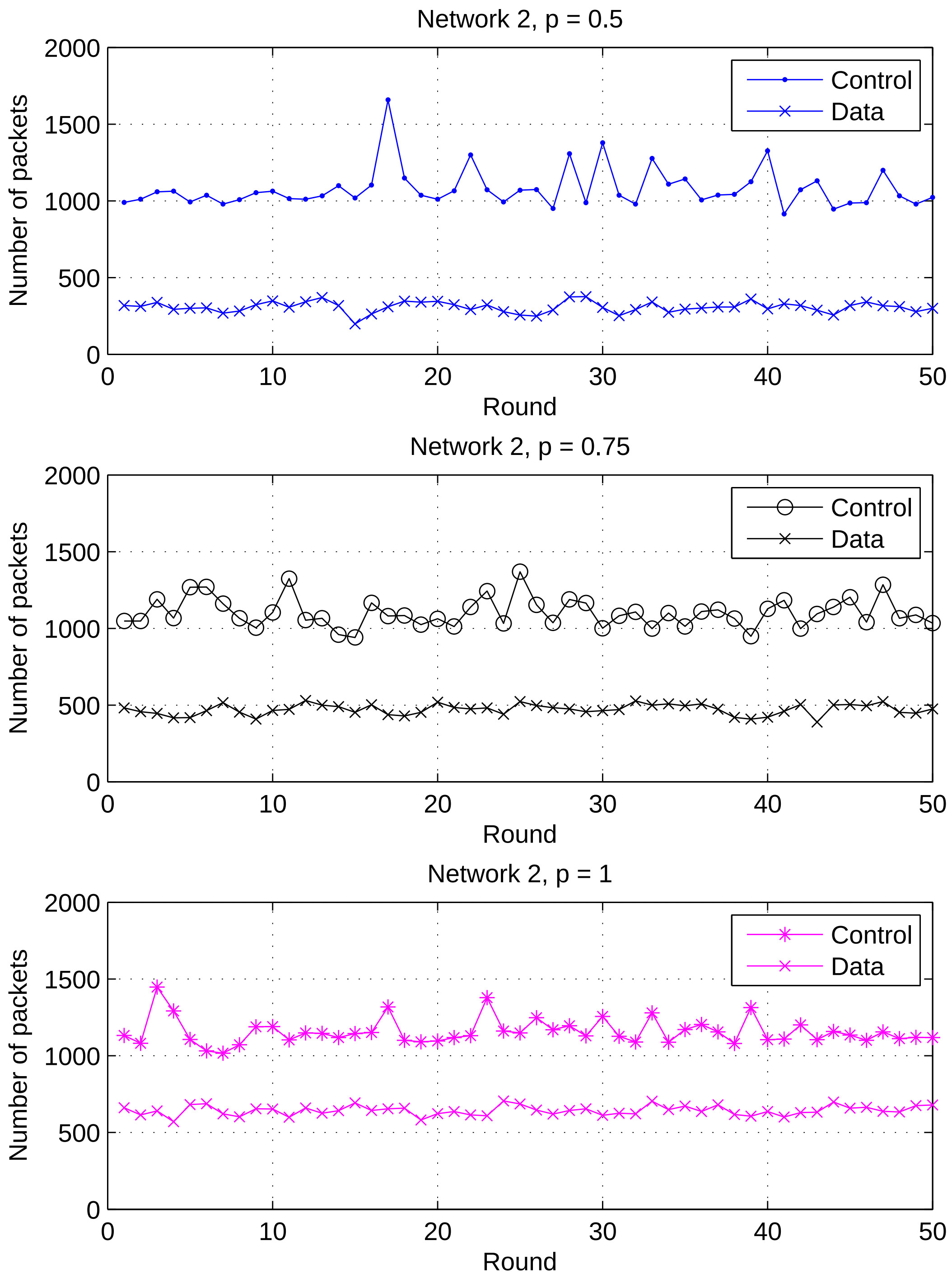


Figure 13: Number of control and data packets sent throughout network 2 for $p = 0.5$, $p = 0.75$, and $p = 1$.

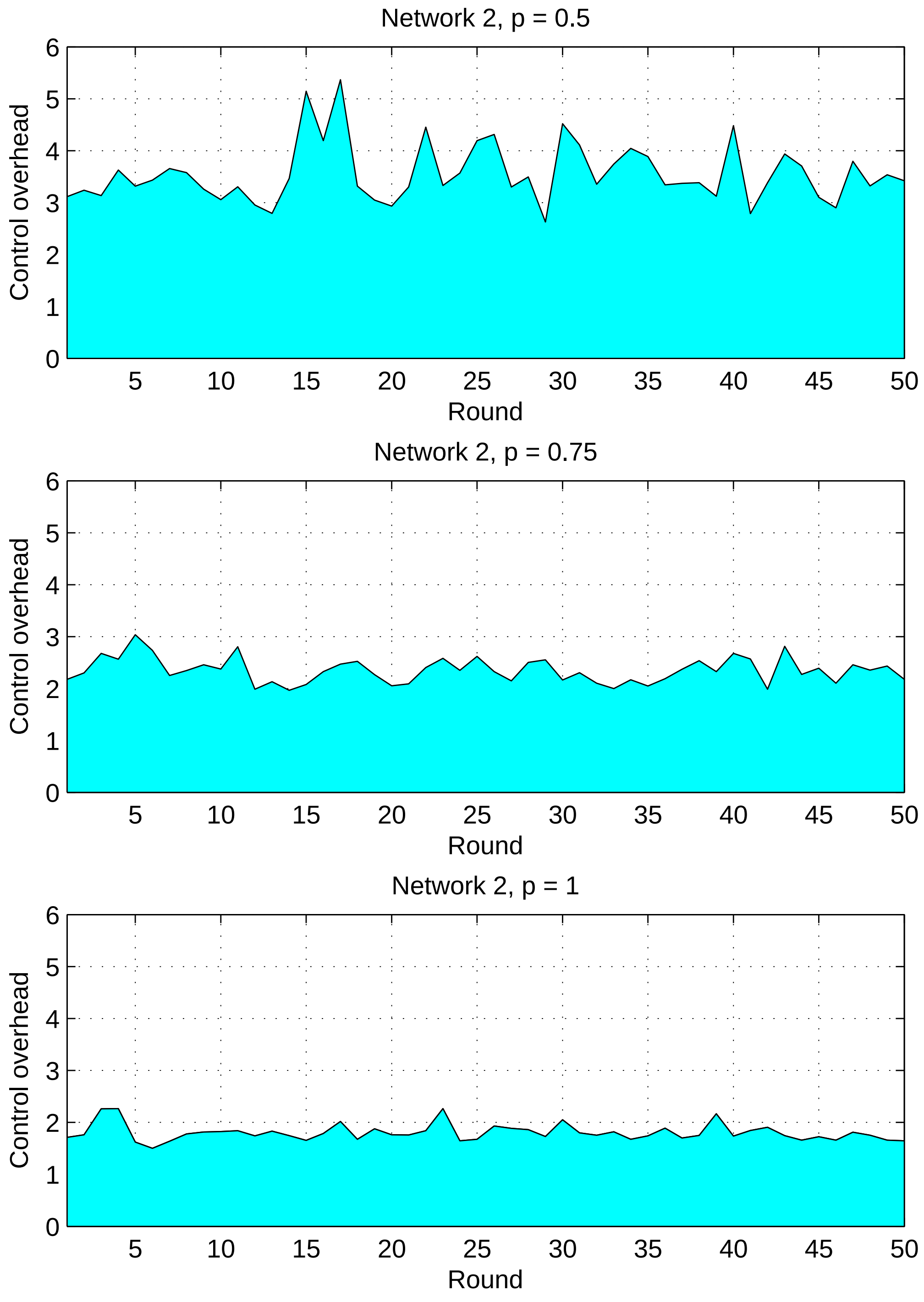
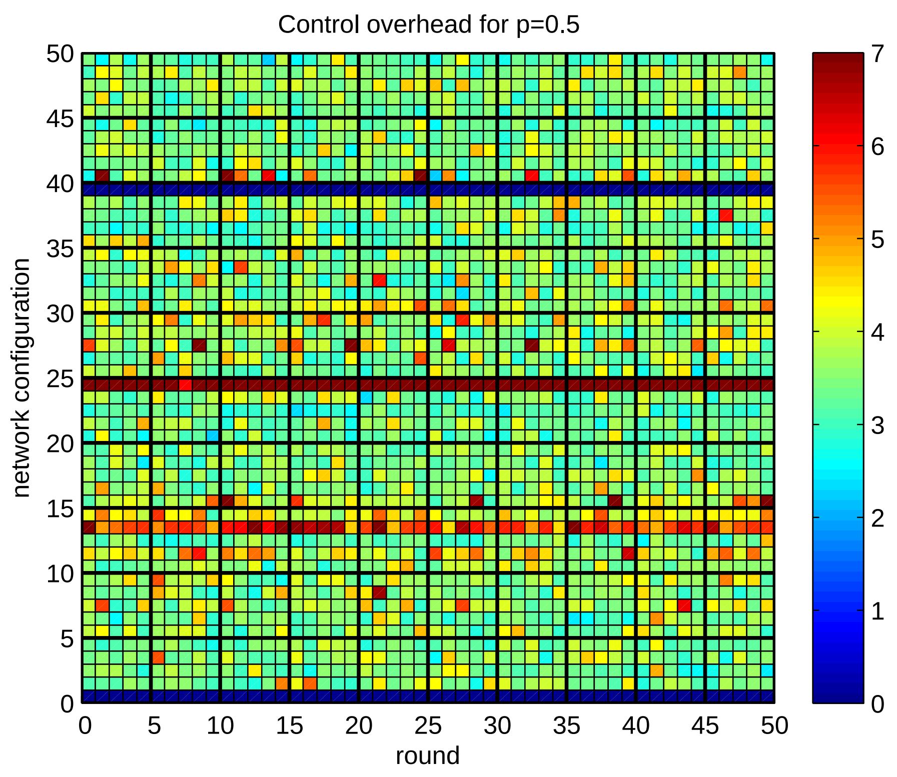


Figure 14: Control overhead for network 2 for $p = 0.5$, $p = 0.75$, and $p = 1$.



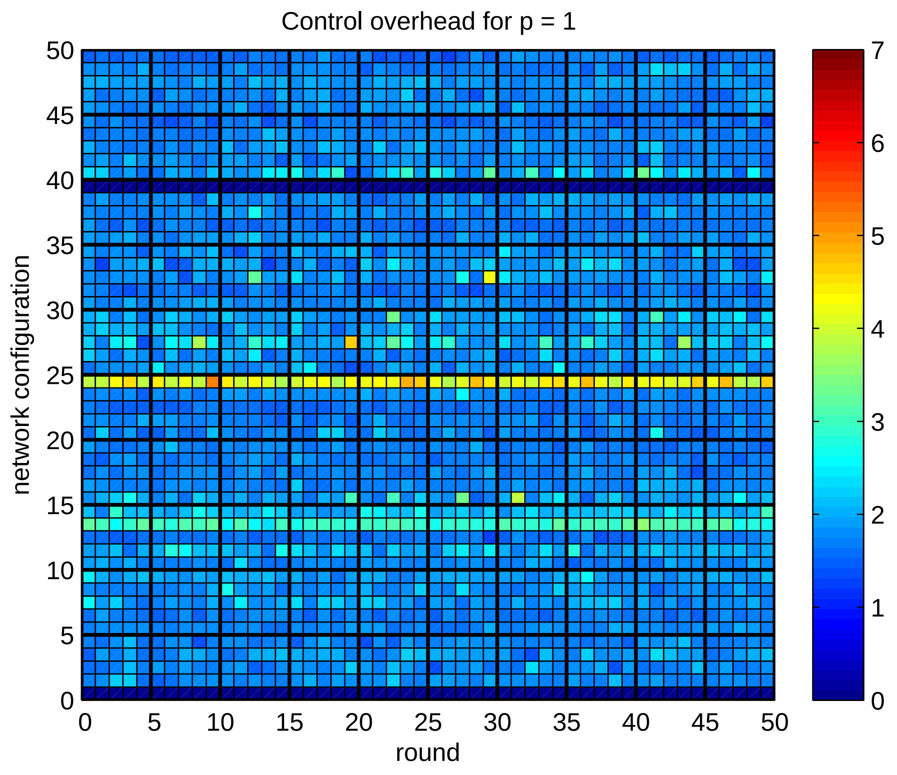
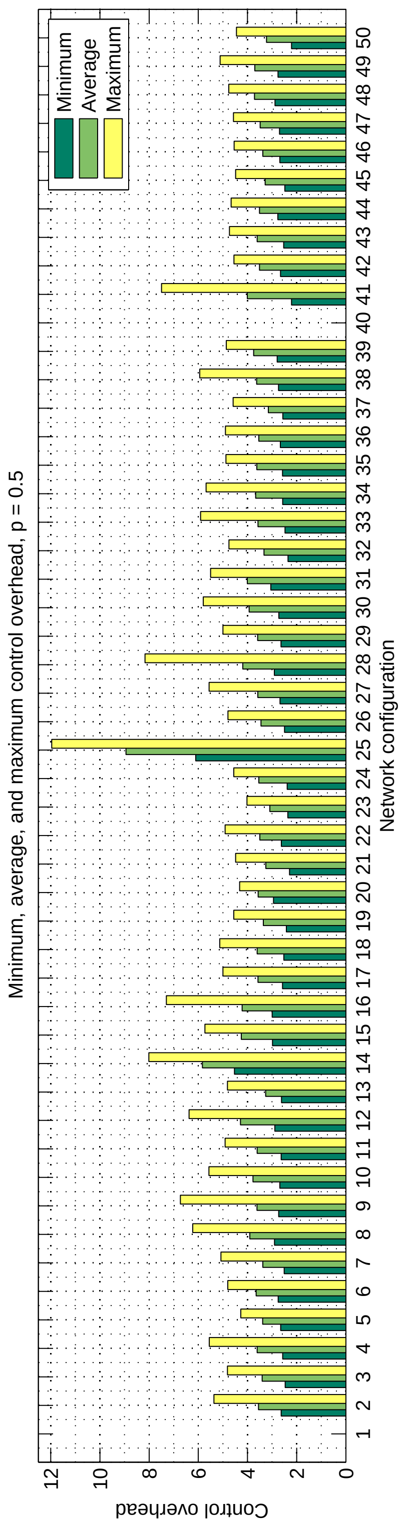


Figure 15: Control overhead for all networks and rounds, for $p = 0.5$ and $p = 1$.



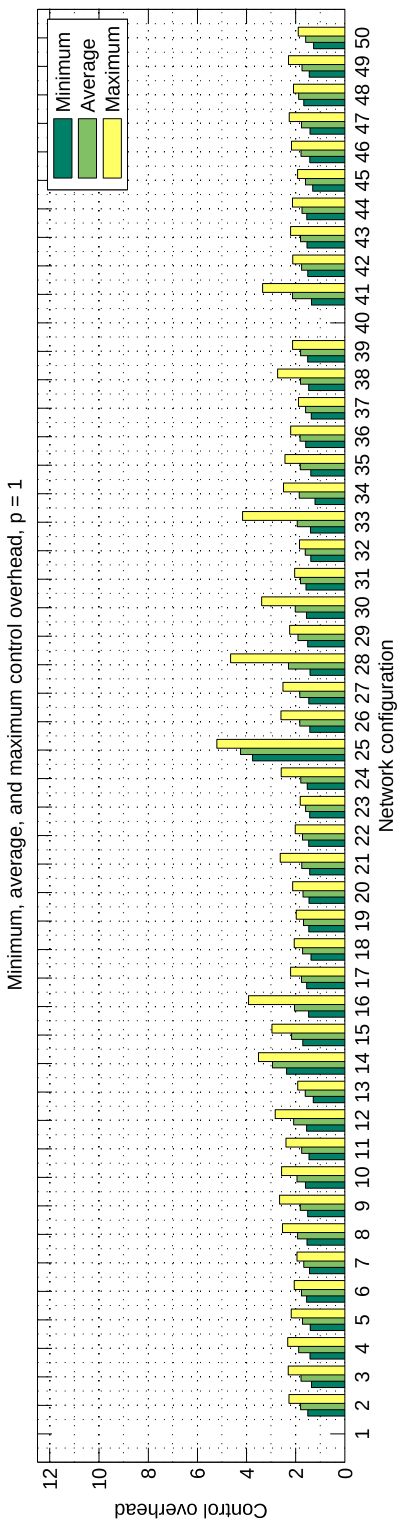


Figure 16: Minimum, average, and maximum control overhead for all networks and $p = 0.5$ and $p = 1$.

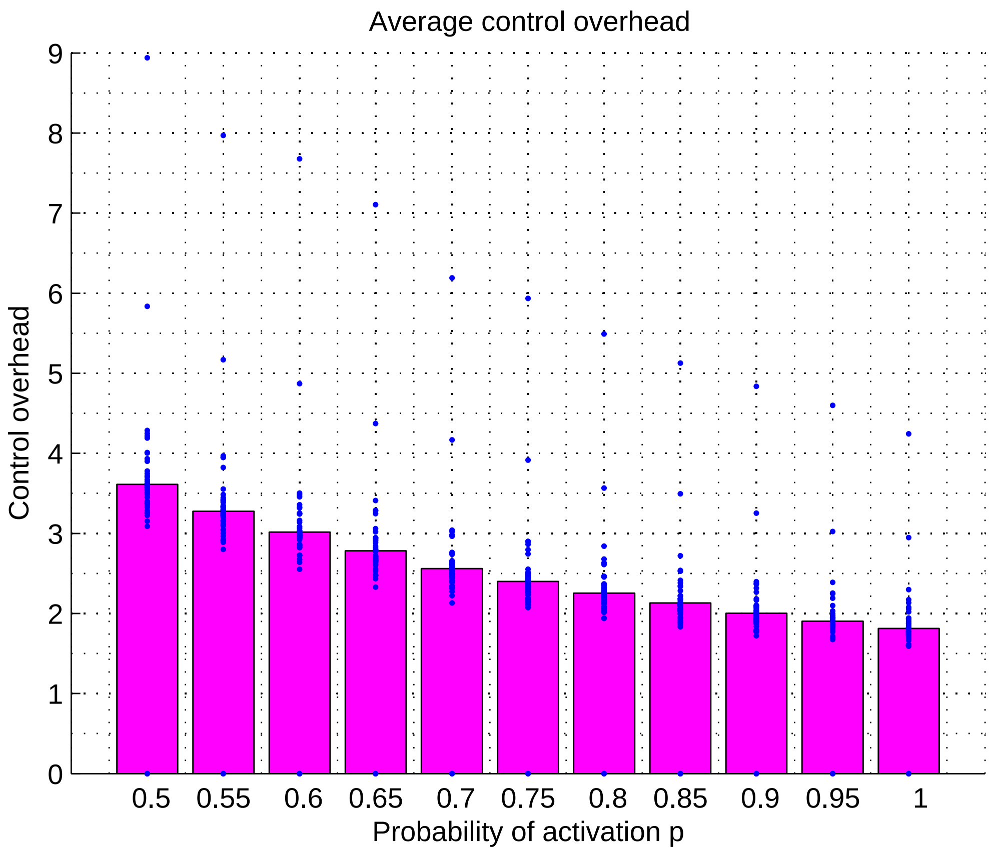
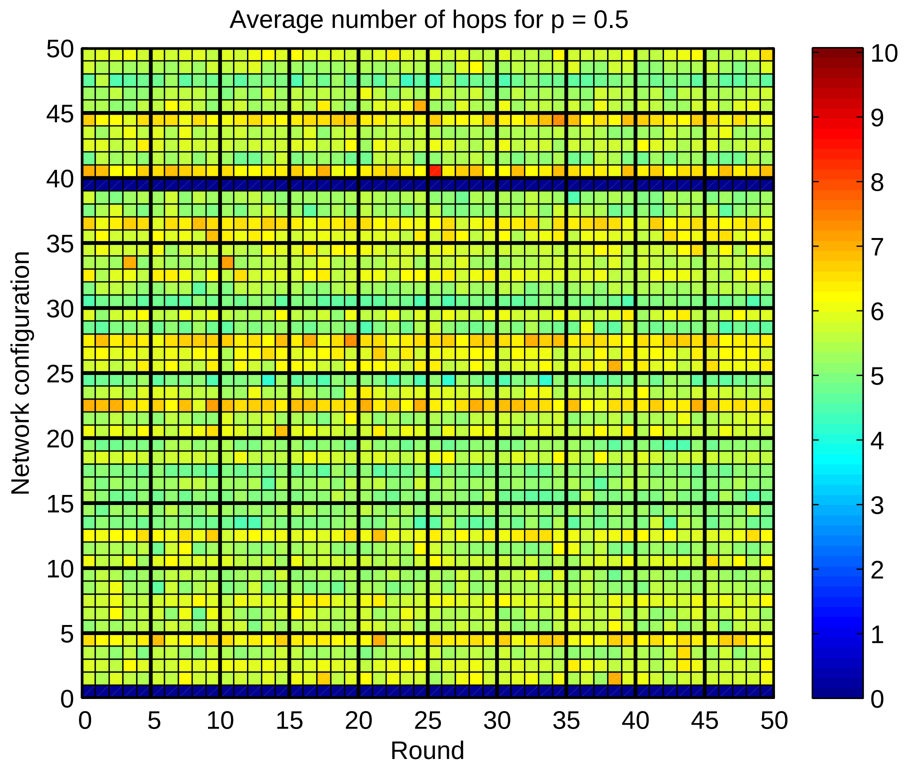


Figure 17: Average control overhead for all networks as p increases.



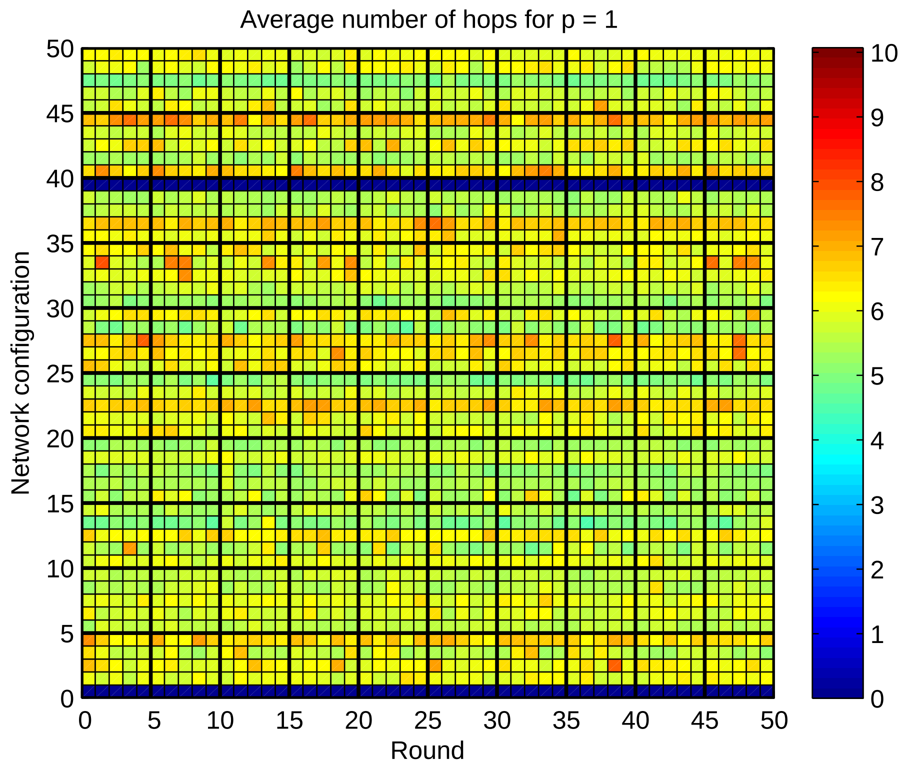
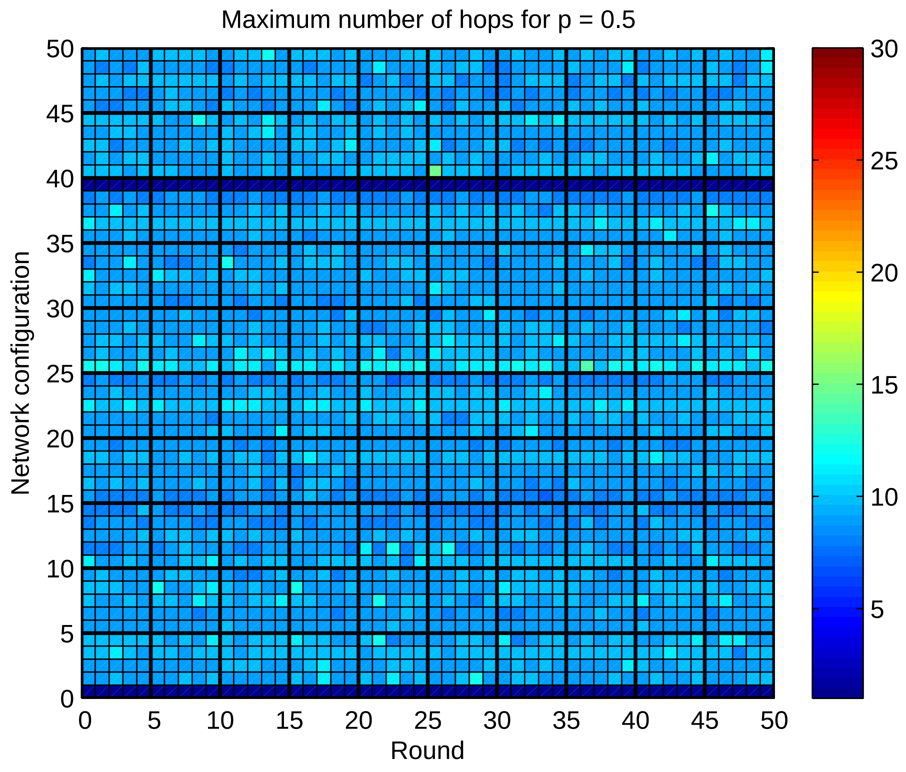


Figure 18: Average THL for all the network configurations and rounds and $p = 0.5$ and $p = 1$.



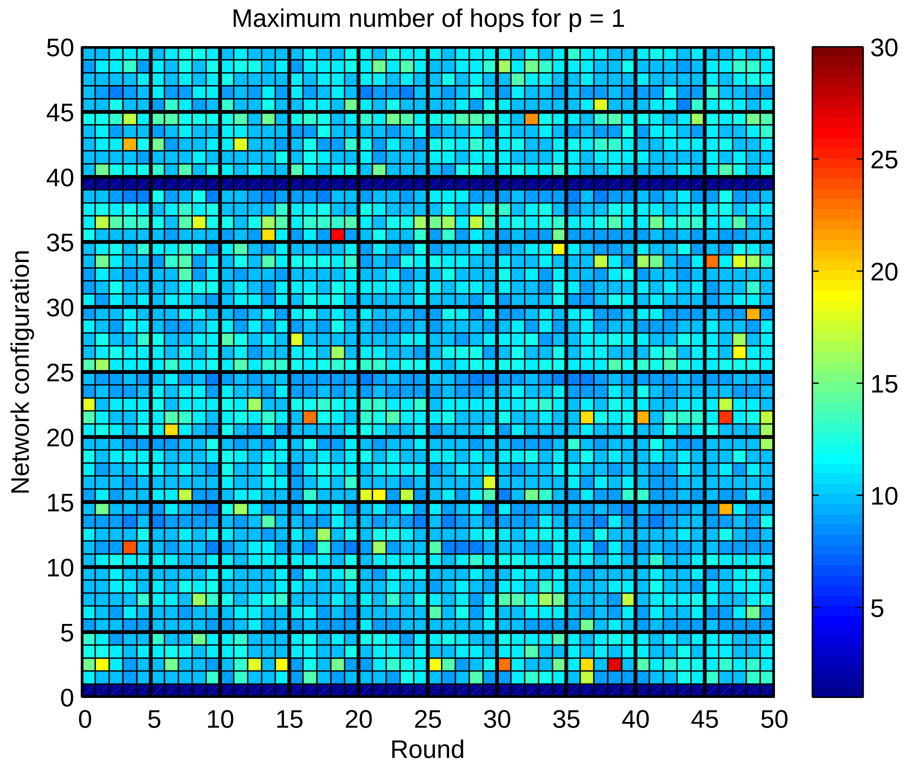


Figure 19: Maximum THL for all the network configurations and rounds and $p = 0.5$ and $p = 1$.

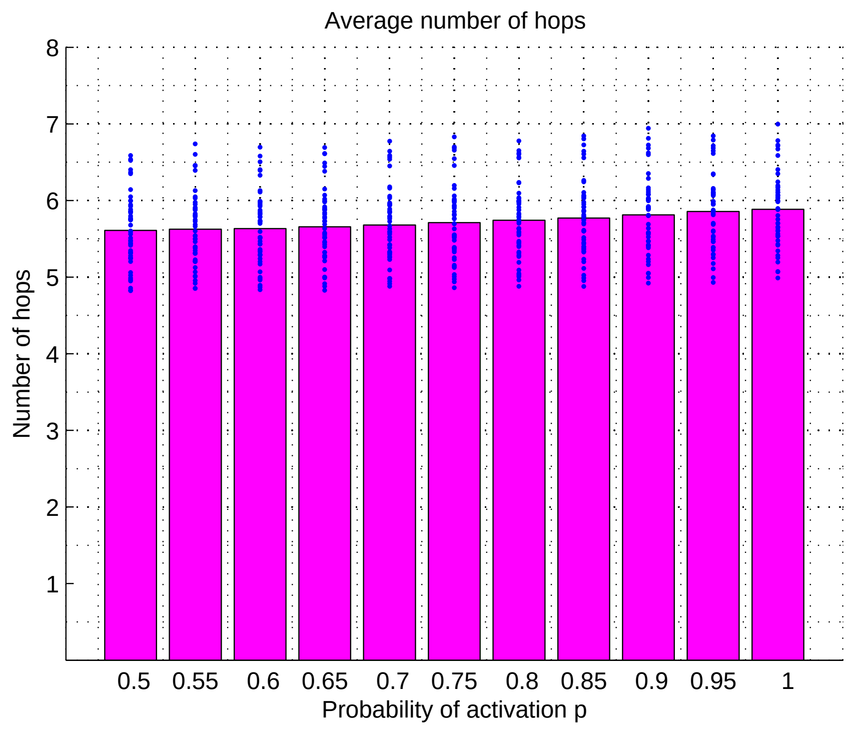


Figure 20: Average THL for all networks as p increases.

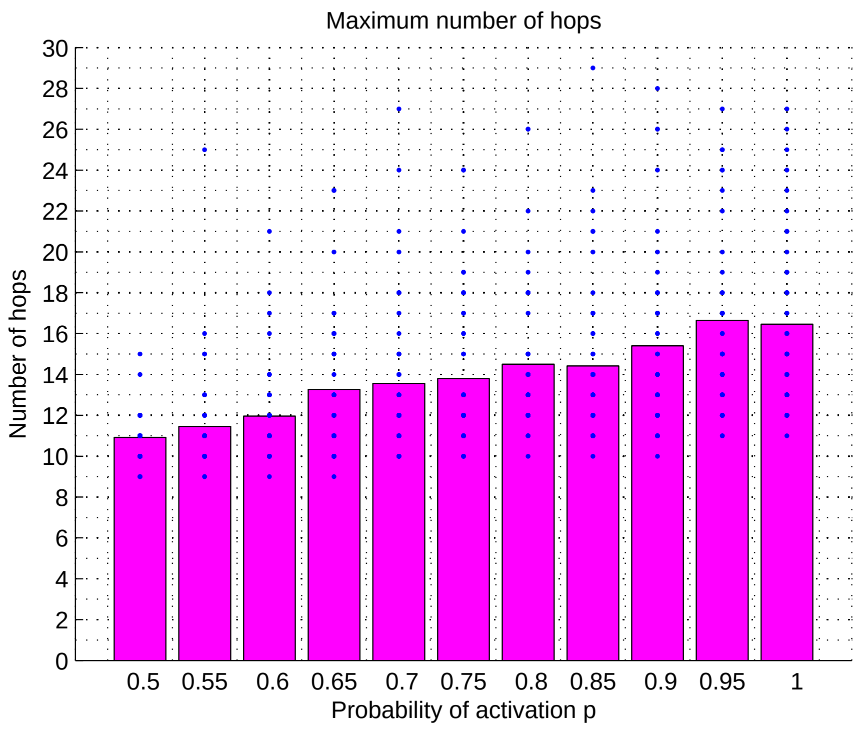
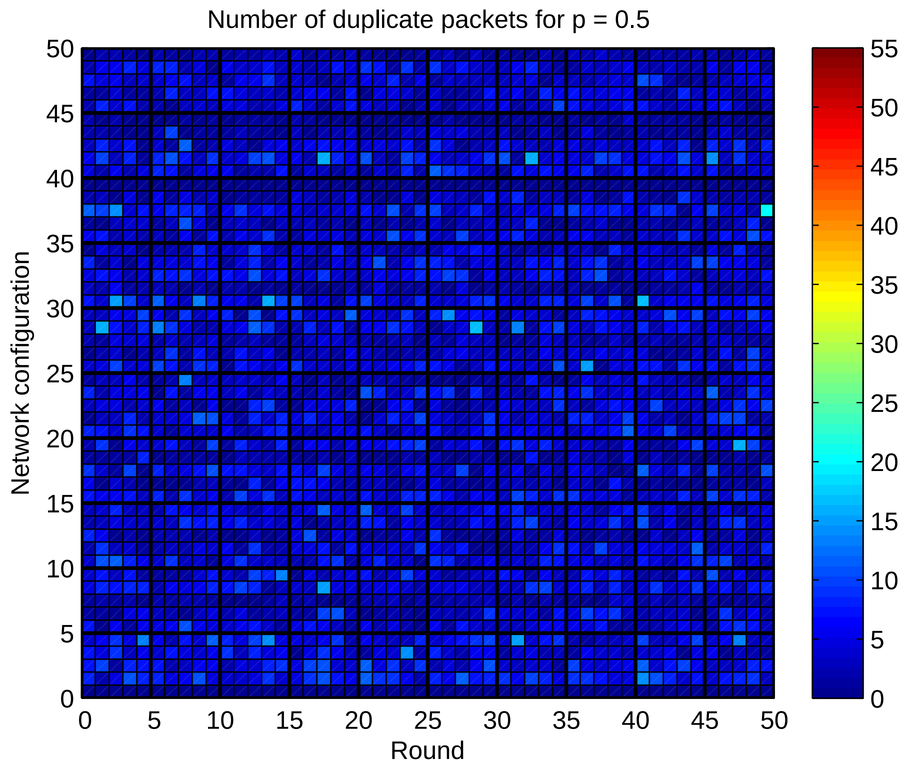


Figure 21: Maximum THL for all networks as p increases.



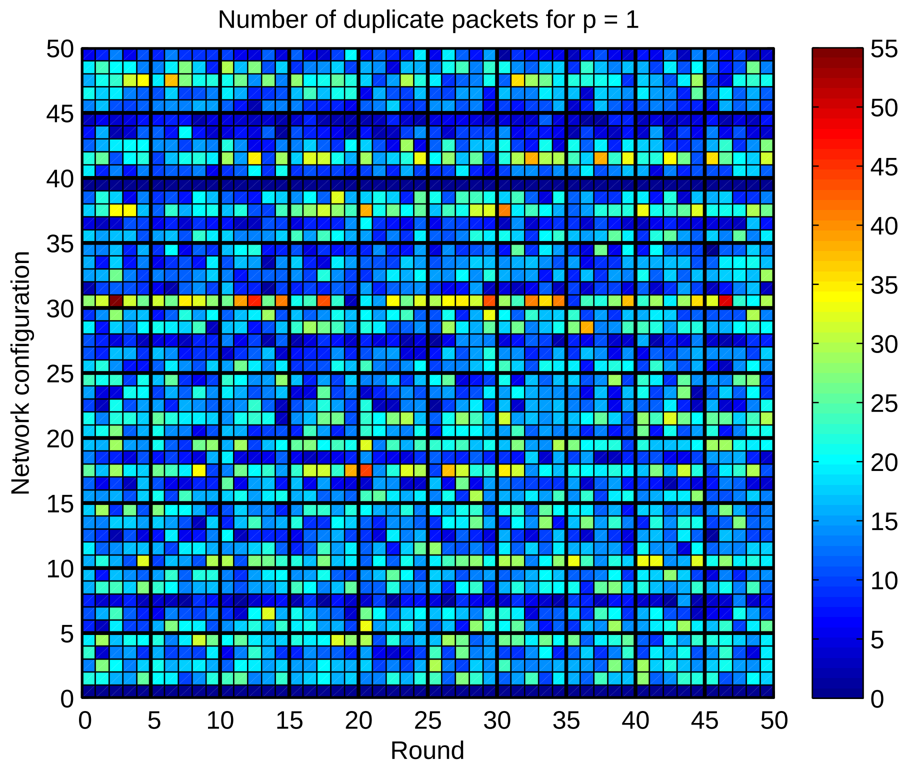


Figure 22: Number of duplicate packets for all the network configurations and rounds for $p = 0.5$ and $p = 1$.

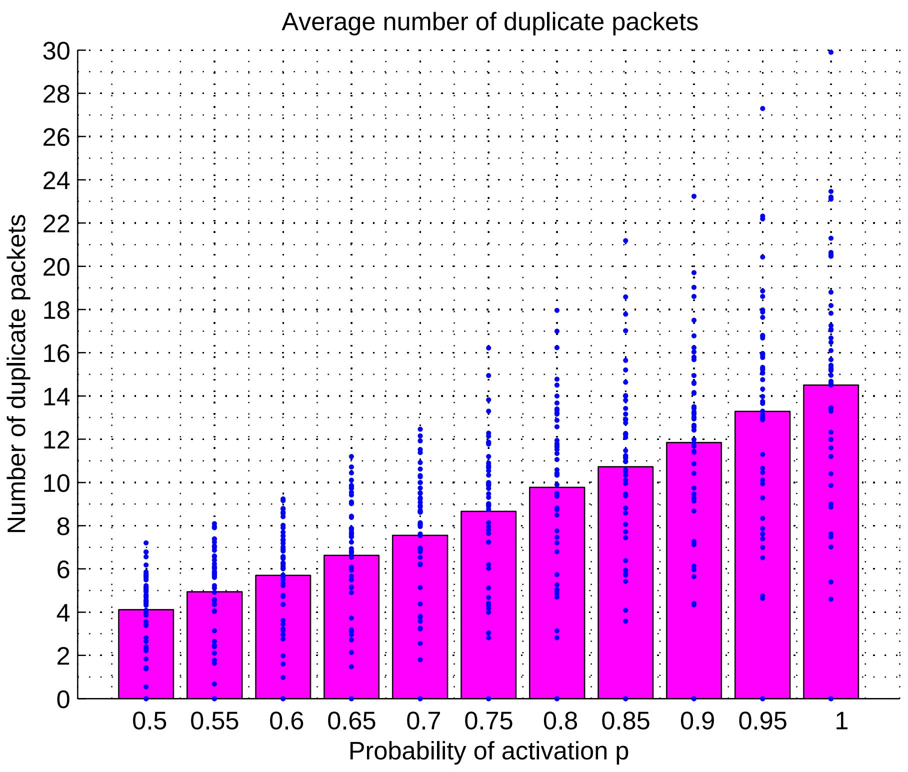


Figure 23: Average number of duplicate packets for all networks as p increases.