Two Topology Control Algorithms in the Wiselib: CBTC and LSP

Josep Anguera Peralta

Winter 2013
Dedicat a la Maria, per la paciència infinita.
A la Mariàngels, l’Elvira i la Tania, pel suport incondicional.
Al Josep Àngel i al Víctor, per Braunschweig.
A la Dafne, per fer que el meu anglès fagi menys vergonya.
I a tots aquells que en algun moment d’aquests anys m’han preguntat:
“Qué tal el projecte?”
## Contents

1 Topology Control .............................................................. 7
   1.1 Motivations for Topology Control ........................................... 7
      1.1.1 Topology Control and energy conservation ....................... 7
      1.1.2 Topology Control and network capacity .......................... 8
      1.1.3 Defining Topology Control ........................................... 10
   1.2 Distributed Topology Control ............................................ 11
      1.2.1 Ideal Features of a Topology Control Protocol .................... 12
      1.2.2 The Quality of Information .......................................... 13
   1.3 Dealing with node mobility ............................................... 13

2 The Wiselib ........................................................................... 15
   2.1 Introduction ........................................................................ 15
   2.2 Wiselib’s challenges .......................................................... 17
      2.2.1 Heterogeneity .............................................................. 17
      2.2.2 C++ in Embedded Systems .............................................. 19
   2.3 Implementation Details ....................................................... 20
      2.3.1 Architecture .............................................................. 20
      2.3.2 External Interface ........................................................ 20
      2.3.3 pSTL ........................................................................... 22
      2.3.4 Algorithm Support ....................................................... 22

3 Programming TC Algorithms in the Wiselib ............................. 23
   3.1 BASIC Topology algorithm ................................................... 23

4 The CBTC algorithm ............................................................. 28
   4.1 Introduction ......................................................................... 28
   4.2 The Cone Based Topology Control algorithm ............................. 29
      4.2.1 Asymmetric neighbours .................................................. 31
      4.2.2 $\alpha$ parameter: $5\pi/6$ vs $2\pi/3$ ................................... 32
   4.3 Optimizations ................................................................. 32
      4.3.1 The shrinkback operation ................................................. 32
      4.3.2 Asymmetric edge removal .............................................. 34
      4.3.3 Pairwise edge removal .................................................... 34
   4.4 Dealing with reconfiguration, asynchrony and failures ............... 36
      4.4.1 Neighbour Discovery Protocol ....................................... 36
      4.4.2 The reconfiguration algorithm ....................................... 37
      4.4.3 Beaconing Power and Optimizations ................................. 38
      4.4.4 Failures and other issues .............................................. 38
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>CBTC implementation</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>40</td>
</tr>
<tr>
<td>5.1.1</td>
<td>No Angle-of-Arrival (AoA) hardware</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>The Discovery phase</td>
<td>41</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Messages (in the Discovery Phase)</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Transition</td>
<td>44</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The check_gap test</td>
<td>44</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Optimizations</td>
<td>45</td>
</tr>
<tr>
<td>5.5</td>
<td>The Maintenance phase</td>
<td>49</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Messages (in the Maintenance Phase)</td>
<td>49</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Transition back to the Discovery Phase</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Parameters</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>The LSP algorithm</td>
<td>53</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>The LSP algorithm</td>
<td>53</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Link weight calculation</td>
<td>54</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Link weight information exchange</td>
<td>55</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Local topology construction</td>
<td>56</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Transmission power adjustment</td>
<td>58</td>
</tr>
<tr>
<td>7</td>
<td>LSP implementation</td>
<td>59</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Data Structures</td>
<td>60</td>
</tr>
<tr>
<td>7.2</td>
<td>The four steps</td>
<td>60</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Link weight calculation</td>
<td>60</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Link weight information exchange</td>
<td>62</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Local topology construction</td>
<td>63</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Transmission power adjustment</td>
<td>65</td>
</tr>
<tr>
<td>7.3</td>
<td>Parameters</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>Results</td>
<td>67</td>
</tr>
<tr>
<td>8.1</td>
<td>Scenarios</td>
<td>67</td>
</tr>
<tr>
<td>8.2</td>
<td>Measure method</td>
<td>67</td>
</tr>
<tr>
<td>8.3</td>
<td>Neighbours per node</td>
<td>68</td>
</tr>
<tr>
<td>8.4</td>
<td>Distances for every two pairs of nodes</td>
<td>69</td>
</tr>
<tr>
<td>8.5</td>
<td>Transmission Power</td>
<td>69</td>
</tr>
<tr>
<td>8.6</td>
<td>Traffic simulation</td>
<td>71</td>
</tr>
<tr>
<td>8.7</td>
<td>Number of messages and estimated cost</td>
<td>72</td>
</tr>
<tr>
<td>8.8</td>
<td>Comparison with the original papers</td>
<td>74</td>
</tr>
<tr>
<td>8.9</td>
<td>Time and memory usage</td>
<td>74</td>
</tr>
<tr>
<td>9</td>
<td>Project costs</td>
<td>75</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The case for multihop communication: node $u$ must send a packet to $v$, which is at distance $d$; using the intermediate node $w$ to relay $u$’s packet is preferable from the energy consumption’s point of view. ................................................. 8
1.2 Conflicting wireless transmissions. The circles represent the radio coverage area with transmission power $P$. ......................... 9
1.3 Conflicting wireless transmissions. The circles represent the radio coverage area with transmission power $P$. In the left part all the nodes are transmitting with the same power, while in the right part they diverge. ............................................. 10
2.1 Wiselib Architecture .................................................. 20
4.1 Angle of Arrival ...................................................... 28
4.2 Asymmetric neighbour ............................................... 31
4.3 Shrinkback ............................................................. 33
4.4 Pairwise ............................................................... 36
5.1 HELLO message ..................................................... 42
5.2 ACK message ......................................................... 43
5.3 ASYMMETRIC message .............................................. 43
5.4 NDP message ........................................................ 50
6.1 LSP Step 1 ............................................................ 54
6.2 LSP Step 3 ............................................................ 57
7.1 NMapping and NMatrix .............................................. 60
8.1 Neighbours per node ............................................... 68
8.2 Distances for every two pairs of nodes ......................... 69
8.3 Transmission power of nodes after algorithm execution .... 70
8.4 Traffic simulation cost .............................................. 71
8.5 Average number of messages sent per node ................ 72
8.6 Estimated cost of messages sent per node ................... 73
List of Tables

2.1 Evaluation of target platforms when the initial Wiselib design was made. The columns refer to the type of microcontroller, the standard operating system, the programming language for it, what kind of dynamic memory is available, the amount of ROM and RAM, and the bit width. ............................................. 17

2.2 Availability of C++ compilers for selected platforms ............................................. 19

6.1 The power in abstract units needed by every node to reach its direct neighbours. The bigger the value, the more expensive it will be to reach that neighbour ............................................. 55

6.2 Information received in step 2 of LSP by every node about neighbours of neighbours and actions taken on it ............................................. 56

6.3 The result of applying the Dijkstra’s algorithm in every node in the network ............................................. 58

7.1 An example of the output of Dijkstra’s algorithm regarding the array Parents and its meaning ............................................. 65

8.1 Comparison with the original papers ............................................. 74

B.1 Neighbours per node ............................................. 111

B.2 Distances for every two pair of nodes ............................................. 112

B.3 Transmission Power ............................................. 113

B.4 Traffic simulation ............................................. 114

B.5 Number of messages sent ............................................. 115

B.6 Estimated cost of messages sent ............................................. 116
Chapter 1

Topology Control

What is exactly topology control? Why is it needed? In this chapter, we state the two main reasons why topology control is preferable over a solution without it. Later, we will try to give a definition that clearly states what the boundaries of topology control are, or what is considered or what is not topology control. Finally we will talk about distributed topology control and how these algorithms have traditionally been dealt with node mobility.

1.1 Motivations for Topology Control

There are two fundamental motivations for topology control: energy conservation and network capacity.

1.1.1 Topology Control and energy conservation

The efficient use of the scarce energy resources available to Wireless Sensor Networks (WSN) is one of the fundamental tasks of the network designer. Since nodes consume a considerable amount of energy to transmit and/or receive messages, reducing the energy consumed for radio communications is an important issue.

In figure 1.1, we have the typical scenario in its simplest form in WSN for two nodes trying to communicate and a third one in the "middle". Suppose node $u$ must send a message to node $v$, which is at distance $d$. Node $v$ is within $u$’s transmitting range at maximum power, so direct communication between $u$ and $v$ is possible. However, there also exists a node $w$ in the region $C$ circumscribed by the circle of diameter $d$ that intersects both $u$ and $v$. Since $\delta(u,w) = d_1 < d$ and $\delta(v,w) = d_2 < d$, sending the message using $w$ as a relay is also possible. With these two possible alternatives, the interesting question regarding energy efficiency is: which one is more convenient from the energy-consumption point of view?

To answer this question, we must refer to a specific wireless channel and energy-consumption models. For simplicity, let’s assume the radio signal propagates according to the free space model and that we are interested in minimizing the transmission power only. With these assumptions, the power needed to send
CHAPTER 1. TOPOLOGY CONTROL

Figure 1.1: The case for multihop communication: node $u$ must send a packet to $v$, which is at distance $d$; using the intermediate node $w$ to relay $u$’s packet is preferable from the energy consumption’s point of view.

the message directly from $u$ to $v$ is proportional to $d^2$. So in case the message is relayed by node $w$, the total power consumption is proportional to $d_1^2 + d_2^2$.

Consider the triangle $uwv$, and let $\gamma$ be the angle opposite to side $uv$. By elementary geometry, we have

$$d^2 = d_1^2 + d_2^2 - 2d_1d_2 \cos \gamma.$$

Since $w \in C$ implies that $\cos \gamma \leq 0$, we have that $d_2 \geq d_1^2 + d_2^2$. It follows that, from the energy-consumption point of view, it is better to communicate using short, multihop paths between the sender and the receiver. And so this observation leads us to the first argument in favour of topology control: instead of using a long and energy-inefficient edge, communication can take place along a multihop path composed of short edges that connect the two endpoints of the long edge. The goal of topology control is to identify and somehow “eliminate” these energy-inefficient edges from the communication graph.

1.1.2 Topology Control and network capacity

Contrary to the case of wired point-to-point channels, wireless communications use a shared medium, the radio channel. The use of a shared communication medium implies that particular care must be paid to avoid that concurrent wireless transmissions corrupt each other.

A typical conflicting scenario is depicted in figure 1.2: node $u$ is transmitting a packet to node $v$ using a certain transmission power $P$. Since $\delta(v, w) = d_2 < \delta(v, u) = d_1$, the power of the interfering signal received by $v$ is higher than that of the intended transmission from $u$, and the reception of the packet sent by $u$ is corrupted.

Note that the amount of interference between concurrent transmissions is strictly related to the power used to transmit messages. We clarify this point with an example. Assume that node $u$ must send a message to node $v$, which is experiencing a certain interference level $I$ from other concurrent radio communications. For simplicity, we treat $I$ as a received power level, and we assume
CHAPTER 1. TOPOLOGY CONTROL

Figure 1.2: Conflicting wireless transmissions. The circles represent the radio coverage area with transmission power $P$.

that a packet sent to $v$ can be correctly received only if the intensity of the received signal is at least $(1 + \eta)I$, for some positive $\eta$. If the current transmission power $P$ used by $u$ is such that the received power at $v$ is below $(1 + \eta)I$, we can ensure correct message reception by increasing the transmission power to a certain value $P' > P$, such that the reception power at $v$ is above $(1 + \eta)I$. This seems to indicate that increasing the transmission power is a good choice to avoid packet drops due to interference. On the other hand, increasing the transmission power at $u$ increases the level of interference experienced by the other nodes in $u$'s surrounding. So, there is a trade-off between the ‘local view’ ($u$ sending a packet to $v$) and the ‘network view’ (reducing the interference level in the whole network): in the former case, a high transmission power is desirable, while in the latter case, the transmission power should be as low as possible. The following question then arises: how should the transmission power be set, if the designer’s goal is to maximize the network traffic capacity?

To answer that question, the required formal demonstration is not as straightforward as the one in the previous subsection, but in [9] a simple enough one is given. The conclusion of that demonstration is that, from the network capacity point of view, it is better to communicate using short, multihop paths between the sender and the destination.

Although not a formal proof, in figure 1.3 it is possible to intuitively see how the lack of a topology control affects the network capacity. In the left part of the picture, it is illustrated how this specific network would look in the absence of a topology control. All the nodes in the network are transmitting at its maximum power. The red circle around every node (the black points) delimits the range of the node and so its interference region. On the other hand, in the right part of the picture, some kind of topology control has been applied (that preserves connectivity).

To further ease to illustrate what is happening, the more opaque red means that more signals overlap. So it can be seen that, without doing any kind of topology control, when one node transmits, almost every other node in the
network cannot do so.

Figure 1.3: Conflicting wireless transmissions. The circles represent the radio coverage area with transmission power $P$. In the left part all the nodes are transmitting with the same power, while in the right part they diverge.

1.1.3 Defining Topology Control

In the previous section, we have presented at least two arguments in favour of a careful control of the network topology: reducing energy consumption and increasing network capacity. Although we have used “topology control”, a clear definition of it has not been introduced yet.

There is no such thing as a standard or formal definition regarding “topology control”. However, all the algorithms self defined as TC ones share some characteristics: they gather and use some kind of information to modify in some way the original network graph (the one resulting from all the nodes in the network transmitting at its highest power), in order to achieve some property (generally reducing the energy consumption and/or increasing the network performance). Nevertheless, this collection of characteristics is too much vague to only group what is known as TC algorithms. A better definition that tries to really establish the correct boundaries of TC is presented in [9]. It states: “quite informally, topology control is the art of coordinating nodes’ decisions regarding their transmitting ranges, in order to generate a network with the desired properties (e.g. connectivity) while reducing node energy consumption and/or increasing network capacity”. They later go on to add that, while this definition is quite general, they believe it captures the very distinguishing feature of topology control with respect to other techniques used to save energy and/or increase network capacity: the networkwide perspective. Or in other words, nodes making local choices (setting the transmission level) with the goal of achieving a certain global, networkwide property.

This definition is good in the sense that it includes all of what is today considered topology control and at the same time rules out things that, while sharing some characteristics to TC algorithms they are not considered as such. For example, an energy-efficient design of the wireless transceiver cannot be
classified as topology control because it has a nodewide perspective. The same applies to power-control techniques, whose goal is to optimize the choice of the transmission power level for a single wireless transmission, possibly along several hops; in this case, we have a channelwide perspective.

The aforementioned definition excludes other techniques that sometimes are referred to as topology control. For example, several authors consider as topology control techniques also mechanisms used to superimpose a network structure on an otherwise flat network organization. This is the case, for instance, of clustering algorithms, which organize the network into a set of clusters, which are used to ease the task of routing messages between nodes and/or to better balance the energy consumption in the network. In a typically clustering protocol, a distributed leader election algorithm is executed in each cluster, and cluster nodes elect one of them as the clusterhead. Message routing is then performed on the basis of a two-level hierarchy: the message originating at a cluster node is destined to the clusterhead, which decides whether to forward the message to another clusterhead (intercluster communication) or to deliver the message directly to the destination (intercluster communication).

Although clustering protocols can be seen as a means of controlling the topology of the network by organizing its nodes into a multilevel hierarchy, a clustering algorithm does not fulfill the previous given definition of topology control since typically the transmit power of the nodes is not modified. In other words, a clustering algorithm is concerned with hierarchically organizing the network units assuming the nodes’ transmitting range is fixed, while a topology control protocol is concerned with how to modify the nodes’ transmitting range in such a way that a communication graph with certain properties is generated.

One last minor thing about this definition is that it makes a strong emphasis on the modification of the transmission range. However, not in all topology control literature is this assumed as an obligatory step. For example, in [11], they refer to that step as an optional one. That is, all the steps of the algorithm are performed to determine the logical neighbours subset of a node, and the final step, and that would be to adjust the transmission range to the minimum to reach these neighbour nodes, is described as an optional step.

1.2 Distributed Topology Control

There are many classifications that can be made regarding topology control. One such classification is centralized vs distributed. In centralized topology control algorithms, it is typically assumed that all information regarding node position is available to a central entity, which uses this information to compute the ‘optimal’ topology. In this approach to topology control, the emphasis is on the properties of the generated topology, rather than on the process needed to build such topology. Given that no such central entity is present, by definition, in ad hoc and sensor networks (it is not that such a central authority cannot coexist in a sensor network, but let’s say it is not expected or it should not be), centralized approaches are doomed to perform poorly in realistic application scenarios. So distributed topology control is a more practical approach to the topology control problem, where the challenge is to design lightweight, fully distributed protocols that generate a ‘reasonably good’ topology. Here, the focus is then on the topology generation process, rather than on the ‘quality’ of
the resulting topology.

Both algorithms described in this document are distributed topology control ones.

1.2.1 Ideal Features of a Topology Control Protocol

Which are the features that a topology control protocol should ideally have?

There are several features that can be pointed out regarding topology control, but five outstanding features are listed below (what we consider the most important), not in any specific order:

- be fully distributed and asynchronous;
- be localized;
- generate a topology that preserves the original network connectivity and relies on bidirectional links only;
- generate a topology with a small physical node degree;
- rely on ‘low-quality’ information.

We have already explained why only solutions that can be implemented in a fully distributed and asynchronous fashion have some practical relevance. By asynchronous, we refer to the lack of a central clock to synchronize the actions on the nodes.

The second feature, locality, refers to the nodes’ ability to build their view of the network topology by using local information only; that is, information regarding only their 1-hop neighbours. Localized solutions have several advantages with respect to approaches that require a networkwide information exchange: since network nodes can build their local view of the topology by exchanging a few messages with neighbour nodes only, localized protocols can be classified as lightweight solutions, which can be implemented in very large networks also; furthermore, locality implies that the network topology can be easily reconfigured when nodes leave/join the network, or in the presence of node mobility.

As mentioned above, the goal of a distributed topology control protocol is to build a ‘reasonably good’ topology. But what do we mean by ‘reasonably good’? What are the features that the generated topology should have? Well, the answer is point 3 of the features presented above: the topology generated by the topology control protocol should rely on bidirectional links only. Furthermore, it is desirable that the topology control algorithm preserves connectivity; in other words, if the network is connected when all nodes communicate with maximum transmission power (i.e. if the maxpower graph is connected), then it should preserve this property also after every node in the network has executed the topology control protocol. So, only redundant links should be removed from the network topology. The reasons behind bidirectional links are discussed in § but what is basically said is that, while unidirectional wireless links networks are technically feasible, the advantages of using unidirectional links is questionable. For instance, in case of routing protocols, the high overhead needed to handle unidirectional links outweighs the benefits that they can provide, and a better performance can be achieved by simply avoiding them.
CHAPTER 1. TOPOLOGY CONTROL

Regarding point four, building a network topology in which nodes have a small physical degree is highly desirable, since this parameter is a measure of the interference generated by a transmitting node: if the physical degree of node $u$ is small, the number of nodes impacted by $u$’s transmission is relatively small, and spatial reuse is increased.

Finally, an aspect to consider in the design of a topology control mechanism is the quality of information required by the algorithm: since obtaining very accurate information such as node locations is, in general, quite expensive (in terms of additional hardware required on the nodes, or message overhead, or both), it is desirable that the protocol relies on ‘low-quality’ information. This issue is discussed deeper in next section.

1.2.2 The Quality of Information

An important aspect to be considered in the design of topology control algorithms is the type of information used by the nodes to build the local view of the topology: nodes can use high-quality information (e.g. neighbour node locations), medium-quality information (e.g. directional information, or distance to neighbours), or low-quality information (number and identity of neighbour nodes). In general, there is a direct relationship between information quality and energy efficiency of the computed topology: the more accurate the information available to the nodes, the more energy savings can be achieved. However, information quality (and, thus, energy savings) must be carefully traded off with the cost incurred for making the information available to the nodes. The cost is due to either some additional hardware required on the nodes (e.g. low-power GPS receivers in case of location information) or the message overhead needed to produce/update high-quality information, or both.

Another argument in favour of designing protocols based on low-quality information is that they can be used in a wider range of application scenarios. For instance, it is well known that location estimation techniques perform poorly in indoor environments because of the hardly predictable propagation of the radio signal.

1.3 Dealing with node mobility

One of the prominent features of wireless sensor networks is node mobility. It’s not that it is mandatory to labeled it as such, but in many application scenarios, the wireless devices that form the network, or at least a significant part of them, are mobile. Even in a network where the nodes are stationary, nodes may die if they run out of energy. In addition, new nodes may be added to the network.

Traditionally, topology control algorithms have not taken into account that special behaviour, at least not directly. The solution given to that behaviour is to simply rerun the algorithm periodically. The obvious advantage of this approach is of course its simplicity; that is, no changes are made to the algorithms. On the other side, the problem is that no effort is made to detect and/or prevent any change to the topology so, if one such changes is produced just after one of the executions, this change will not be noticed by the other nodes until the next execution. Now you have two problems, because you have to decide which frequency is used to run the algorithm. If it is too low (that is, more spatiated in
time), the network will start to misbehave. That is because some links between nodes no longer exist, but the nodes think they do. And being topology control in such a low level in the protocol stack that the other levels rely on, the impact is critical. On the other hand, if the frequency is too high, the network will be flooded by the messages of the topology control protocol, instead of the actual data and, additionally, it will have a serious impact on the energy of the nodes, going against one of the purposes of topology control.

However, that approach is good enough in networks where node mobility is almost inexistent. For example, in networks where nodes do not move (or do it rarely and/or in an insignificant distance) and their source of energy is guaranteed (or almost). In this kind of networks, it is perfectly feasible to use that approach.

One notable exception is the CBTC algorithm (one of the two algorithms this document is about). The mechanism employed by this algorithm is explained in detail in chapters 4 and 5. Essentially, what it does is to start another protocol once the main protocol is done, where the main protocol would be analogous to the whole process of the regular topology control algorithms. This second protocol basically issues messages periodically so that the neighbour nodes know that it is still alive. In turn, the neighbours detect changes in the emission of these presence messages and are able to know the situation with more or less accuracy. In some way, this is similar to the traditional approach; the difference is that the CBTC approach is more lightweight. It is so, because not the whole topology control algorithm has to be executed and because it is executed only when changes are detected.

Although this subalgorithm of CBTC is slightly coupled with the main topology control algorithm, it could be reused for other topology control algorithms that do not have a specific mechanism of this kind. Actually, topology construction and topology maintenance sometimes are considered two different subproblems and effectively algorithms specifically tailored to that problem exist.
Chapter 2

The Wiselib

“One unfortunate consequence of the success story of Wireless Sensor Networks (WSN) in separate research communities is an ever-growing gap between theory and practice. Even though there is an increasing number of algorithmic methods for WSNs, the vast majority has never been tried in practice; conversely, many practical challenges are still awaiting efficient algorithmic solutions. The main cause for this discrepancy is the fact that programming sensor nodes still happens at a very technical level. To remedy the situation Wiselib is conceived, an algorithm library that allows for simple implementations of algorithms onto a large variety of hardware and software. This is achieved by employing advanced C++ techniques such as templates and inline functions, allowing to write generic code that is resolved and bound at compile time, resulting in virtually no memory or computation overhead at runtime.”[2]

2.1 Introduction

Since the initial visions proposed in the SmartDust[6] project several years ago, WSN have seen a tremendous development, both in theory and in practice. On the practical side, we see working sensor networks and applications in many areas, from academia to industrial appliances. There is a large variety of hardware and software to choose from that is easy to set up and use.

This success story has also led to a serious practical issue that has not been sufficiently addressed in the past: Sensor node brands are very different in their capabilities. Some nodes have 8-bit microprocessors and tiny amounts of RAM, while others burst with power, being able to run desktop operating systems such as Linux. Consequently, the software running on these systems is very different on the various nodes. While it is easy to write code for a specific platform, it is a very challenging task to develop platform-independent code. Even worse, the operating systems on most sensor nodes provide barely enough functionality to implement simple algorithms. This means that the developer is forced to spend great attention on low-level details, making the process painfully complex and slow.

A parallel success story can be observed on the theoretical side, where the
CHAPTER 2. THE WISELIB

development of distributed algorithms for many actual or hypothetical problems has grown into a research field of its own. This has led to a large variety of highly sophisticated algorithms for all kinds of tasks. Unfortunately, many of them have never been tried in practice, due to the overly difficult implementation process. Where algorithms are implemented, they are hard to share and compare, as implementations cannot be easily ported to new platforms. Moreover, many important challenges are not even addressed, as they can only be identified and resolved by close collaboration between theory and practice.

This growing gap between theory and practice forms a major impediment for exploiting the possibilities of complex distributed systems. The Wiselib is an attempt to remedy this unfortunate situation. Wiselib is a framework, written in C++, for platform-independent algorithm development. Each algorithm written for the Wiselib can be compiled for any supported system without changing any line of code. It provides simple interfaces to the algorithm developer, with a unified API and ready-to-use data structure implementations. The Wiselib addresses the following issues:

- **Platform independence**: Wiselib code can be compiled on a number of different hardware platforms, usually without platform-dependent configurations, i.e., no “#ifdef” constructions. See subsection 2.2.1 for details.

- **OS independence**: Wiselib code can be compiled for different operating systems. This includes systems based on C like Contiki\[^3\], as well as C++ (the iSense firmware \[^5\]) and nesC \[^8\] (TinyOS \[^14\]).

- **Exchangeability**: Algorithms and applications can be composed of different components that interact using well-defined interfaces, called concepts. Components can be exchanged with other implementations without affecting the remaining code. Moreover, both generic components and highly optimized platform-specific components can be used simultaneously.

- **Broad algorithm coverage**: The Wiselib currently covers a large variety of algorithms. It will contain algorithms for each of the following categories:

  1. Routing algorithms
  2. Clustering algorithms
  3. Time-synchronization algorithms
  4. Localization algorithms
  5. Topology algorithms
  6. Data dissemination
  7. Target tracking
  8. ...

- **Cross-layer algorithms**: In Wiselib an algorithm can be designed to use other algorithm concepts, thus enabling the use of existing algorithms for the implementation of more complex ones. Moreover, we can stack protocols on top of each other, extending their functionality.
• **Standard compliance**: The library is written in a well-defined language subset of ISO C++. This has a number of benefits over custom languages such as nesC: the compilers are more mature and better supported, and there is a large user base that knows C++ from desktop development.

• **Scalability and efficiency**: The Wiselib is capable of running on a great variety of hardware platforms, with CPUs ranging from 8-bit microcontrollers to 32-bit RISC CPUs, and with memory ranging from a few kilobytes to several megabytes. Algorithms need to be very resource-friendly on the platforms from the lower end, and at the same time be able to use more resources if available.

### 2.2 Wiselib’s challenges

#### 2.2.1 Heterogeneity

When developing an algorithm library for sensor networks, one must deal with a great variety of different hardware and software platforms. Table 2.1 shows an overview of platforms that were taken into account for the development of the Wiselib.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Firmware/OS</th>
<th>CPU</th>
<th>Language</th>
<th>Dyn Mem</th>
<th>ROM</th>
<th>RAM</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSense</td>
<td>iSense-FW</td>
<td>Jennie</td>
<td>C++</td>
<td>Physical</td>
<td>128kB</td>
<td>92kB</td>
<td>32</td>
</tr>
<tr>
<td>ScatterWeb</td>
<td>MSB SCW-FW</td>
<td>MSP430</td>
<td>C</td>
<td>None</td>
<td>48kB</td>
<td>10kB</td>
<td>16</td>
</tr>
<tr>
<td>ScatterWeb</td>
<td>MSB SCW-FW</td>
<td>MSP430</td>
<td>C</td>
<td>None</td>
<td>60kB</td>
<td>2kB</td>
<td>16</td>
</tr>
<tr>
<td>Tmote Sky</td>
<td>Contiki</td>
<td>MSP430</td>
<td>C</td>
<td>Physical</td>
<td>48kB</td>
<td>10kB</td>
<td>16</td>
</tr>
<tr>
<td>MicaZ</td>
<td>Contiki</td>
<td>ATMega128L</td>
<td>C</td>
<td>Physical</td>
<td>128kB</td>
<td>4kB</td>
<td>8</td>
</tr>
<tr>
<td>TNOde</td>
<td>TinyOS</td>
<td>ATMega128L</td>
<td>nesC</td>
<td>Physical</td>
<td>128kB</td>
<td>4kB</td>
<td>8</td>
</tr>
<tr>
<td>Mote2</td>
<td>TinyOS</td>
<td>Intel XScale</td>
<td>nesC</td>
<td>Physical</td>
<td>32MB</td>
<td>32MB</td>
<td>32</td>
</tr>
<tr>
<td>GumStix</td>
<td>Emb. Linux</td>
<td>Intel XScale</td>
<td>C</td>
<td>Virtual</td>
<td>16MB</td>
<td>64MB</td>
<td>32</td>
</tr>
<tr>
<td>Desktop PC</td>
<td>Shawn</td>
<td>various</td>
<td>C++</td>
<td>Virtual</td>
<td>unlimited</td>
<td>unlimited</td>
<td>32/64</td>
</tr>
<tr>
<td>Desktop PC</td>
<td>TOSSIM</td>
<td>(ATMega128L)</td>
<td>nesC</td>
<td>(Physical)</td>
<td>unlimited</td>
<td>unlimited</td>
<td>(8)</td>
</tr>
</tbody>
</table>

Table 2.1: Evaluation of target platforms when the initial Wiselib design was made.

The columns refer to the type of microcontroller, the standard operating system, the programming language for it, what kind of dynamic memory is available, the amount of ROM and RAM, and the bit width.

The operating systems vary from system-specific implementations such as iSense and ScatterWeb to generic approaches such as Contiki, TinyOS, and Linux. The preferred programming languages vary with the OSs. The iSense firmware has been developed in C++, whereas the ScatterWeb firmware uses plain C. TinyOS uses a custom language, the C extension nesC. Support for dynamic memory, `malloc()` and `free()`, is only available for some systems. Using the ScatterWeb firmware, the size of all memory blocks must be known at compile time, whereas the iSense firmware provides a full implementation for the C++ operators `new` and `delete`. This is done with the aid of an own memory allocation implementation. Similar approaches are provided by TinyOS via `TinyAlloc`, and Contiki via the `managed memory allocator` or `memb block memory allocator`. Only the Linux-based node supports virtual address space for processes. There are also significant differences in the amount of available memory, ranging from a few kilobytes to 64 MByte in the GumStix. Finally, we must also deal with different bit widths. The Atmel Atmegas are 8-bit microcontrollers, the MSP430 are 16-bit microcontrollers, whereas the rest are 32-bit.
microcontrollers. There are a number of challenges stemming from the nodes’ properties and capabilities. These became additional library requirements.

**Limited Memory**

The algorithms may run on tiny microcontrollers for which the provided memory is very limited. On one hand, this affects the ROM. The generated code for an algorithm must be as small as possible to fit into memory. On the other hand, the RAM is affected. Routing tables, for example, cannot be arbitrarily long so as not to exhaust the limited main memory. Additionally, the node representation that is used for storing the neighbourhood must be as small as possible, but must also meet the demands of the used algorithms. At the same time, when running on a node with plenty of memory, performance gains can and should be achieved by employing more advanced data structures.

**Physical Dynamic Memory**

The availability of dynamic memory allocation is already a big step forward, allowing for efficient data structures. However, most implementations only provide physical addresses, and some are even unable to join adjacent freed memory blocks. Shifting of pages to join free blocks is impossible on all nodes with physical memory. Even a simple vector implementation with $O(\log n)$ amortized insertion time would leave behind a trail of $O(\log n)$ free blocks of various sizes. Therefore, data structures are carefully re-analyzed to take these special considerations into account.

**Limited Computation Power**

Because algorithms may run on small microcontrollers, efficiency plays an essential role. Examples are message reception in an interrupt or iterating over a neighbour table to select the next routing node. This also constrains the Wiselib not to enforce the use of slow operations (such as excessive pointer indirection) through the provided framework.

**Compiler Variance**

Wiselib must run on multiple hardware platforms. Different compiler versions must be supported, so it is important that only standard features of the selected programming language are used.

**Data Access**

When accessing data at arbitrary locations in memory, alignment problems can occur. For example, a cast of a 16bit integer works for both MSP430 and Jennic, when it starts at an even address. But when it starts at an odd address, it fails on both platforms. However, a cast of a 32bit integer works on all even addresses on a MSP430, but for Jennic only on quad-byte boundaries.

Moreover, when exchanging data in heterogeneous systems, the byte order must be taken into account, because some systems are big endian, whereas others are little endian.
### 2.2.2 C++ in Embedded Systems

The Wiselib must cover all of the previously mentioned hardware and software platforms; the latter are developed in different programming languages. Hence, an appropriate programming language must be picked. The language chosen was C++ [12], because it combines modern programming techniques with the ability of writing efficient and performant software. The use of C++ in embedded systems has already been evaluated [4].

C++ allows modern OO designs. Object-Oriented programming is standard on the desktop for quite some time by now, and has proven to ease the development of complex systems. Moreover, C++ is a fully typesafe language. This speeds up the development process, as it catches type errors at compile time. Given the tediousness of debugging on sensor nodes, this is a huge achievement.

The most important language feature for the Wiselib are templates [15]. Templates can be used to develop very efficient and flexible applications. The basic functionality of templates is to allow the use of generic code that is fully resolved by the compiler when specific types are given. Thereby, only the code that is actually needed is generated, and methods and parameters as template parameter can be accessed directly. Wiselib use the well-established technique of template-based “concepts” and “models”, where the former are not specified as actual code, but rather as formal specifications in documentation. It lists the required and provided types, as well as member function signatures. Models are implementations of concepts, using template specializations, without any inherent runtime overhead. Both concepts and models allow for polymorphism, including multiple inheritance. The Wiselib employs these methods using standard compiler features without custom additions.

Another basic feature in C++ is virtual inheritance. When declaring a method as virtual, the compiler has to generate a vtable consisting of function pointers to the appropriate methods. Whenever such a method is called, it has to be looked up in the vtable first, thereby requiring pointer indirection. This leads to an increase of both program memory and runtime, and makes some compiler optimizations impossible. Hence, virtual inheritance is not used in the Wiselib. In Wiselib this feature is substituted by templates.

Two more features that are not used in the Wiselib are runtime type information (RTTI) and exceptions. Both result in significant runtime and code-size overhead, as already shown in [4].

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Compiler</th>
<th>Binary</th>
<th>Base</th>
<th>libstdc++</th>
<th>Basic C++ Syntax</th>
<th>Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jennic</td>
<td>baelf-g++</td>
<td>√</td>
<td>GCC 4.2.1</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>MSP430</td>
<td>msp430-g++</td>
<td>-</td>
<td>GCC 3.2.3</td>
<td>-</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>ATMega128L</td>
<td>avr-g++</td>
<td>-</td>
<td>GCC 4.1.2</td>
<td>-</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Intel XScale</td>
<td>xscale-g++</td>
<td>√</td>
<td>GCC 3.3.1</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 2.2: Availability of C++ compilers for selected platforms

There are C++ compilers available for all of the target platforms. See Table 2.2 for an overview. Some platforms lack support for libstd++, which includes the operators new and delete. The STL is also not available everywhere. All compiler support the C++ features we build upon, i.e. template and member specializations.

All compilers are based on GCC, and thus there are no considered drawbacks from compiler incompatibilities. There are some minor limitations due to the
CHAPTER 2. THE WISELIB

missing libstdc++ on some systems, which have no impact on the Wiselib.

2.3 Implementation Details

The core design pattern for the Wiselib are generic programming techniques that are implemented using C++ templates. The basic idea is to pass the important functionality as template parameters to an algorithm: implementations of OS specific code, and data structures. Hence, it is possible to compile an algorithm exactly for the current needs.

2.3.1 Architecture

The fundamental design principle of the Wiselib consists of concepts and models, which have already been discussed in Subsection 2.2.2. Wiselib features an architecture with three main pieces: algorithms, OS facets, and data structures. The idea is shown in Figure 2.1.

![](image)

Figure 2.1: Wiselib Architecture

First of all, there are concepts for algorithms. There is one concept per category, whereby a category groups algorithms by their basic functionality, e.g. routing or localization. Any algorithm model implements one or multiple concepts, and is basically a template expecting various parameters. These parameters can be both OS facets and data structures.

OS facets represent the connection to the underlying operating system or firmware—for example, concepts for a radio or timer interface. Thus, the facets provide a lightweight abstraction layer to the OS. Note that the facets are merely type definitions and wrapper functions, they are supposed to contain no replication of OS functionality.

With the aid of data structures, an algorithm can scale to the platform it is compiled for. For instance, static data structures can be passed on tiny platforms without dynamic memory management, whereas highly dynamic and efficient data structures are passed on powerful microcontrollers or desktop PCs.

2.3.2 External Interface

The “external interface”, consisting of OS facets, represents the connection to the underlying OS. Implementations of these facets are passed to an algorithm
CHAPTER 2. THE WISELIB

as template arguments. The compiler should mostly be able to directly resolve such calls to the OS. For example, when registering a timer can be done using one line of code, it is implemented as an inline function in the appropriate timer model. Hence, the result would be a direct call to the OS function, and thus there would be no overhead, neither in code size nor in execution time. In C-based operating systems (we see TinyOS in this group), the OS facets have to provide a translation between C++ member function calls and C function calls, and they have to convert C++ members to C callback pointers. This is where an actual price of generality has to be paid. Fortunately this price is very low.

Several models of the same concept for an OS facet can also be made available, each with its own advantages for special purposes. The user can pass the best available model to an algorithm at compile time, without extra overhead.

An example for a model of the OS facet “radio” is as follows. It is for the C++-based iSense firmware:

```cpp
template<...> class iSenseRadioModel {
  static int send( Os *os, id_t id, size_t len, data_t *data)
  { os->radio().send( id, len, data, 0, 0); }
}
```

The example shows the implementation of a simple send method offered by a radio model. Since it is only one function call, it can be directly resolved by the compiler without generating any overhead.

**Concept Inheritance**

The above example of the radio’s `send()` method with destination address and payload is defined in the basic radio concept. Routing algorithms, for example, which only need to send and receive messages without any further information such as RSSI values, or requirements such as reliable delivery, can use the implementations of this concept.

We also allow for concept inheritance, so that the basic radio concept can easily be extended. If an algorithm needs access to RSSI (or LQI) values, a derived concept can be used. It extends the basic one with a receive method that provides additional values.

**Stackability**

A major design aspect for the radio concept is stackability, i.e. the possibility to build a layered structure of multiple radios. The topmost layer is not aware of which and how many layers it is connected to. The big advantage of this approach is that we can build a “virtual radio” that runs on top of a radio model, and is passed to an algorithm in its radio template parameter. Doing so, we can easily implement an algorithm for heterogeneous sensor networks. It is even possible to communicate between nodes that use different kinds of node IDs — because the virtual radio hides the real node addresses and provides virtual ones, e.g. generic 128 bit addresses.

Another possibility is to hide a complete routing algorithm behind an OS facet. For example, when writing out debug messages, this happens generally to the UART. But by passing another model, we can forward debug messages over a routing algorithm to a gateway, where all these messages are collected. The topmost algorithm does not need to be aware of the model it works on — it must only use the appropriate concept.
CHAPTER 2. THE WISELIB

Message Delivery in Heterogeneous Systems

Another problem that is addressed using this software design is message delivery in heterogeneous networks. There are basically two problems that occur: different byte-order and differences in alignment handling. Byte order issues are solved by sticking to network byte order in messages; while alignment is addressed via template specialization. Wiselib provides a serialization class that provides generic `read` and `write` methods for all data types.

2.3.3 pSTL

Not all of our target systems provide dynamic memory allocation. To our knowledge, no variant of the STL fulfills Wiselib’s requirements: not using `libstdc++`, `new/delete`, exceptions, and RTTI.

Consequently, Wiselib provides the pSTL, an implementation of parts of the STL that does neither use dynamic memory allocation nor exceptions nor RTTI. It is ensured that each of the provided data structures works on each supported hardware platform.

2.3.4 Algorithm Support

The central piece of the Wiselib are the algorithms. They are grouped into categories, see Section 2.1. Algorithm implementations can belong to several categories, which is common for cross-layer algorithms.

Each algorithm class consists of a concept for the algorithm itself, and some concepts for the data structures that are typically necessary for this class. This decouples the algorithm logic, which is invariant over different platforms, from data storage, which heavily changes when an algorithm is ported to a platform of different characteristics.

The benefit of having a well-defined algorithm interface is that algorithms are easily interchanged for testing purposes. Ideally this is done by simply altering a class name in the initialization code. The second -much more important- benefit is that an algorithm developer can start coding by copy-and-paste, instead of having to go through a design phase. Such a design phase can be quite lengthy, if the goal is to achieve maximal portability. Until now, theoreticians wishing to evaluate high-level algorithms often found it hard to develop for embedded devices: this lowers the bar considerably.

Providing a diverse set of data structure implementations serves the goal of scalability: for each data structure, e.g. routing tables, neighbourhood cluster maps, and position maps, a set of implementations matching the span of platforms is provided. For low-end architectures such as the MSP430, structures are needed that use static storage whose size is known at compile-time. Such structures will inevitably be inefficient in terms of runtime. For high-end architectures using Xscale processors or simulation environments, highly optimized data structures with dynamic memory management and huge memory overhead can be employed, resulting in high efficiency. It is even feasible to utilize the STL. The choice of data structures has no impact on the algorithm code, and can simply be configured at algorithm initialization. This results in algorithms that not only scale down to very limited devices, but also scale up to powerful nodes, utilizing all the available resources on them.
Chapter 3

Programming Topology Control Algorithms in the Wiselib

3.1 BASIC Topology algorithm

To illustrate how a topology algorithm is implemented in the Wiselib, following is a very basic and rudimentary topology algorithm. What it does is to broadcast a message a given number of times. After that, the node considers to be its neighbours all the nodes from which it has received a message.

```cpp
#include "algorithms/topology/topology_control_base.h"
#include "util/pstl/vector_static.h"

namespace wiselib
{

  template<typename OsModel_P, uint16_t MAX_NODES = 32>
  class BasicTopology: public TopologyBase<OsModel_P>
  {

    typedef OsModel_P OsModel;
    typedef Radio_P Radio;

    public:

    //Definitions
    typedef OsModel_P OsModel;
    typedef Radio_P Radio;

    The first file included is mandatory in all of the topology control algorithms. It contains a single function needed to notify the upper algorithms that its job is done. The second file is not obligatory but as within the Wiselib there is no such thing as a standard library, these files provide a Vector (C++ standard lib-like) implementation but with a fixed compile-time capacity.

    As commented in section 2.2.2 the code in Wiselib uses templates to be able to use the same code for different architectures. In the case of topology algorithms, just the OS is needed in the templates parameters and additionally, and this is common practice, the maximum number of neighbours (or in some case the number of nodes in the network) a node is expected to have is specified. This number is used to build the structures needed, like arrays, etc. Finally, all the topology algorithms must be a subclass of TopologyBase.
CHAPTER 3. PROGRAMMING TC ALGORITHMS IN THE WISELIB

```cpp
typedef typename OsModel_P::Radio Radio;
typedef typename OsModel_P::Timer Timer;

#ifdef DEBUG_BASIC_TOPOLOGY
typedef typename OsModel::Debug Debug;
#endif

typedef BasicTopology<OsModel, MAX_NODES> self_type;
typedef typename OsModel::Os Os;
typedef typename OsModel_P::size_t size_type;
typedef typename Radio::node_id_t node_id_t;
typedef typename Radio::size_t size_t;
typedef typename Radio::block_data_t block_data_t;
typedef typename Timer::millis_t millis_t;
typedef vector_static<OsModel, node_id_t, MAX_NEIGHBORS> Neighbors;

Some common definitions that will be useful in the rest of the code.

```cpp
//Functions
BasicTopology();
~BasicTopology();
void enable( void );
void disable( void );
void timer_elapsed( void *userdata );
void receive( node_id_t from, size_t len, block_data_t *data );
inline void set_work_period( millis_t work_period ) { work_period_ = work_period; };
inline void set_iterations( uint8_t iterations ) { iterations_ = iterations; };
Neighbors &topology(){
   return N;
}
```

The functions `enable` and `disable` must be implemented by every algorithm in the Wiselib. The function `topology` has to be implemented by all the sub-classes of `TopologyBase`. Function `timer_elapsed` is the one that gets executed by the timer and `receive` will do it every time the radio receives a message; both of them will be have to be registered to do so later. Finally, `set_work_period` is used to set the time the timer has to wait and `set_iterations` sets the number of iterations the timer has to do.

```cpp
private:
   Radio& radio()
   { return *radio_; }
   Timer& timer()
   { return *timer_; }
#endif
Debug& debug()
```
{ return *debug_; }
#endif

Radio * radio_;
Timer * timer_;
#ifdef DEBUG_BASIC_TOPOLOGY
Debug * debug_;
#endif

//variables
millis_t work_period_;
uint8_t iterations_;
uint8_t iterations_done;
Neighbors N;

The vector N is where we are going to store the IDs of our neighbours. As commented before, here we make use of the constant MAX_NODES to construct this array.

BASICTopology()
: os_( 0 ),
work_period_( 5000 ),
iterations_(5)
{};

Some defaults if they are not specified.

~BASICTopology()
{
#ifdef DEBUG_BASIC_TOPOLOGY
debug().debug("%i:␣BasicTopology␣Destroyed\n", radio().id());
#endif
}

The Wiselib provides one interface (Debug::debug) identic to the function printf that guarantees it will work in all the architectures. Also, is common practice to make the debug messages optionally at compile-time to save some weight in the final executable.

void enable( void )
{

Until the enable function is called, the algorithm does not start to run.

iterations_done = 0;
N.clear();
radio().enable_radio();

The radio has to be enabled prior to start transmitting and receiving.

#ifdef DEBUG_BASIC_TOPOLOGY
debug().debug("%i:␣BasicTopology␣Boots\n", radio().id());
#endif
radio().template reg_recv_callback
<self_type, &self_type::receive>(this);
timer().template set_timer<self_type, &self_type::timer_elapsed>
(startup_time_, this, 0);

Here is where the callback registration of both the timer and the receive function actually happens.
void disable( void )
{
#ifdef DEBUG_BASIC_TOPOLOGY
    debug().debug("%i:␣BasicTopology␣disabled\n", radio().id());
#endif
}

void timer_elapsed( void* userdata )
{
    if(iterations_done == iterations_){
        TopologyBase<OsModel>::notify_listeners();

        return;
    }
}

uint8_t[2] message = {4, 2};

This is the message we will be transmitting. It could be anything, but in the example we have put these random values just to illustrate how to send a message.

radio().send( radio().BROADCAST_ADDRESS,
    sizeof(uint8_t) * 2,
    &message
);

Here is shown how to send a message: the first parameter is the OS; the second is the destination. In this case is the broadcast reserved address, but it could be an ID of a node to send a unicast message. The third parameter is the size of the message; and the fourth is a pointer to the actual data.

iterations_done_ += 1;
timer().template set_timer<self_type, &self_type::timer_elapsed>(startup_time_, this, 0);

We have to register the timer again. The registration only executes the callback once per registration.

void receive( node_id_t from, size_t len, block_data_t *data )
{

    uint8_t* message = (uint8_t*) data;
    if ( from == radio().id() )
        return;

    Here we check if the message is not from ourselves. If positive, we discard it.
for(unsigned int i; i < N.size(); i++){
    if(from == N[i])
        return;
}
N.push_back(from);

And here we add to our vector of neighbours if we have not already.
Chapter 4

The CBTC algorithm

As explained by “Analysis of a Cone-Based Distributed Topology Control Algorithm for Wireless Multi-hop Networks”, by Li Li, Joseph Y. Halpern, Paramvir Bahl, Yi-Min Wang and Roger Wattenhofer.

In this chapter we introduce Cone Based Topology Control algorithm. It is divided mainly in three blocks: the basic CBTC algorithm, the optimizations and the protocol mechanism to maintain the network once these two first phases are done.

4.1 Introduction

CBTC is a direction-based topology control algorithm. This kind of algorithms rely on the ability of nodes to estimate the relative position of their neighbours. This is relatively less accurate information than knowing exact node locations, as the former type of information can be determined if the latter is known, but not vice versa.

Several techniques for estimating the direction from which a certain node is transmitting have been proposed and discussed in the IEEE Antenna and
Propagation community. This problem is known as the Angle-of-Arrival (AoA) problem, and it is typically solved by equipping nodes with more than one directional antenna. So, in the case of directional information, some extra hardware on the nodes (with respect to the standard assumption of nodes equipped with a single, omnidirectional antenna) is also needed in order to provide the requested information. An advantage of using AoA-based techniques, in which the exact location of the nodes is provided or calculated, instead of location-based techniques, is that the AoA can be accurately estimated in indoor environments as well.

4.2 The Cone Based Topology Control algorithm

The Cone Based Topology Control (CBTC) algorithm was proposed by Roger Wattenhofer, Li Li, Paramvir Bahl and Yi-Min Wang in [16] and further optimized and corrected in [7]. This design and implementation is based in the latter. As mentioned before, it is a direction-based topology control algorithm. As for all the topology control algorithms, it is to establish a logical network topology on top of the physical one for communicating, while maximizing the network longevity and maintaining global connectivity with reasonable throughput. In order to maximize the network longevity, you have to be energy-efficient (i.e. to use less power), since the main power consuming source in wireless sensor networks is transmission. CBTC algorithm pays special attention to a tight adjustment of the transmission powers of the nodes resulting in the outcoming logical topology.

The topology control problem can be formalized as follows: we are given a set $V$ of possibly mobile nodes located in the plane. Each node $u \in V$ is specified by its coordinates, $(x(u), y(u))$ at any given point in time. Each node $u$ has a power function $p$ where $p(d)$ gives the minimum power needed to establish a communication link to a node $v$ at distance $d$ away from $u$. Let’s assume that the maximum transmission power $P$ is the same for every node, and the maximum distance for any two nodes to communicate directly is $R$, i.e. $p(R) = P$. If every node transmits with power $P$, then we have an induced graph $G_R = (V, E)$ where $E = (u, v) | d(u, v) \leq R$ ($d(u, v)$ is the Euclidean distance between $u$ and $v$).

To explain CBTC we consider three communicatoin primitives: broadcast, send and receive, defined as follows:

- $bcast(u, p, m)$ is invoked by node $u$ to send message $m$ with power $p$; it results in all nodes in the set $\{ v | p(d(u, v)) \leq p \}$ receiving $m$.
- $send(u, p, m, v)$ is invoked by node $u$ to send message $m$ to node $v$ with power $p$. This primitive is used to send unicast messages, i.e. point-to-point messages.
- $recv(u, m, v)$ is used by $u$ to receive message $m$ from $v$.

Let’s assume that when node $v$ receives a message $m$ from node $u$, it knows the reception power $p'$ of message $m$. This is, in general, less than the power $p$ with which $u$ sent the message, because of the radio signal attenuation in space. Moreover, we assume that, given the transmission power $p$ and the reception power $p'$, $u$ can estimate $p(d(u, v))$. This assumption (in terms of use by the algorithm) is reasonable in practice.
The basic idea of the algorithm is that a node $u$ transmits with the minimum power $p_{u,\alpha}$ required to ensure that in every cone of degree $\alpha$ around $u$, there is some node that $u$ can reach with power $p_{u,\alpha}$. To put it differently, this is the equivalent to considering the maximum angle between two consecutive neighbour nodes to be $\alpha$. Node $u$ starts running the algorithm by broadcasting a “Hello” message using the lowest transmission power, it waits for the possible answers to this message and collects them. It gradually increases the transmission power to discover more neighbours. Every time a node is discovered by $u$ it adds the direction where it came from to an internal list; then checks whether each cone of degree $\alpha$ contains a node. That is easy to check: having the nodes sorted by the angle relative to some reference node it is immediate that there is a gap of more than $\alpha$ between the angles of two consecutive nodes if, and only if, there is a cone of degree $\alpha$ centered at $u$ that contains no nodes. If there exists such gap, then node $u$ broadcasts with greater power. This continues until either node $u$ have no such gap anymore or $u$ is transmitting at maximum power.

**Algorithm 1** The Basic CBTC running at each node $u$.

**INPUT:** $\alpha$

$N_u \leftarrow \emptyset$; //the set of discovered neighbours of $u$

$D_u \leftarrow \emptyset$; //the directions from which the ACKs have come

$p_u \leftarrow p_0$;

**while** $(p_u < P$ and $\text{gap-}\alpha(D_u))$ **do**

$\text{broadcast}(u, p_u, ("Hello", p_u))$ and gather ACKs;

$N_u \leftarrow N_u \cup \{v : v \text{ discovered}\}$;

$D_u \leftarrow D_u \cup \{\text{dir}_u(v) : v \text{ discovered}\}$;

$p_u \leftarrow \text{Increase}(p_u)$;

**end while**

**OUTPUT:** $N_u, p_u$

Algorithm 4.2 shows the basic CBTC algorithm. In the algorithm, a “Hello” message is originally broadcasted using some minimal power $p_0$. In addition, the power used to broadcast the message is included in the message. The power is then increased at each step using some function $\text{Increase}$. In the original paper [7] no proposal is done neither on how to choose the initial power ($p_0$), nor how to increase the power at each step; it is just assumed a function $\text{Increase}$ such that $\text{Increase}^k(p_0) = P$ given an enough large $k$, i.e., the maximum transmission power can be reached from the initial transmission power $p_0$ by successive increments. Details on how we implemented these issues are given on chapter 5. Upon receiving a “Hello” message from $u$, node $v$ responds with an acknowledgement (ACK) message. Upon $v$’s reception of the ACK from $v$, node $u$ adds $v$ to its set $N_u$ of neighbours and adds $v$’s direction $\text{dir}_u(v)$ (measured as angle relative to some fixed angle) to its set of $D_u$ directions. $\text{gap-}\alpha(D_u)$ tests whether a gap greater that $\alpha$ exists in the angles in $D_u$. And what “gather the ACKs” implies is to receive ($\text{recv}(u,’ack’, v)$) for every neighbour such as $v|p(d(u)) \leq p_u$.  

4.2.1 Asymmetric neighbours

A notable situation that can be reached after the CBTC is performed is what we call asymmetric links. Let $N_\alpha(u)$ be the final set of discovered neighbours computed by node $u$ at the end of running $\text{CBTC}(\alpha)$; let $p_{u,a}$ be the corresponding final power selected by node $u$. Let $N_\alpha = \{(u,v) \in V \times V : v \in N_\alpha(u)\}$. Note that the $N_\alpha$ relation is not symmetric. As the following example shows, it is possible that $(v,u) \in N_\alpha$ but $(u,v) \not\in N_\alpha$.

A possible escenario follows. Suppose that $V = u_0, u_1, u_2, u_3, v$. Further suppose that $d(u_0,v) = R$. Choose $\epsilon$ with $0 < \epsilon < \pi/12$ and place $u_1, u_2, u_3$ so that:

1. $\angle vu_0u_1 = \angle vu_0u_2 = \pi/3 + \epsilon = \alpha/2$
2. $\angle u_1vu_0 = \angle u_2vu_0 = \pi/3 - \epsilon$ (so that $\angle vu_1u_0 = \angle vu_2u_0 = \pi/3$)
3. $\angle vu_0u_3 = \pi$ (so that $\angle u_1u_0u_3 = \angle u_2u_0u_3 = 2\pi/3 - \epsilon$)
4. $d(u_0, u_3) = R/2$

Note that, given $\epsilon$ and the positions of $u_0$ and $v$, the positions of $u_1, u_2, u_3$ are determined. Since $\angle u_1u_0v > \angle u_0u_1v > \angle vu_1u_0$, it follows that $d(u_1, v) > d(u_0, v) = R > d(u_0, u_1)$; similarly $d(u_2, v) > R > d(u_0, u_2)$. It easily follows that $N_\alpha(u_0) = u_1, u_2, u_3$ while $N_\alpha(v) = u_0$, as long as $2\pi/3 < \alpha \leq 5\pi/6$. Thus, $(v, u_0) \in N_\alpha$, but $(u_0, v) \not\in N_\alpha$.

![Figure 4.2: Asymmetric neighbour](image)

Let $G_\alpha = (V, E_\alpha)$, where $V$ consists of all nodes in the network and $E_\alpha$ is the symmetric closure of $N_\alpha$; that is, $(u,v) \in E_\alpha$ if, and only if either $(u,v) \in N_\alpha$ or $(v,u) \in N_\alpha$. Note that this example shows the need for taking the symmetric closure in computing $G_\alpha$. Although $(u,v) \in G_R$, there would be no path from $u_0$ to $v$ if we considered just the edges determined by $N_\alpha$, without taking the symmetric closure. As it has been previously explained, each node $u$ knows the power required to reach all nodes $v$ such that $(u,v) \in E_\alpha$: it is the maximum of $p_{u,a}$ and the power required by $u$ to reach each all nodes $v$ from which it received a “Hello” message.

Definitions:
• $\text{rad}_{u,\alpha}$: the distance $d(u,v)$ of the neighbour $v$ farthest from $u$ in $N_{\alpha}(u)$; that is, $p(\text{rad}_{u,\alpha}) = p_{u,\alpha}$

• $\text{rad}_{u,\alpha}$: the distance $d(u,v)$ of the neighbour $v$ farthest from $u$ in $E_{\alpha}$

### 4.2.2 $\alpha$ parameter: $5\pi/6$ vs $2\pi/3$

There is only one parameter needed on a theoretical level by CBTC: the $\alpha$ parameter, which denotes the angle used to test the existence of the gap. It is desirable that the value of this parameter is as big as possible. In principle, it seems that the greater the value of $\alpha$ is, the better. With a great $\alpha$ the condition to get the second phase is met earlier (the bigger $\alpha$ is, the chance that a neighbour node fills the gap is more probable) so the node configures itself with a lower transmission power and saves energy.

But of course, the angle $\alpha$ can not be made as great as desired, since the final logical topology has to preserve the connectivity. In other words, if there is a path from node $s$ to node $t$ when every node communicates at maximum power then, whichever $\alpha$’s size is chosen, there has to be a path in the smallest symmetric graph $G_{\alpha}$ containing all edges $(u,v)$ such that $u$ can communicate with $v$ with power $p_{u,\alpha}$.

Throughout this document, two thresholds are considered for the angle $\alpha$: $2\pi/3$ and $5\pi/6$. The initial version of the CBTC algorithm considered $\alpha = 2\pi/3$ a good angle that preserved connectivity. Later, it was demonstrated that taking $\alpha = 5\pi/6$ is a necessary and sufficient condition to guarantee that network connectivity is preserved. When $\alpha > 5\pi/6$, connectivity is not necessarily preserved. Additionally, it was found that with $\alpha \leq 2\pi/3$, the output logical topology can be optimized by removing the asymmetric links described in the previous section. This is one of the three optimizations that can be applied to the basic CBTC algorithm. We describe them in the next section.

### 4.3 Optimizations

Optimizations on the basic CBTC algorithm aim at reducing the transmission power chosen by the nodes to build the logical topology, not at reducing the number of neighbours. Reducing their number of neighbours (or connections) does not imply saving more power and additionally it can cause performance penalties. In the following, we explain the existing optimizations that apply to the CBTC.

#### 4.3.1 The shrinkback operation

At the end of the basic CBTC($\alpha$) there will be some nodes that still have an $\alpha$-gap. In CBTC terminology these nodes are called boundary nodes. A direct consequence of this is that those nodes are transmitting at maximum power. This first optimization deals with that kind of nodes.

It is natural to see that the transmission power could be reduced at least to the maximum power needed to reach all your neighbours. In fact, and going further, all the nodes not contributing to the coverage could be removed, however, CBTC focuses on power minimization (not the number of neighbours).
The CBTC approach to this shrinking phase is to reduce gradually the transmission power of these nodes while coverage is preserved. Where coverage is defined in the following way: given a set \( \text{dir} \) of directions (angles) and an angle \( \alpha \).

\[
\text{cover}_\alpha(\text{dir}) = \{ \theta; \text{ for some } \theta' \in \text{dir}, |\theta - \theta'| \mod 2\pi \leq \alpha/2 \}
\]

So, roughly speaking, the coverage is all the space formed by the angle \( \alpha/2 \) around both sides of the directions where the neighbours of the node point at.

In order to include this optimization, the basic CBTC algorithm has to be modified so that, at each iteration, every node \( u \) includes in \( N_u \) is tagged with the power used the first time it was discovered (i.e. tagged with the minimum power to reach that node). As well, suppose that the power levels used by node \( u \) during the algorithm were \( p_1, \ldots, p_k \), if \( u \) is a boundary node, \( p_k \) is the maximum power \( P \). Then, the procedure to follow is to successively remove nodes tagged with power \( p_k \), then \( p_{k-1} \), and so on, as long as their removal does not change the coverage. That is, let \( \text{dir}_i, i = 1, \ldots, k \) be the set of directions found with all power levels \( p_i \) or less, then the minimum \( i \) such that \( \text{cover}_\alpha(\text{dir}_i) = \text{cover}_\alpha(\text{dir}_k) \) is found.

It is easy to check that after that, the network preserves the connectivity.

To summarize in one specific scenario, in Figure 4.3 at the left picture the nodes X and Y would be removed as the distance between A and B is less than \( \alpha \) so the same space is covered. Similarly for Y there are the nodes B and C. In the left picture, the situation has changed a little; now, while all the previous nodes remain in the same position, there are two additional nodes L and M that are in another power level transmission, even farther X and Y. In the case of L, its situation is the same as X and Y, as there are two nodes (C and Y) that cover it. But there is also M, that changes the whole situation as there are no nodes in lower power levels to cover it, so there is no node in this scenario that can be removed. Remember that if such a node is found, the nodes in the same and below power levels cannot be removed.

Following is the pseudocode for the shrinkback function. It is convenient to think of \( N_u \) as a linked list and not as a vector because of the meaning of functions \( \text{find_previous} \) and \( \text{find_next} \). The \( \text{dist} \) function of a node object calculates the distance between the node calling and the neighbour of the parameter and it is not a "commutative" function: \( x.\text{dist}(y) = 2\pi - y.\text{dist}(x) \).
CHAPTER 4. THE CBTC ALGORITHM

4.3.2 Asymmetric edge removal

Again, to implement that optimization, the basic CBTC needs to be slightly modified. This time, after finishing CBTC(\(\alpha\)), a node \(u\) must send a message to each node \(v\) to remove himself from \(N_\alpha(v)\) when constructing \(E^-_\alpha\) (that is, the resulting of the basic CBTC algorithm without the asymmetric links).

The shrinkback optimization and this one can be applied together. It does not matter in which order, in terms of connectivity. But from an energy-performance point of view it is wiser to apply the shrinkback first, as it only depends on internal calculations, so power and neighbours have decreased and this optimization requires sending messages.

It is clear that there is a tradeoff between using CBTC(\(5\pi/6\)) and using CBTC(\(2\pi/3\)) with asymmetric edge removal. In general, \(p_u,5\pi/6\) (i.e. \(p(\text{rad}_{u,5\pi/6})\)) will be smaller than \(p_u,2\pi/3\). However, the power \(p(\text{rad}_{u,5\pi/6})\) with which \(u\) needs to transmit may be greater than \(p_u,5\pi/6\) since \(u\) may need to reach nodes \(v\) so that \(u \in N_{5\pi/6}(u)\), but \(v \notin N_{5\pi/6}(u)\). In contrast, if \(\alpha = 2\pi/3\), then asymmetric edge removal allows \(u\) to still use \(p_u,2\pi/3\) and may allow \(v\) to use power less than \(p_u,2\pi/3\).

4.3.3 Pairwise edge removal

This third and final optimization aims at further reducing the transmission power of each node. This time again two additional pieces of information are needed. First, each node \(u\) is assigned a unique identifier denoted as \(ID_u\), and that \(ID_u\) is included in all of \(u\’s\) messages. Note that, although you usually get these

Algorithm 2 Shrinback pseudocode

INPUT: \(\alpha\), \(N_u\)

\[
L \leftarrow N_u\text{.levels}(); // \text{the set of levels of nodes in } N_u
L\text{.sort}() // \text{from higher to lower}
L\text{.uniq()}
L \leftarrow L[\text{:} -1] // \text{Remove the lowest power}
\]

for \(l\) in \(L\) do
  for \(n\) in \(N_u\) do
    if \(n\text{.level} = l\) then
      \(k \leftarrow N_u\text{.find}\_\text{previous}(function x: x\text{.power} < l)\)
      \(m \leftarrow N_u\text{.find}\_\text{next}(function x: x\text{.power} < l)\)
      if \(k\text{.dist}(n) + n\text{.dist}(m) > 2 * \alpha\) then
        return
      end if
    end if
  end for
end for

OUTPUT: \(N_u\)
CHAPTER 4. THE CBTC ALGORITHM

Algorithm 3 Asymmetric Removal pseudocode

\begin{verbatim}
INPUT: \(N_u\)

for \(n\) in \(N_u\) do
    if \(n\).asymmetric then
        send\((u, n.power, "asymmetric", n)\)
        \(N_u\).remove(function \(x: x.id == n.id\))
    end if
end for

OUTPUT: \(N_u\)
\end{verbatim}

Identifiers easily from most hardware, the basic CBTC algorithm does not need this information in order to work properly. The IDs are needed because, when this optimization is executed, some neighbours are removed at every side of the edge. But, of course, and because we do not want asymmetric links, the same edge has to be removed respectively at every side. The mechanism employed to do that uses these IDs to guarantee that this happens.

The second piece of information needed is that, given any node \(u\) and any pair of its neighbours \(v\) and \(w\), node \(u\) needs to know which of them is closer. This can be achieved as follows. As explained in section 4.2, every node, when executing the CBTC algorithm, grows its radius of transmission in discrete steps and the transmission power level used is included in every “Hello” message that the node sends. At the same time, each discovered neighbour (i.e. the one receiving the “Hello” message) also includes its own transmission power level in the ACK message. With this information, assuming the existing relation between reception power and distance, \(u\) can deduce which one is farther comparing the first two power levels received.

With this information we can proceed to reduce once more the transmission power by removing more edges while still preserving connectivity. Even after the shrinkback operation and possibly asymmetric edge removal, there are still edges that can be removed, thus possibly reducing the power of others. Given the triangle formed by three edges, it is obvious that if you remove any of them the connectivity is preserved. Going further, as demonstrated in [7], given nodes \(u, v_1, v_2\), if there is an edge between \(u\) to \(v_1\) and from \(u\) to \(v_2\), it is possible to remove the longer edge, even if there is no edge from \(v_1\) to \(v_2\), as long as \(d(v_1, v_2) < \max(d(u, v_1), d(u, v_2))\). A condition sufficient to check that \(d(v_1, v_2) < \max(d(u, v_1), d(u, v_2))\) is that the angle formed by \(v_1, u\) and \(v_2\) is smaller than \(\pi/3\), i.e. \(\angle v_1uv_2 < \pi/3\), since the longest edge will be opposite to the largest angle, and an angle smaller than \(\pi/3\) will not be the largest, by definition.

To put that more formally, we need to define the notion of an edge ID: for every edge \((u, v)\) let \(eid(u, v) = (i_1, i_2, i_3)\) be its edge ID, where \(i_1 = d(u, v)\), \(i_2 = \max(ID_u, ID_v)\) and \(i_3 = \min(ID_u, ID_v)\). Recall that \(ID_u\) and \(ID_v\) are the identifiers of nodes \(u\) and \(v\), respectively. The IDs of two edges can be compared lexicographically so that \((i, j, k) < (i', j', k')\) if, and only if either:

1. \(i < i'\), or
2. \(i = i'\) and \(j < j'\), or
3. $i = i', j = j'$ and $k < k'$

We have that, if $v$ and $w$ are neighbours of $u$, $\angle uvw < \pi/3$, and $eid(u, v) > eid(u, w)$, then $eid(u, v)$ is a redundant edge and can be removed.

Note that here we are removing the longer edge thus allowing us to reduce power as now we will not be forced to transmit to the farther node.

Once again, as in with shrinkback operation, there is a situation where neighbours could be removed, because they are redundant, but it is not of interest. A neighbour will be removed if all the neighbours in its power level transmission and higher can also be removed. Remember that the interest resides in saving power, not in removing edges.

4.4 Dealing with reconfiguration, asynchrony and failures

Wireless sensor networks are, by nature, dynamic networks. Even if nodes do not move physically, they may die if they run out of energy or suffer some environmental accident. Moreover, new nodes might possibly be added. There is a need for a mechanism that is able to track these changes and also a way to recover once this is noticed. In this section, we explain how we deal with this dynamic nature.

4.4.1 Neighbour Discovery Protocol

The mechanism employed to track these changes is what is called a Neighbour Discovery Protocol (NDP). An NDP is usually a simple beaconing protocol whose mission is to notify the neighbours of one node that is alive. Of course this can be implemented in several ways, but in CBTC it is implemented as follows.

On the side of the node willing to note its presence, the node starts beaconing every $t$ time. The node’s ID and the power used to transmit the beacon is
CHAPTER 4. THE CBTC ALGORITHM

Algorithm 4 Pairwise Removal pseudocode

INPUT: \( N_u \)

\[
L \leftarrow N_u.\text{levels}(); \quad \text{// the set of levels of nodes in } N_u \\
L.\text{sort}() \quad \text{// from higher to lower} \\
L.\text{uniq}() \\
L \leftarrow L[\ldots -1] \quad \text{// Remove the lowest power}
\]

for \( l \) in \( L \) do
  for \( n \) in \( N_u \) do
    if \( n.\text{level} = l \) then
      \( k \leftarrow N_u.\text{find\_previous}(function\ x: x.\text{power} < l) \)
      \( m \leftarrow N_u.\text{find\_next}(function\ x: x.\text{power} < l) \)
      if \( k.\text{dist}(n) > \pi/3 \) and \( n.\text{dist}(m) > \pi/3 \) then
        return
      end if
    end if
  end for
end for

OUTPUT: \( N_u \)

included in it. On the other side, a neighbour is considered failed if a predefined number of beacons are not received for a certain time interval \( r \). A node \( v \) is considered a new neighbour of \( u \) if a beacon was received from \( v \) during the previous \( r \) interval.

With this information we classify all changes in the lifetime of the network by three basic events:

- A \( \text{join}_u(v) \) event happens when node \( u \) detects a beacon from node \( v \) for the first time.
- A \( \text{leave}_u(v) \) event happens when node \( u \) misses some predetermined number of beacons from node \( v \).
- A \( \text{change}_u(v) \) event happens when \( u \) detects that \( v \)’s angle with respect to \( u \) has changed.

Note that if for some reason a \( \text{leave}_u(v) \) event happens, then node \( v \) will be treated as if it was a new node; that is, if \( u \) receives a new beacon from \( v \) after that \( \text{leave}_u(v) \), an event \( \text{join}_u(v) \) will be raised.

4.4.2 The reconfiguration algorithm

CBTC’s reconfiguration algorithm is simple. For convenience it is assumed that each node is tagged with the power used when it was first discovered, as in the shrinkback operation (this is not necessary, but it minimizes the number of times that CBTC needs to rerun).

It basically reacts to one or several of the three basic events:
CHAPTER 4. THE CBTC ALGORITHM

• If a leave\(_u(v)\) event happens, and if there is an \(\alpha\)-gap after dropping \(\text{dir}_u(v)\) from \(D_u\), node \(u\) reruns CBTC(\(\alpha\)), starting with power \(p(\text{rad}_{u,\alpha})\) (i.e. taking \(p_0 = p(\text{rad}_{u,\alpha})\)).

• If a join\(_u(v)\) event happens, \(u\) computes \(\text{dir}_u(v)\) and the power needed to reach \(v\). As in the shrinkback operation, \(u\) then removes nodes, starting with the farthest neighbour nodes and working back, as long as their removal does not change the coverage.

• If a change\(_u(v)\) event happens, node \(u\) modifies the set \(\text{dir}_u(v)\) of directions appropriately. If an \(\alpha\)-gap is then detected, then CBTC(\(\alpha\)) is rerun, again starting at \(p(\text{rad}_{u,\alpha})\). Otherwise, nodes are removed, as in the shrinkback operation, to see if less power can be used.

In general, there may be more than one change event that is detected at a given time by a node \(u\) (for example, if \(u\) moves, then there will be in general several leave, join and change events detected by \(u\)). If more than one change event is detected by \(u\), we perform the changes suggested above as if the events are observed in some order, as long there is no need to rerun CBTC. This will happen when, after a leave or a change event, an \(\alpha\)-gap is detected. If CBTC needs to be rerun, it deals with all changes simultaneously.

4.4.3 Beaconing Power and Optimizations

An important point to note is what power has to be used to broadcast the beacons. Intuitively a node should broadcast with sufficient power to reach all of its neighbours in \(E_\alpha\) (or \(E^-_\alpha\), if \(\alpha \leq 2\pi/3\)). And that would work with the basic CBTC(\(\alpha\)) and with asymmetric edge removal (in that case, power \(p(\text{rad}_{u,\alpha})\) would suffice). But it is not the case with the other two optimizations.

When the shrinkback is performed, using the power to reach just the neighbours left after the operation is not enough. Suppose that the network is temporarily partitioned in two. There will be at least two nodes, one at each partition, that will become boundary nodes. If one (or both) nodes change its position and become in range again, but not as closer as to be covered by their current range, they both will never become aware of their presence. To solve this situation, all boundary nodes will beacon with the power computed by the basic CBTC(\(\alpha\)).

Similarly, with pairwise edge removal it is necessary to broadcast with \(p(\text{rad}_{u,\alpha})\) (or \(p(\text{rad}_{u,\alpha})\) in the case of \(\alpha < 2\pi/3\)), the power needed to reach all of \(u\)'s neighbours in \(E_\alpha\) (or \(N_\alpha\) in the aforementioned case), not just the power needed to reach all of \(u\)'s neighbours after the optimization.

4.4.4 Failures and other issues

CBTC has no data integrity methods or any kind of detection of malformed messages. CBTC just considers crash failures: either a node crashes (or runs out of battery) and stops sending messages, or it follows the algorithm correctly. So the algorithm would start misbehaving in that situation. It is not inherent to the algorithm, it is just that there are no built-in mechanisms in this primitive design.

Intuitively, this reconfiguration algorithm preserves connectivity. This is mostly true but there is one conditionant that can provoke a loss connectivity.
If the topology changes frequently enough, the reconfiguration algorithm may not ever catch up with the changes, so there may be no point at which the connectivity of the network is actually preserved. What CBTC guarantees is that, if the topology ever stabilizes, so that there are no further changes, then the reconfiguration algorithm eventually results in a graph that preserves the connectivity of the final network, as long as there are periodic beacons.

Finally, this reconfiguration algorithm guarantees that each cone of degree $\alpha$ around a node $u$ is covered, except for boundary nodes; just as the basic algorithm does, thus the connectivity is preserved.

According to what was stated in paper [7], it is worth noting that the reconfiguration algorithm works perfectly well in an asynchronous setting. Now that nodes are assumed to communicate asynchronously, messages may get lost or duplicated, and nodes may fail. Node failures result in leave events, as lost messages do. If node $u$ gets a message after many messages having been lost, there will be a join event corresponding to the earlier leave event.

### 4.5 Summary

<table>
<thead>
<tr>
<th>Algorithm 5</th>
<th>CBTC with optimizations and Maintenance mode pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT:</strong></td>
<td>$\alpha$</td>
</tr>
<tr>
<td><strong>loop</strong></td>
<td></td>
</tr>
<tr>
<td>basic_CBTC()</td>
<td></td>
</tr>
<tr>
<td>shrink_back()</td>
<td></td>
</tr>
<tr>
<td><strong>if</strong></td>
<td>$\alpha &lt;= 2\pi/3$ then</td>
</tr>
<tr>
<td>asymmetric_edge_removal()</td>
<td></td>
</tr>
<tr>
<td><strong>end if</strong></td>
<td></td>
</tr>
<tr>
<td>pairwise_edge_removal()</td>
<td></td>
</tr>
<tr>
<td><strong>while</strong></td>
<td>check_gap() do</td>
</tr>
<tr>
<td>maintenance_mode()</td>
<td></td>
</tr>
<tr>
<td><strong>end while</strong></td>
<td></td>
</tr>
<tr>
<td><strong>end loop</strong></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

CBTC implementation

5.1 Introduction

In this chapter the relevant details of the implementation of CBTC within the Wiselib are explained.

5.1.1 No Angle-of-Arrival (AoA) hardware

Before going into details, there is one important factor that conditions several parts of the implementation of CBTC. None of the hardware supported by the Wiselib is AoA capable; that is, whenever the sensor receives a message, it is not able to determine which direction it came from. To overcome this limitation, it is possible to use some kind of Global Positioning System (GPS). The only thing needed then is to send your own position on every message and the receiving node just has to compare its own position with the one of the received message to compute the direction. Note that you do not have to perform these steps in every message sent, just on the ones where direction has to be known.

Although in the original paper [7] this way to proceed is contemplated (as a valid one), it has several disadvantages; obviously, the need for one GPS device, and its derived consequences:

The GPS itself One of the main advantages of the direction-based topology control algorithms, and specifically of CBTC, is that generally they work without the need of GPS. Instead, they just rely on direction information. This is an advantage in itself for the various reasons given below.

Energy The main purpose of a topology algorithm is to decrease the power used in each node, or globally in the network, in order to extend networks lifespan. A GPS will always be more energy consuming because it has to transmit to a satellite.

Availability GPS information may not be always available. For instance, in indoor environments is not. Anyway, you always depend on external information that you do not control.
Response time With AoA techniques, the time to determine the direction is instantaneous. But with GPS, as you have to know your position previously, this incurs in high delays. In a mostly static network you could cache your position and work with that one, but even then, you have to periodically refresh your cache. With enough mobile networks, this situation would lead to constantly wrong information about the direction.

Validity When AoA is used, you know with certainty which direction the message came from. But with GPS you rely on trusting the other devices’ position and it could be wrong because of a malicious node deliberately giving incorrect information, or the other (or even the node itself) could have wrong information about themselves.

Accuracy Even if it is correct information, the position’s accuracy will always be more precise with an AoA technique than with GPS. This flaw would be more significant and visible when the distance between nodes and the maximum error distance are the same and then the information will be completely misguiding.

Length of messages Of course, since you are transmitting your position every time you send a message, this increases the length of your messages.

5.2 Overview
The CBTC implementation can be easily divided (for the purpose of being explained) in two phases. The first phase (or "Discovery phase"), is the one that finalizes once the basic algorithm plus the applicable optimizations (2 or 3, depending of $\alpha$) have been performed. The second phase (or "Maintenance phase") is the one where the reconfiguration algorithm is working. Have in mind that you can go back to the first phase while in the second as described by the CBTC algorithm.

All the interactions between the nodes are originated from either the two timers (one for each phase) or the messages sent during their executions.

5.3 The Discovery phase
When the node starts to work for the first time, it does it in the Discovery phase. This phase’s timer broadcasts a HELLO message at every iteration. When the first message is broadcasted the transmission power is in the minimum the hardware is able to transmit. Then at every iteration it increases in steps the transmission power and sends again the HELLO message. In that regard, the Wiselib abstracts the radio of the different hardware and offers an interface to do that even if the device’s radio transmission power does not work with integers (i.e. it is configured with a float parameter. For instance the simulator Shawn [10], the Wiselib supported simulator, works with a float in the range $0.0 - 1.0$). Every time a HELLO message is received, the receiver immediately answers with an ACK message.

```c
typedef struct triplet_t{
    double angle;
};
```
Throughout all the execution we store in the array Neighbours all of our neighbours in a given moment. Every time the node receives either a HELLO or an ACK message we store it in Neighbours. Additionally when an ACK message is received we check if there still is a $\alpha$-gap. We just check for the existence of the gap at the arrival of an ACK message because, when checking for the gap, we have to take it into account.

The Discovery phase ends when either there is no gap or when we enter again the timer and the last transmitted HELLO message was at the maximum transmission power.

### 5.3.1 Messages (in the Discovery Phase)

**HELLO message**

<table>
<thead>
<tr>
<th>Message</th>
<th>Type</th>
<th>Id</th>
<th>Transmission power level</th>
<th>Position</th>
</tr>
</thead>
</table>

Figure 5.1: HELLO message

In the Discovery Phase, each node issues a HELLO message at every iteration of the timer of this phase. The message includes: the id of the node itself, the power used to transmit it and, additionally, its own location to emulate the AoA (as explained before).

In the Discovery Phase, on reception of a HELLO message from $v$, a node $u$ looks in its Neighbours array if $v$ is already in it. If it is not, $v$ is added to $u$’s Neighbours sorted by angle relative to $u$’s position. We keep Neighbours sorted all the time because, in order to do the gap checking and the optimizations, we have to have them sorted this way. Additionally, along the angle, $u$ also stores the id of $v$, the power used to transmit the message (as it is included in itself) and finally a flag that determines if it is asymmetric to $u$. The easiest way to define if a node $v$ is asymmetric to $u$ is if $u$ has received at least one HELLO message from $v$ but has not received any ACK message from it.

Otherwise, if $v$ is already in the array, we update its position (angle) in case it has changed (and sort it adequately) and only change the power if it is less than the one stored. It has to be this way because we are interested in the minimum power to reach a node and, as explained several times before, in the Discovery Phase a node $v$ will issue a HELLO message at every iteration, each time with a greater power. So, if $u$ updated the power every time it receives a HELLO message from $v$, the power $u$ would have regarding $v$ would be the last received from $v$’s Discovery Phase, not the first one. This way of procedure could lead to false information as well. Basically, if a node $u$ stores a certain power in regard to $v$ and then $v$ goes out of battery or $v$ simply moves farther in respect to $u$, the power $u$ thinks it needs in order to reach $v$ would not be enough to actually
CHAPTER 5. CBTC IMPLEMENTATION

reach it. In this, the reconfiguration algorithm of the Maintenance phase will amend it and, anyway, there is no other way to keep track of the minimum power needed to reach another node. Finally, on reception of a HELLO message and when the sender node is already on Neighbours, the asymmetric flag is left unchanged whether it is set or not.

Whether the node $v$ is in Neighbours or not, we immediately issue an ACK message afterwards. We will explain this further in the next section.

ACK message

<table>
<thead>
<tr>
<th>Message Type Id</th>
<th>Transmission power level</th>
<th>Position</th>
</tr>
</thead>
</table>

Figure 5.2: ACK message

Assuming no malfunctioning conditions, a node $u$ receives an ACK message from $v$ only if $u$ has previously issued a HELLO message and $v$ has received it. That is, an ACK message is only a confirmation of the reception of a HELLO message by one neighbour. The way it works is pretty simple. As briefly explained in the previous section, when node $v$ receives a HELLO message from $u$, it automatically sends an ACK message back to $u$; but previously, it adjusts the power to the same used to send the HELLO message (as it is included in the message).

This adjustment is necessary because there are two states in which the transmission power would not be enough for a message to be received:

- if node $v$ is in the Discovery phase but in a previous state than $u$ so it is transmitting with less power.
- if node $v$ is in the Maintenance phase but has reached its state with less power than $u$ is currently using.

On the other hand, when $u$ receives the ACK from $v$ we follow the procedure explained in the previous section about storing the information in Neighbours. With the difference that now it is an ACK message so the flag gets updated accordingly. After that, it is time (and actually, it is the only time we do it) to check if there are no gaps bigger than $\alpha$ between $u$’s neighbours. If there are not such gaps, $u$ enters to the Maintenance phase (previous to that, the applicable optimizations are performed).

ASYMMETRIC message

<table>
<thead>
<tr>
<th>Message Type Id</th>
</tr>
</thead>
</table>

Figure 5.3: ASYMMETRIC message

The only reason to be of asymmetric messages is to implement the second optimization: asymmetric edge removal. The only time they are sent by a
node \( (u) \) is during the execution of this optimization, in the transition from the Discovery to the Maintenance phase.

Asymmetric edge removal is implemented as follows. Remember that in Neighbours \( u \) the node has stored all of its neighbours including a flag that indicates if a node \( v \) is asymmetric to us or not, that is: \( u \) has received one or more HELLO messages from \( v \) but has not received any ACK message from it. So, when the time of this optimization comes, \( u \) sends an ASYMMETRIC message to all the remaining nodes in Neighbours that are still flagged as asymmetric.

The way it works for the node receiving the message is not what could be expected. When \( v \) receives an ASYMMETRIC message from \( u \) it does not immediately delete \( u \) from its own Neighbours. Instead, \( v \) stores \( u \) in a different array whose purpose is to store the IDs of the neighbours that have deleted \( v \) from their respective Neighbours array. The effective deletion of these nodes is in the beginning of the transition to the Maintenance Phase, just after the positive answer to the check_gap test and before the optimizations. The motivation behind this behaviour is that, if node \( v \) deletes \( u \) from Neighbours immediately after receiving, it is highly probable that it gets re-added shortly after and thus making the optimization useless. This probable re-addition is caused by the fact that a node \( u \) in the Maintenance Phase still responds with an ACK (you will find more on that in section 5.5.1) and that \( v \), as it is still in the Discovery Phase, will issue a HELLO message in the next iteration of the timer of this phase and this will cause that \( u \) goes into \( v \)'s Neighbours array again.

NDP message

In the Discovery Phase, the nodes neither send NDP messages nor do anything on the reception of them: when an NDP message is received while in the Discovery Phase, the node simply drops it.

5.4 Transition

The transition from the Discovery Phase to the Maintenance Phase starts when the check_gap test succeeds and ends when the timer of the Maintenance Phase is activated. In the middle, the applicable optimizations are performed.

5.4.1 The check_gap test

Every time an ACK message is received, CBTC checks if the gap still exists. When checking for the existence of the gap, the asymmetric neighbours, although they are in the Neighbours array, are not taken into account. They are not considered for the same reason that, on reception of a HELLO message, this test is not done. So what the routine does is to check that the angle difference between every consecutive node is not bigger than \( \alpha \).

```c
check_gap(){
  size_type first, i, j;
  if(neighbours.size() < 2)
    return true;
```
6
for ( first = 0; first < neighbours.size(); ++first )
   if (!neighbours[first].asymmetric)
      break;
if(first >= neighbours.size() - 1)
   return true;

j = first;
for ( i = first + 1; i < neighbours.size(); ++i ){
   if (!neighbours[i].asymmetric) {
      if(neighbours[i].angle - neighbours[j].angle > alpha)
         return true;
      j = i;
   }
}
return PI * 2.0 + neighbours[first].angle - neighbours[j].angle >
alpha;

If the test is successful – that is, if there is no gap – the first phase is
considered done and so the transition to the second phase starts. As explained
before, if the node has reached the maximum transmission power and the gap
still exists, the first is also finished, and additionally that node is considered a
boundary node.

5.4.2 Optimizations

Once in the transition, there is a step that needs to be done prior to the op-
timizations. It is not really needed, but it allows us to reduce future message
transmissions and also saves time in the Maintenance phase. All of the current
neighbours of the node are copied to the NDP array that will contain the infor-
mation needed by the Neighbour Discovery Protocol. Because if a node already
knows the minimum power needed to reach a certain neighbour, it will not have
to exchange future messages to rediscover it again; especially since in the Man-
tenance Phase is not as straightforward as in the Discovery Phase. What is
being done here is to anticipate something that is highly probable that it will
happen in the immediate future.

Shrinkback operation

The shrinkback operation is the first optimization done. It basically checks
that, for every level of transmission power (starting from the highest), if every
neighbour in this level has another two neighbours in an inferior level that are at
most \( \alpha \) distance of each other surrounding it. This optimization is only applied
to the boundary nodes.

```c++
shrinkback()
{
   if(neighbours.size() < 3)
      return;
   int p_threshold;
   size_type i, j, k;
...
for (p_threshold = (Power::MAX).to_ratio();
    p_threshold > (Power::MIN).to_ratio();
    p_threshold--) {
    for (i = 0; i < neighbours.size(); ++i){
        if (neighbours[i].power < p_threshold)
            break;
    } //There is no one or only one
    if (i >= neighbours.size() - 1)
        return;
    k = i;
    j = i + 1;
    for (; j < neighbours.size(); ++j){
        if (neighbours[j].power < p_threshold) {
            if (j - k > 1 &&
                neighbours[j].angle - neighbours[k].angle > alpha)
                return;
            k = j;
        }
    } //There is only one
    if (k == i)
        return;
    if( !(i == 0 && k == neighbours.size() - 1) &&
        PI * 2 + neighbours[i].angle - neighbours[k].angle > alpha){
        return;
    }
    //delete same level
    neighbours.delete_by_power(p_threshold);
}

Asymmetric edge removal

This is actually the most simple of the optimizations: the only thing needed is to iterate Neighbours and for every asymmetric neighbour found, the node $u$ sends to that specific neighbour $v$ a special message and just after that $u$ deletes $v$ from Neighbours.

When $v$ receives that message it can be whether in the first phase or in the second. If it is in the second, it will drop the message, as we will explain later in 5.5.1. However, if node $v$ happens to be in the first phase, it will not delete immediately $u$ from its own Neighbours. The problem with this behaviour is that a node in the first phase will continue issuing HELLO messages and a node in the second will continue ACKing them; so if the node that receives the ASYMMETRIC message deletes immediately that node at the next HELLO broadcast, with its respective ACK, node $v$ will include $u$ again, and thus making the asymmetric removal useless. What node $v$ actually does is to store $u$ in a special array and, when doing the transition, it removes from its Neighbours all the nodes stored in that array.
Asymmetric edge removal

```cpp
// Asymmetric edge removal
if (alpha <= 2 * M_PI / 3) {
    // Nodes told to be deleted
    for (i = 0; i < neighbours.ATR.size(); ++i) {
        neighbours.delete_by_id(neighbours.ATR[i]);
    }
    neighbours.ATR.clear();
}

// Tell nodes to delete self
for (i = 0; i < neighbours.size(); ++i) {
    if (neighbours[i].asymmetric) {
        Radio::send(os(),
                    neighbours[i].id,
                    TopologyMessage::ASYMMETRIC_SIZE,
                    (uint8_t*) &asymmetricMessage);
        neighbours.delete_by_index(i);
    }
}
}
}
```

Pairwise edge removal

This last optimization is very similar to the first: for every level of transmission power, starting from the highest and going down, it checks if all the neighbours at this level can be removed. We keep all the neighbours from the level where the first node found that it could not be removed and all the neighbours below that level. However, in this case, the way to remove a node is if there is another node from that level or below that is at most \( \pi / 3 \) distance from it.

```cpp
for (p_threshold = (Power::MAX).to_ratio();
p_threshold > (Power::MIN).to_ratio();
p_threshold--) {
    for (fst_below = 0; fst_below < (int)neighbours.size(); ++fst_below) {
        if (neighbours[fst_below].power < p_threshold)
            break;
    }
    if (fst_below == (int)neighbours.size())
        return;
    i = fst_below + 1;
    last_current = -1;
    last_below = fst_below;
    for (; i < (int)neighbours.size(); ++i) {
        // Node on the power level to be deleted
        if (neighbours[i].power == p_threshold) {
            if (last_current == -1) {
                if (neighbours[i].angle - neighbours[last_below].angle > PI / 3)
                    last_current = i;
            }
        }
    }
}
```
CHAPTER 5. CBTC IMPLEMENTATION

25 //Otherwise node with a power below power to be deleted
26 else {
27     if(last_current != -1) {
28         for(j = last_current; j < i; j++) {
29             if(neighbours[i].angle - neighbours[j].angle > PI / 3)
30                 break;
31         }
32         if (j < i)
33             return;
34         else
35             last_current = -1;
36     }
37     last_below = i;
38 }
39 }
40 }
41 //There is just one nodes with power below power to be deleted
42 if(last_below == fst_below){
43     pairwise_one_node(fst_below, p_threshold);
44     return;
45 }
46 }
47 //If all nodes at the end of the vector with power to be deleted
48 //can be deleted
49 if(last_current == -1) {
50     angle = neighbours[last_below].angle - 2 * PI;
51     for(i = 0; i < fst_below; i++){
52         if(neighbours[i].angle - angle > PI / 3)
53             break;
54     }
55 }
56 //Otherwise, if there are some nodes with power to be deleted
57 //after the last node with power below
58 else {
59     angle = neighbours[fst_below].angle + 2 * PI;
60     for(i = last_current; i < (int)neighbours.size(); i++){
61         if(angle - neighbours[i].angle > PI / 3)
62             return;
63     }
64     i = 0;
65 }
66 }
67 for(; i < fst_below; i++){
68     if(neighbours[fst_below].angle - neighbours[i].angle > PI / 3)
69         return;
70 }
71 }
72 neighbours.delete_by_power(p_threshold);
73 }

End of transition

To end the transition, first we notify the upper algorithms that we are ready – a
mandatory step for all topology algorithms in the Wiselib. Then we change all
the asymmetric neighbour survivors (none if $\alpha < 2\pi/3$) to symmetric, as in the
Maintenance Phase we make no distinctions between symmetric and asymmetric
neighbours. Finally we start the timer of the Maintenance Phase.

1 TopologyBase<OsModel>::notify_listeners();
2
for ( i = 0; i < neighbours.size(); ++i )
    neighbours[i].asymmetric = false;

Timer::template set_timer<self_type, &self_type::timer_elapsed_second_phase>(
    os(), work_period_2, this, 0 );

5.5 The Maintenance phase

The timer of this phase also broadcasts a message, this time, an NDP message. In this phase, what really matters is the transmission and reception of these messages, as they are mostly the only source of interaction with the other nodes while in this phase. The paper [7] does not clarify what to do on the reception of the other kind of messages in this phase. This implementation tries to be as pure as possible in the sense that it does the minimum with the rest of the messages and in the following sections there are the actions done and the reasoning behind them.

Besides issuing an NDP message in every iteration, the timer in this phase checks, every certain \( r \) number of iterations, how many beacons has received of every neighbour. If the number of beacons is below \( t \) threshold, then this neighbour is removed. Both \( r \) and \( t \) are configurable. As commented before, in this phase we use the NDP array. The NDP array contains a structure with four fields: the node id, the minimum power needed to reach it, a boolean to determine if has just appeared and a counter for the NDP itself.

5.5.1 Messages (in the Maintenance Phase)

In these phase, some assumptions have been made on how to proceed when HELLO, ACK and ASYMMETRIC messages are received, since it is not specified by [7]. They are explained in their respective sections.

HELLO message

A node \( u \) in the second phase does not transmit HELLO messages. It is very common to receive a HELLO message while in this phase, as all the neighbours in the first phase are issuing them. Node \( u \) will answer with an ACK message upon reception. It would be possible not to answer and the algorithm would continue to work correctly. But what would happen is that the node issuing the HELLO messages would go all the way until it reached the maximum transmission power and then it would become a boundary node. After that it would start using the NDP messages it received from the nodes that previously did not answer to its HELLO messages. Then gradually it would be aware of its neighbours finally reaching its stable state. So basically answering with an ACK message in this stage is a shortcut to a situation that it will happen anyway.

Additionally, when a node receives a HELLO message in the Maintenance Phase, it will add it to its NDP array if it was not there already.

ACK message

The only ACK message a node \( u \) will receive from a node \( v \) are the ACK messages generated in response to the last HELLO message issued by \( v \) before...
reaching the Maintenance Phase. Anyway, the reception of such messages in
this state raises a similar situation as when receiving a HELLO message: act
like in the first phase or not. In this case we have decided to just drop the
message. The node \( v \) could be added to the Neighbours just as in the first
phase but it is going to be added within the NDP mechanism regular operation
anyway.

Also, as with the HELLO message, node \( v \) gets added to NDP if it is not
there already.

**ASYMMETRIC message**

As mentioned before, in this phase, we make no differences between symmetric
and asymmetric nodes, so we just drop the message. We do not even put it in
NDP because ASYMMETRIC messages do not contain power level transmission
information.

**NDP message**

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Transmission power level</th>
<th>Position</th>
</tr>
</thead>
</table>

Figure 5.4: NDP message

The NDP messages are the basis of this phase. They are issued periodically
at every iteration of the timer of this phase to notify its neighbours of its pres-
ence. When \( u \) receives from \( v \) one of these messages, node \( v \) gets included or
updated in the NDP vector just as with ACK and HELLO messages and addition-
ally node \( v \) is included or updated in Neighbours. When it is an addition,
after it, the same shrinkback function from the Shrinkback optimization is
done. On the other hand, if the neighbour was already there and the direction
of \( v \) in respect to \( u \) had changed, the existence of a gap would be checked again.
If there was a gap, then node \( u \) would go back to the Discovery Phase.

### 5.5.2 Transition back to the Discovery Phase

The transition back to the Discovery Phase will be done when either a neighbour
is not detected anymore (by the NDP mechanism) or because one or more
neighbours have changed their position and, after any of these two situations,
a gap is detected. In this case, we clear the NDP array and stop the timer of
the Maintenance Phase and the timer of the Discovery Phase starts again.

### 5.6 Parameters

**Parameter**: \( \alpha \)

**Fuction**:

```cpp
inline void set_alpha(double angle)
```
CHAPTER 5. CBTC IMPLEMENTATION

Parameter: Maximum number of neighbours
Fuction: Not applicable. This parameter is one of the template parameters.
Default: 32
Description: This parameter affects the maximum capacity of the structures of the code (i.e. Neighbours, NDP array). To make the algorithm behave correctly, the number of real neighbours for any node in the network can not be greater than the value of this parameter. A safe way to assign this value is to put the number of nodes minus 1 of the entire network.

Parameter: Start-up time
Fuction:
```c
inline void set_startup_time( millis_t startup_time )
```
Default: 2000
Description: This is the time, in milliseconds, that the node will wait from the moment the enable function is called until the timer of the first phase gets executed for the first time. If it is zero, it will start immediately.

Parameter: Work period of the timer of the Discovery Phase
Fuction:
```c
inline void set_work_period_1( millis_t work_period )
```
Default: 5000
Description: This is the time between iterations of the timer of the Discovery Phase. It should be greater than the time needed to send a message, receive it by another node, and answered with another message. On the other hand, the smaller the parameter is, the less time it will be necessary to get to the Maintenance Phase.

Parameter: Work period of the timer of the Maintenance Phase
Fuction:
```c
inline void set_work_period_2( millis_t work_period )
```
Default: 15000
Description: This is the time between iterations of the timer of the Maintenance Phase. This is the phase where the node will spend most of the time (specially in networks where mobility is little or non existent). So, besides the same time constraints of the timer of the Discovery Phase, it is desired to be as big as possible so less messages will be sent by the same amount of time, thus saving energy.
**Parameter**: Number of iterations in NDP before checking

**Function**: inline void set_NDP_iterations(uint8_t iterations)

**Default**: 5

**Description**: This is the number of times the timer of the Maintenance Phase has to be executed to check that the number of beacons of every neighbour are enough to keep maintaining them as neighbours of the node. Of course this parameter makes sense only in conjunction with the following parameter.

---

**Parameter**: Threshold of lost messages in NDP

**Function**: inline void set_NDP_threshold(uint8_t iterations)

**Default**: 3

**Description**: This is the number of beacons a node has to receive from its neighbours in order to keep them as its neighbours.
Chapter 6

The LSP algorithm

A shortest-path-based algorithm, called Local Shortest Path (LSP), for topology control in wireless multihop networks is described in this chapter. The LSP algorithm was originally proposed in [11], where experiments based in simulations show its good performance with respect to other well-known topology control algorithms. However, the LSP algorithm was never implemented on real sensor networks. We have implemented the algorithm in theWiselib using some of the generic concepts designed for the CBTC algorithm.

Our experiments include simulations on a bigger network’s benchmark set and its testing on a real iSense-based sensor network testbed.

6.1 Introduction

In LSP, each node locally computes the shortest paths connecting itself to nearby nodes based on some link weight function, and then it selects all the second nodes on the shortest paths as its logical neighbours in the final topology. Any energy model can be employed in LSP to design the link weight function whose value aims at representing the power consumption required in the transmission along a link.

Regarding the flexibility in the energy model, one advantage of LSP over all the previous approximations is that it is well suited for heterogenous networks. Traditionally, all the topology algorithms have assumed that a long link consumes more power than a short link, but this assumption is unpractical in heterogenous networks, as demonstrated in [11].

6.2 The LSP algorithm

In order to ease the description of the algorithm, let us first define some common terms. The topology of a wireless network is modeled with each node using its maximal transmission power as an undirected graph $G = (V, E)$ in two dimensional plane, where $V = \{v_1, v_2, ..., v_n\}$ is the set of nodes in the network and $E$ is the set of bidirectional links. This network may be heterogeneous and hence each node $v_i$ may have its own transmission power $p_i$, which can be adjusted by itself. LSP only treats bidirectional links and so discards asymmetric ones.
Therefore, the bidirectional link \((v_i, v_j) \in E\) implies that both \(v_i\) and \(v_j\) are covered by each other.

The physical neighbour set of each node \(v_i\) is defined as

\[
NS_p^i = \{ v_k | (v_i, v_k) \in E(G) \text{ or } (v_k, v_i) \in E(G) \}. \tag{6.1}
\]

The logical neighbour set \(NS_l^i\) is a subset of \(NS_p^i\).

In LSP, each bidirectional link is assigned a weight that can be derived from a certain weight function \(w\). Thus the weight of a link \((v_i, v_j)\) can be expressed by \(w(i, j)\). The weight on each link is used to represent the power consumption required in the transmission along that link, while the path weight is used to represent the sum of all link weights of a path. Therefore, the minimal energy path is defined as the path with minimal path weight among all the paths connecting two given nodes. The computation of \(w(i, j)\) usually relates only to \(v_i\) and \(v_j\), at most to their neighbours. This makes it possible that each node runs LSP only according to the locally collected information.

The algorithm can be divided in the following steps:

1. Link weight calculation
2. Link weight information exchange
3. Local Topology Construction
4. Transmission power adjusting

### 6.2.1 Link weight calculation

Each node in this step locally collects the information needed in the weight calculations for all links associated with it and it calculates link weights.

![Figure 6.1: LSP Step 1](image-url)
CHAPTER 6. THE LSP ALGORITHM

models, the two endpoints of a link \((v_i, v_j)\) may deduce different weights (called unidirectional link weights and denoted as \(\vec{w}(i, j)\) and \(\vec{w}(j, i)\) respectively). In this case, \(v_i\) is able to know \(\vec{w}(j, i)\) and \(v_j\) is also able to know \(\vec{w}(j, i)\) in the following second step by information exchanging. Thus \(w(i, j)\) can be designed as

\[
w(i, j) = (\vec{w}(i, j) + \vec{w}(j, i))/2 \quad (6.2)
\]

In this step it is simply assumed that each node derives the unidirectional link weights.

Example

To illustrate how the algorithm works, we will use the tiny network shown in figure 6.1. To simplify the explanation, the network used is composed of homogeneous nodes. So, all nodes have the same range, all links are bidirectional and \(\vec{w}(i, j) = \vec{w}(j, i)\); that is, it costs the same for both nodes to reach one another.

In figure 6.1, the numbers next to the edges are the cost of the nodes at every end of the edge to transmit to each other. As explained, the purpose of this step in the algorithm is to find out (no matter which method is employed) these values next to the edges. So after the step is performed, the nodes have the information shown in table 6.1.

The maximum range is the one that needs 100 power.

<table>
<thead>
<tr>
<th>Node</th>
<th>Power cost to reach direct neighbours</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a-b: 100, a-c: 44</td>
</tr>
<tr>
<td>b</td>
<td>b-a: 100, b-c: 71, b-d: 31, b-e: 88</td>
</tr>
<tr>
<td>c</td>
<td>c-a: 44, c-b: 71, c-d: 24, c-f: 79</td>
</tr>
<tr>
<td>d</td>
<td>d-b: 31, d-c: 24, d-e: 34, d-f: 31</td>
</tr>
<tr>
<td>e</td>
<td>e-b: 88, e-d: 34, e-f: 13</td>
</tr>
<tr>
<td>f</td>
<td>f-c: 79, f-d: 31, f-e: 13</td>
</tr>
</tbody>
</table>

Table 6.1: The power in abstract units needed by every node to reach its direct neighbours. The bigger the value, the more expensive it will be to reach that neighbour.

6.2.2 Link weight information exchange

In this step, each node \(v_i\) exchanges the unidirectional link weight information \(\vec{w}(i, j)\) derived from the previous step with each of its 1-hop neighbours \(v_j\). It does so by broadcasting at its maximum transmission power \(p_{\text{max}}^i\). After that, \(v_i\) knows all the weights calculated at the nodes in the set of \(\{v_i\} \cup NS^i_p\). Therefore, for any two nodes \(v_x, v_y \in \{v_i\} \cup NS^i_p\), if there is a bidirectional link \((v_x, v_y)\), \(v_i\) can derive \(w(x, y)\) according to equation (6.2). Nevertheless, two aspects should be mentioned:

- \(v_i\) may receive \(\vec{w}(j, i)\) from node \(v_j\) while \(v_j\) is out of its range. In this case, link \((v_j, v_i)\) is not considered by \(v_i\). This it what effectively makes LSP free of asymmetric links.
- \(v_i\) only receives \(\vec{w}(x, y)\) and does not receive \(\vec{w}(y, x)\), which is because \(v_y \notin \{v_i\} \cup NS^i_p\). Thus, link \((v_x, v_y)\) is still not considered.
Of course, the information exchange in this step could be avoided if each node \( v_i \) could obtain the information required to calculate the unidirectional link weights for all the nodes in \( N_{Sp}^i \). Were this possible, each node could do all the weight calculations for all links. Although this approach needs more computation time and stricter preconditions, it reduces the number of information exchanges to one.

<table>
<thead>
<tr>
<th>Node</th>
<th>Accepted</th>
<th>Discarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b-a:100, b-c:71, c-a:44, c-b:71</td>
<td>b-d:31, b-e:88, c-d:24, c-f:79</td>
</tr>
</tbody>
</table>

Table 6.2: Information received in step 2 of LSP by every node about neighbours of neighbours and actions taken on it.

**Example**

Following the example introduced in the previous subsection, in table 6.2 we can see the information received by every node in the network about the neighbours of their neighbours. As we are dealing with an homogeneous network, the only information is discarded is because of the second reason described in this subsection: the node \( v_i \) receives \( \vec{w}(x,y) \) but does not receive \( \vec{w}(y,x) \).

**6.2.3 Local topology construction**

In this step, each node \( v_i \) computes the local shortest path connecting it to every node \( v_j \in N_{Sp}^i \) according to the derived links weights. In the original paper [11], the Dijkstra’s algorithm is recommended as a way to solve this step or the Bellman-Ford algorithm if negative link weights are considered[1]. It is important to note, though, that both algorithms provide more information than what is needed: LSP is not interested in the total minimum weight to reach a certain node, but just the path. In fact, it is not even interested in the whole path, but just on those neighbours that allow to reach all the others in \( N_{Sp}^i \) at minimum cost (i.e., weight). We express that formally in the following section.

---

1Note that in some networks, transmissions along a link may reduce the power consumptions of the nearby nodes by using some strategy such as turning their radios off. In this case, the link may have a negative weight.
CHAPTER 6. THE LSP ALGORITHM

Denote the local shortest path connecting $v_i$ to $v_j \in NS_p$ as

$$\text{path}^{i,j} = (v_{p_0}^{i,j} = v_i, v_{p_1}^{i,j}, ..., v_{p_{n-1}}^{i,j}, v_{p_n}^{i,j} = v_j)$$ (6.3)

where $v_{p_m} \in NS_p, m = 1, 2, ..., n$ and $n \geq 1$. Then the logical neighbour set $NS^i$ of node $v_i$ can be represented by

$$NS^i = \{v_{p_x}^i | v_x \in NS_p^i\}.$$ (6.4)

That is, all the second nodes on the shortest paths from $v_i$ to all other nodes in $NS_p$ compose the logical neighbour set of node $v_i$.

Note that the path is bidirectional, since every link on the path is also bidirectional. However, the path $\text{path}^{i,j}$ is not the reverse of the path $\text{path}^{j,i}$, since $NS_p^i \neq NS_p^j$. The network topology under LSP is all the nodes in $V$ and their individually perceived logical neighbour relations.

Figure 6.2: LSP Step 3

Example

Continuing with the example of this chapter now we have all the information to build the logical neighbour set. First we apply the Dijkstra’s algorithm: in table 6.3, we can see the results for every node in the network. Note that while we see the whole picture, the nodes only know a subset of the network (e.g. node $v_a$ knows about nodes $v_b$ and $v_c$; node $v_f$ knows about nodes $v_c$, $v_d$ and $v_e$; and so on), so they do not know how to reach every node in the network and apply the Dijkstra’s algorithm only to the known subset.

Finally, in bold in the table 6.3 we have the aforementioned second nodes on the shortest paths. These nodes are what constitute the logical neighbour set of every node. For example, node $v_a$ has nodes $v_b$ and $v_c$ as the second nodes in its two paths, so it keeps them, and in that case the physical and the logical sets remain the same. But in the case of node $v_c$, the second nodes are $v_a$ and $v_d$ (although three times, but it does not matter) so its logical neighbour set becomes $\{a, d\}$ while its physical neighbour set is $\{a, b, d, f\}$.

In figure 6.2 we can see the resultant network composed only of the logical neighbours sets of every node.
6.2.4 Transmission power adjustment

After each node $v_i$ derives its logical neighbour set $NS_i$, it computes the minimal transmission power $p_{adj}^i$ required to cover all of its logical neighbours, and then it adjusts its transmission power to be $p_{adj}^i$. The computation of $p_{adj}^i$ relates to the energy model and may require information such as the distances from $v_i$ to its logical neighbours and their power consumption properties.

Example

To conclude, every node in the example should adjust its power to the minimum needed in order to reach its farthest neighbour of its logical neighbour set. In the case of node $v_a$, it would remain transmitting at its maximal power (100) because of node $v_b$. In the case of node $v_c$, it should adjust to 44 because of its farthest neighbour $v_a$. 

<table>
<thead>
<tr>
<th>Node</th>
<th>Dest.</th>
<th>Path</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a - b</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>a - c</td>
<td>44</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b - a</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>b - c - d</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>b - d</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>b - d - e</td>
<td>65</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>c - a</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>c - d - b</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>c - d</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>c - d - f</td>
<td>55</td>
</tr>
<tr>
<td>d</td>
<td>b</td>
<td>d - b</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>d - c</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>d - e</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>d - f</td>
<td>31</td>
</tr>
<tr>
<td>e</td>
<td>b</td>
<td>e - d - b</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>e - d</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td>e - f</td>
<td>13</td>
</tr>
<tr>
<td>f</td>
<td>c</td>
<td>f - d - c</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>f - d</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>f - e</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 6.3: The result of applying the Dijkstra’s algorithm in every node in the network.
Chapter 7

LSP implementation

In this chapter, we will detail the implementation done of the LSP protocol (as described in chapter 6) for the Wiselib framework.

7.1 Introduction

The implementation done is roughly a direct translation from the four steps described in chapter 6. However, there are certain issues that arise when translating those steps to code that are worth mentioning. The issues come from the fact that there are some time constraints inherent to the LSP design. The problem arises from the second step; that is, the exchange of information. In this step, the nodes broadcast the information they have collected in the previous step. This information (when received) is vital to be able to continue the execution of the algorithm. So the problem is that if some node misses this information exchange for whatever reason (it is added later to the network, it executes later the algorithm), then it will not be able to compute its logical neighbour set, or it will simply compute it wrong.

This implementation tries to relax this constraint to a certain degree and indeed does the algorithm more flexible (so the nodes do not have to start at the same precise time) but, because of the design presented in [11], there will always be this limitation. To that matter, two aspects should be mentioned. First, LSP could be easily extended to overcome this situation. And second, and maybe the most important, the authors in the original paper [11] state the information exchange is obviously avoidable if each node can collect the same information by themselves, without the interaction with the other nodes. If that was the case, this whole problem would not exist anymore. However the authors do not say with what mechanism should this be accomplished.

This relaxation works this way: the first two steps (link weight calculation and information exchange) are performed, then a certain amount of time is waited (user defined by a parameter) to allow all the nodes to reach this point. After that, the rest of the steps (such as links discarded by lack of information) are performed. It is important to note, though, that some sort of synchronization mechanism is needed either way.
7.1.1 Data Structures

As mentioned before, LSP is more a framework than an algorithm in the sense that it specifies what should be accomplished but not how to do it. So strictly speaking there are no data structures required (imposed) by LSP. That said, at least one structure will be needed to represent the weighted graph.

In the following snippet of our implementation, we can see the two structures we have used: NMatrix and NMapping. We use them to represent the weighted graph.

```c
int NMatrix[MAX_NEIGHBORS][MAX_NEIGHBORS];
Neighbors NMapping;
```

NMatrix is a matrix of dimensions MAX_NEIGHBORS x MAX_NEIGHBORS, where NMatrix[i][j] represents the cost for node vi to send a message to vj. Of course, there has to be a way to map these indices (i and j) to the real IDs of the nodes and that is the function of the vector NMapping, that simply stores node IDs. The position of a node ID in NMapping is the index of that node in NMatrix. Figure 7.1 is a graphical representation of this implementation. It is remarked what would be the cost to transmit from node 0x76 to node 0xf4.

![Figure 7.1: NMapping and NMatrix](image)

Additionally, before starting to fill NMapping (and NMatrix accordingly) with the information gathered in step one (subsection 7.2.1), we add the node’s own ID to NMapping in the first slot; so we assure that in row zero there will always be the information regarding the node itself. In the example of figure 7.1, that NMapping and NMatrix would correspond to the node of ID 0x1e.

7.2 The four steps

In this section, we explain all the relevant parts of the code of this implementation. The section is structured in the same way as in the previous chapter, with the four steps described in it.

7.2.1 Link weight calculation

In this step, each node has to determine the cost (that is, the link weight) to reach its physical neighbours; in other words, the neighbours that a node can
communicate to when transmitting at maximum power.

No method is described in the original paper [11] to calculate the link weight. We have employed a slightly modified version of the method introduced by CBTC [16], as described in section 4.2. The method is as follows. A node \( v_i \) starts broadcasting a HELLO message at the lowest possible level of transmission and waits for its neighbours to reply with an ACK message. Then it broadcasts this HELLO message at every level of transmission that the node is capable of transmitting. By transmitting gradually with greater transmission power, the node is able to discover its whole physical neighbourhood and to know the transmission power needed to communicate with each of its neighbours. The first time node \( v_i \) receives the reply from a neighbour node \( v_j \), it derives the weight of the link \((v_i, v_j)\) from the power value.

The HELLO message includes the power used to transmit it in itself. The only motivation to include that information in the message is to be able to differentiate between HELLO messages. When node \( v_j \) receives a HELLO message issued by node \( v_i \), node \( v_j \) answers back with an ACK message that includes the same value contained in the HELLO message so node \( v_i \) is able to determine which HELLO message is referring to. This behaviour could be avoided if it was assumed that all neighbours of node \( v_i \) in position to answer to a certain HELLO message were able to answer in less time that the time elapsed between two consecutive HELLO messages. In that case, the first time discovered nodes would be tagged with the power used to transmit the latest HELLO message.

There are two main differences between the mechanism used by CBTC and this particular LSP implementation. The first one is that CBTC assumes (and requires) an homogeneous network so the information about the power used stored in the HELLO messages is used by the replying neighbours to determine what power is needed in order to answer back. LSP on the other hand is designed to work in heterogeneous networks, so no assumption can be made regarding the power needed to reach \( v_i \) (the one that issued the HELLO message). The solution to that is to transmit at maximum power the ACK messages. The second difference is that in CBTC (as explained in section 4.2) if a certain condition is met, node \( v_i \) stops transmitting HELLO messages and in our LSP implementation node \( v_i \) only stops when it reaches the maximum transmission power level. It is this way because, contrary to CBTC, LSP wants to know this information about all its physical neighbour set.

Finally, although LSP is designed to work in heterogeneous networks, there has to be some homogeneous way to represent the link weight across the different type of nodes. For example, let’s assume that in the same network there is some hardware that represents its transmission power levels in range from 1 to 10 and there is another hardware whose range is formed with values from -32 to -4. In that scenario and after the information exchange of the link weight, the links of the latter would always be considered to have less cost than the former. Fortunately, to resolve this issue nothing has been done because in that regard the Wiselib provides a uniform interface that hides the differences between different platforms.

The following piece of the implementation illustrates the mechanism described above. It is a timer programmed to be triggered when the algorithm starts to run. It reschedules itself in the lines 27-29 and the only way to stop that is when the condition at line 4 is met. This condition will eventually be met, because as can be seen in line 15, the variable \texttt{selfpower} is incremented at ev-
ery iteration of the timer, except for the first iteration. The variable `selfpower` is used to track at what power level the node is transmitting in that moment so that is why in line 19 the radio of the node is set to its value. As it can be seen from the code, the node running it will issue a HELLO message at each power level and will wait `work_period_` milliseconds between each iteration. Finally, when the condition in line 4 is met (and that means that in the previous iteration of the timer the HELLO message was transmitted at maximum power), two things are done: step two, that is not shown (as denoted by the suspension points in line 5) and detailed in the next subsection 7.2.2 and the timer that takes care of step three and four (explained in subsections 7.2.3 and 7.2.3 respectively) is scheduled to trigger after `wait_time_` milliseconds.

Both `work_period_` and `wait_time_` are parameters the user can change. See section 7.3.

```c
void
timer1( void* userdata )
{
  if(selfpower == Power::MAX) {
    ...
    Timer::template set_timer<self_type, &self_type::timer2>(
        os(), wait_time_, this, 0
    );
    return;
  }

  if(just_started)
    just_started = false;
  else
    selfpower++;

  helloMessage.set_power(selfpower.to_ratio());

  Radio::set_power(os(), selfpower);
  Radio::send(
    os(),
    Radio::BROADCAST_ADDRESS,
    LSPMessage::HELLO_SIZE,
    (uint8_t*)&helloMessage
  );

  Timer::template set_timer<self_type, &self_type::timer1>(
    os(), work_period_, this, 0
  );
}
```

7.2.2 Link weight information exchange

The purpose of this is step from the node point of view is to notify all of its physical neighbours about the information gathered in the previous step; that is, the link weight.

The information exchange is pretty straightforward: at maximum transmission power, every node sends a message containing the link weight information of each of its neighbours from whom it received an ACK message.

The following fragment of code is responsible for that. This code is the one that is omitted in the previous subsection (the suspension points). We do not have to set the transmission power to the maximum level because we
are guaranteed to be doing so already (because of the condition at line 4 of the previous code snippet). So we just iterate the first row (as we previously explained, this is the row where the information gathered by the node executing the code stores the information about its neighbours) and for every neighbour we have information on (line 2), we proceed to send a message to all of our neighbours (lines 5-12), an INFORMATION message with its respective ID and weight.

```
for(unsigned int i = 1; i < MAX_NEIGHBORS; i++){
    if(NMatrix[0][i] == WEIGHT_UNKNOWN)
        continue;
    infoMessage.set_other_id(NMapping[i]);
    infoMessage.set_weight(NMatrix[0][i]);
    Radio::send(
        os(),
        Radio::BROADCAST_ADDRESS,
        LSPMessage::INFO_SIZE,
        (uint8_t*)&infoMessage
    );
}
```

### 7.2.3 Local topology construction

This step, local topology construction, as described in the original paper or in subsection 6.2.3, only contains the execution of an algorithm (Dijkstra’s or Bellman-Ford in [11]) in order to determine the logical neighbour set of the node. However, as explained in subsection 6.2.2, it has to be performed some form of purge of the information received prior to executing that algorithm.

So two actions are needed in this phase. The first one, as described in subsection 6.2.2, is to remove links which are not considered (lines 3 and 4 of the following piece of code). And while we are at it, we complete the computation of the bidirectional value of $w$, as described in equation 6.2 (lines 6 and 7), since we are traversing the matrix anyway.

```
for(i = 0; i < NMapping.size(); i++){
    for(j = i + 1; j<NMapping.size(); j++) {
        if(NMatrix[i][j] == WEIGHT_UNKNOWN)
            NMatrix[j][i] = WEIGHT_UNKNOWN;
        else if(NMatrix[j][i] == -1) NMatrix[i][j] = WEIGHT_UNKNOWN;
        else {
            NMatrix[i][j] = (NMatrix[i][j] + NMatrix[j][i]) / 2;
            NMatrix[j][i] = NMatrix[i][j];
        }
    }
}
```

The second of these actions is to execute, at every node $v_i$, the Dijkstra’s algorithm in order to compute the shortest paths from $v_i$ to all the other nodes in the network. Below is our code (a classic implementation of the Dijkstra’s algorithm). It takes the weighted graph (in our code, represented as NMatrix) as input and assumes that each vertex of the graph is mapped to a unique ID in the range $0..N - 1$ being $N$ the number of vertices of the graph. This is accomplished by the NMapping vector. The other input parameter is from which starting node these shortest paths are to be computed. In lines 16 and 17 of the following code is where this input parameter is used. We want the node
running the code to be the node to be used as starting point and that is why it is initialized to the zero index as this will always be, as explained before in subsection 7.1.1, the index of the node running the code.

```cpp
dijkstra()
{
    vector_static<OsModel, bool, MAX_NEIGHBORS> Visited;
    vector_static<OsModel, int, MAX_NEIGHBORS> Parents;
    vector_static<OsModel, int, MAX_NEIGHBORS> Distances;
    priority_queue<OsModel, arc, MAX_NEIGHBORS * MAX_NEIGHBORS> PQ;

    unsigned int n = NMapping.size();

    for(unsigned int i = 0; i < n; i++) {
        Visited.push_back(false);
        Parents.push_back(-1);
        Distances.push_back(std::numeric_limits<int>::infinity());
    }

    Distances[0] = 0;
    PQ.push(arc(-1,0,0));

    while (not PQ.empty()) {
        arc a = PQ.pop();
        unsigned int v = a.v;
        if (not Visited[v]) {
            Visited[v] = true;
            for(unsigned int i = 0; i < n; i++)
                if (v == i or NMatrix[v][i] == WEIGHT_UNKNOWN)
                    continue;
            if (not Visited[i]) {
                if (Distances[i] > Distances[v] + NMatrix[v][i]) {
                    Distances[i] = Distances[v] + NMatrix[v][i];
                    Parents[i] = v;
                    PQ.push(arc(v,i,Distances[i]));
                }
            }
        }
    }

    for(int i = 0; i < Parents.size(); i++)
        if(Parents[i] == 0)
            N.push_back(NMapping[i]);
```

Its output are the arrays `Parents` and `Distances`, but we are just interested in the former. In `Parents`, the algorithm outputs at each position (each position representing a node in the graph) what other node is immediately before in the shortest path. Always from the point of view of the node running the code; that is, the node mapped to the zero position. To show this more clearly, let’s take a look at table 7.1.

We can know from the table 7.1 that nodes 0xf4 and 0x2c (nodes with index 3 and 4 respectively) communicate directly to node 0x1e (the node where the algorithm is being executed). That is, the shortest path does not include any other nodes. On the other hand, node 0x76 has as parent node 0xf4, that we just said directly communicates with node 0x1e. Similarly, node 0x09 has as parent node 0x76, that in turn has as parent 0xf4, that finally communicates with the original node.
CHAPTER 7. LSP IMPLEMENTATION

### Table 7.1: An example of the output of Dijkstra’s algorithm regarding the array Parents and its meaning

<table>
<thead>
<tr>
<th>Node</th>
<th>Index</th>
<th>Parent</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1e</td>
<td>0</td>
<td>-</td>
<td>the node itself</td>
</tr>
<tr>
<td>0x09</td>
<td>1</td>
<td>2</td>
<td>1 - 2 - 3 - 0</td>
</tr>
<tr>
<td>0x76</td>
<td>2</td>
<td>3</td>
<td>2 - 3 - 0</td>
</tr>
<tr>
<td>0xf4</td>
<td>3</td>
<td>0</td>
<td>3 - 0</td>
</tr>
<tr>
<td>0x2c</td>
<td>4</td>
<td>0</td>
<td>4 - 0</td>
</tr>
</tbody>
</table>

7.2.4 Transmission power adjustment

The final step is to adjust the power to only reach the neighbours that are the first nodes in one of the shortest paths. That is, the neighbours that have as parent the original node. And we can see that these nodes are the ones with 0 as parent in table 7.1.

7.3 Parameters

This section shows all the parameters that the user of the algorithm can define in order to adjust the algorithm to its needs, or the information the user needs to supply to the algorithm so it can behave correctly.

- **Parameter**: Maximum number of neighbours
  - **Function**: Not applicable. This parameter is one of the template parameters.
  - **Default**: 32
  - **Description**: This parameter affects the maximum capacity of the structures of the code (e.g. NMatrix, NMMapping). To make the algorithm behave correctly, the number of real neighbours for any node in the network can not be greater than the value of this parameter.

- **Parameter**: Start-up time
  - **Function**: inline void set_startup_time( millis_t startup_time )
  - **Default**: 2000 milliseconds
  - **Description**: This parameter defines the time, in milliseconds, that the node will wait from when the function `enable` gets called until the timer gets executed for the first time. If it is 0, it will start immediately.

- **Parameter**: Work period of the timer
  - **Function**: inline void set_work_period( millis_t work_period )
Default: 5000 milliseconds

Description: This is the time between iterations of the timer used in the step called Link Weight Calculation, described in subsection 7.2.1. It should be greater than the time to send a message, received by another node, and answered with another message. On the other hand, the smaller the work period of the timer, the less time LSP will finish its execution.

Parameter: Time to wait for the other nodes to proceed
Function:

```cpp
inline void set_work_period( millis_t wait_time )
```

Default: 10000 milliseconds

Description: This is the time a node waits after step 2 in order to proceed to the following levels.
Chapter 8

Results

In this chapter, the analysis of the results of the simulation for both algorithms in different scenarios is presented. A little analysis of the memory requirements for both algorithms is also discussed.

8.1 Scenarios

The two algorithms have been tested with Shawn\textsuperscript{[10]}. Shawn, although a simulator, is one of the supported backends of the Wiselib. For CBTC the two $\alpha$ recommended values (the one in the original paper and the one in the later revision) have been tested, and for each $\alpha$, all the applicable optimizations. That is, when $\alpha$ is 5/6, the ‘Asymmetric edge removal’ optimization cannot be applied. It is also important to recall that the optimizations are not applied independently; they are actually accumulative.

To show the different reactions with varying degrees of node densities, four configurations have been chosen:

1. Low: Width: 5, Height: 5, Range: 2, Nodes: 25, Density: 2.5
2. Medium Low: Width: 5, Height: 5, Range: 2, Nodes: 100, Density: 10
3. Medium High: Width: 5, Height: 5, Range: 2, Nodes: 175, Density: 17.5

and each algorithm configuration has been combined with these density configurations.

8.2 Measure method

Each possible configuration has been run 20 times. In order that the randomization of the tests (essentially positions of the nodes) does not affect positively or negatively the different configurations, the seed of the randomization has been kept (as Shawn allows it) and, while different in every run, it was the same throughout the configurations.

Additionally, to keep the results simple, if a given seed produces a network without connectivity when all the nodes are transmitting at full power, this seed
is discarded. For the sake of clarity, all the configurations have kept connectivity so that is why it is not shown in the tables (whether the resultant network has preserved connectivity or not).

### 8.3 Neighbours per node

In the next figure 8.1 and in table B.1 we can see how many neighbours are left in each node after the algorithms have been run. It is important to remember that, as stated in [16], having less neighbours does not mean to be more power efficient. Actually the number of hops can increase and thus effectively becoming less power efficient.

To better understand the chart in each density configuration, the 100% equals the number of neighbours when no topology control is applied. That is, all the neighbours of a node covered by its transmission range.

As we will see in this and in most of the following figures, LSP and CBTC when all optimizations are applied (both for 2/3 and 5/6) behave similarly. And in the other side of chart there is CBTC with $\alpha = 2/3$ and no optimizations; the rest being in the middle. Because of the different philosophy of LSP regarding the purge of unnecessary neighbours (that is, neighbours not needed to satisfy connectivity), it has always less neighbours than the rest, although at the end this difference is minimum.

In any case, the tendency is the same for all: the more dense the network is, the bigger the difference with no topology control, although all of them seem to independently converge to some minimum value.
8.4 Distances for every two pairs of nodes

In the following figure 8.2 and in table B.2, the number of hops for every possible route in the network is shown. That is, given any pair of nodes, the amount nodes a message has to go through to reach its destination. This is actually a good measure of power efficiency. But of course there is a tradeoff between transmission power and distance: if a node reduces its transmission power, it will inevitably reach less nodes than before, so the number of hops will increase.

Figure 8.2: Distances for every two pairs of nodes

Y: Average distance of two nodes relative to no TC
X: Total number of nodes in the network

What is shown in this chart is the average distance increase respective to when no topology control is applied. That is, 1 in the Y axis represents the average distance for each density configuration. This distance is computed assuming a perfect routing algorithm or in other words the minimum distance between two nodes. For example, LSP in the highest density network, in average, takes 4 times the steps that would take when no topology control is present.

Again the trend follows the same pattern as in the previous figure, although this time the differences between the three algorithms in the top are much wider respective to the rest.

Lastly, all of them seem to slowly converge to some maximum value.

8.5 Transmission Power

In figure 8.3 (and in more detail in the table B.3), we can see the power nodes are left transmitting when the algorithm is done. In order to understand the values, a little explanation is needed.
In Shawn, a maximum range is set globally (the range value you can see in the density configurations) and after that you can arbitrarily set a node transmission range with value from 0 to this maximum range value. In the wiselib integration, and in order to reassemble real hardware, you can configure how many steps in this range you want to discretize this range and therefore you obtain range/steps level values. The default number of steps is 10 and for these tests we have used that. So finally there are 10 possible levels, from 1 to 10 (as 0 would is no transmission power).

Finally, and because realistically the transmission power growth is not linear, the values you can see in 8.3 have been powered to the 2 to be more representative. In other words, increasing the transmission range $x$ distance is logarithmically less expensive than increasing it to $2x$.

So the values in 8.3 are in the range $[1..100]$, where 1 would be that all nodes are transmitting at minimum power and 100 that all nodes are transmitting at its maximum power (no topology control).

Looking at the figure above, we can see that this time the behaviour is similar again. It is important to note, though, that while both CBTC with all the optimizations enabled follow almost exactly the same trace, LSP does a little worse job than them. This is important because the metric represented in this figure is the most important of all that we have seen up until this point.
8.6 Traffic simulation

In figure 8.4, there are the results of trying to quantificate the cost of each algorithm with some simulated traffic. In a certain way, this metric is the combination of the three previous metrics.

To calculate the values, 300 messages have been sent between two random nodes and the value of the transmission powers (as calculated in 8.3) of these nodes and all the intermediate nodes in the path (the same method as in 8.2 is used) are added.

The values for no topology control are not shown because the difference was so big (three orders of magnitude) that it would render the figure useless. But they can be seen in table B.4.

The values in 8.4 by themselves do not mean much as they are dependent of the number of messages sent in the simulation. If more messages were, sent basically the differences between them would be bigger. What matters is the trend. Actually, for the first time we can see that not only LSP is performing worse than CBTC with all optimizations (as usual), but this time is the worst of them all. The only explanation for this is that LSP produces a network with more hops between nodes (as can be seen in 8.2), so that in the end all the CBTC configurations perform better.

It should be noted that LSP actively dismisses neighbour nodes that it considers not useful but that they are within range. This added to the circumstance that the routing algorithm can dismiss the information regarding the neighbours supplied by LSP (in this case the only thing used would be the transmission
range) and use all the neighbours within range could potentially mitigate this performance. Nevertheless the trends for them all are similar, so it is not performing that bad, just worse than the rest.

### 8.7 Number of messages and estimated cost

The last two charts show two metrics about the power efficiency of the algorithms themselves. In figure 8.5, the average number of messages sent by node is shown. As CBTC with $\alpha = 5/6$ cannot apply the *Asymmetric removal* optimization, and the other two optimizations do not require any message to be sent, there is only one line for it. So for CBTC with $\alpha = 2/3$ there are two lines: one counting the messages needed for *Asymmetric removal* and one without them. Although as seen in the figure, they are statistically irrelevant compared to the other kind of messages. To see the actual differences in table 8.5, there is the raw data.

![Figure 8.5: Average number of messages sent per node](image)

What is significant is that CBTC with $\alpha = 5/6$ consistently sends less messages that with $\alpha = 2/3$. As the bulk of the messages are the ones involved in the discovery phase, this indicates that nodes in the first case complete the $\alpha$ gap condition earlier than in the second case. Of course that is because the condition in the first case is less strict ($5/6 > 2/3$).

For LSP, two lines are shown. The one labeled *LSP* is the current implementation of LSP. The other one represents an alternative implementation where the messages sent after the discovery phase were just one instead of one for
CHAPTER 8. RESULTS

...every neighbour. As we can see, the number of additional messages – although significant – does not change the trend because this time again, the bulk of the messages belong to the discovery phase. Although both LSP and CBTC have the same discovery mechanism implemented, the difference in number of messages is determined by the fact that LSP does not stop until all the transmission power levels are tested. Contrary to CBTC, that when a certain condition is reached, the discovery mechanism stops.

Figure 8.6: Estimated cost of messages sent per node

In the second figure 8.6, an estimation of the costs of the previous messages is shown (and in more detail in table B.6). The formula used to compute it is: for every message $1 + \text{messageSize} \times \text{power}$. Where power is the transmission power of the node (as calculated in 8.3) at the moment the message is sent. The presence of the 1 is to penalize every transmission so it is more expensive to send two messages with say, for example, the IDs of the two neighbours than sending those two positions in one message.

The chart is very similar to the previous one in what respects to trends, with the only remarkable difference being that LSP is less distant to CBTC. That is because the size of the messages of LSP are not as big as those of CBTC.

This time, for every CBTC line there is an additional one included, which does not account for the position of the node carried. That is because in the ideal CBTC implementation, such information would not be needed as with the Angle of arrival technology this information could be deduced. As it can be seen in the figure, it does not make a big difference other than lowering the cost in a linear manner.
CHAPTER 8. RESULTS

8.8 Comparison with the original papers

Finally, in order to see how the implementations perform compared to the results obtained in the papers we have replicated the main test in the second CBTC paper [7].

In this paper, one hundred executions are done distributing the nodes randomly. The size of the area where the nodes are distributed is of 1500 x 1500. Each node, when transmitting at maximum power is reaching a distance of 500.

We have done the same and in the next table the results are shown. In the case of LSP, as we do not have an exact number because it was another paper, we have approximated the value taking for reference a similar test in that paper.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Neighbours per node</th>
<th>Transmission Power</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig.</td>
<td>Sim.</td>
</tr>
<tr>
<td>No TC</td>
<td>25.6</td>
<td>25.5</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>15.4</td>
<td>11.6</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>12.3</td>
<td>9.0</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>12.8</td>
<td>10.1</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>10.3</td>
<td>8.0</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>7.0</td>
<td>9.3</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>3.6</td>
<td>5.6</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>3.6</td>
<td>5.6</td>
</tr>
<tr>
<td>LSP</td>
<td>4.0*</td>
<td>4.5</td>
</tr>
</tbody>
</table>

In the case of LSP the numbers are similar. And because the reference numbers we have are approximate we cannot extract solid conclusions. However, in the case of CBTC, we can see a consistent and really big improvement when comparing the transmission power values.

A possible explanation to that phenomena is that the networks randomly generated by Shawn are somehow more prone to this results. Another factor to take into account is that in the experiments of the original paper traffic lost and other effects were artificially generated in order to reproduce more realistically a real environment.

8.9 Time and memory usage

Regarding the time constraints in terms of code, there is not much to remark as the real bottleneck for both algorithms is the network – the transmission of messages. In the case of CBTC, the most expensive operations are quadratic and they involve the internal list of neighbour nodes. The same is true for LSP.

However, in memory terms, the typical hardware these algorithms are executed with has such little memory that it becomes an issue. In this regard, LSP is much more expensive as it has to maintain a relation of all its neighbours one to one, that is $O(x^2)$. On the other hand, the biggest CBTC memory structure is a list of its neighbours ($O(x)$).
Chapter 9

Project costs

The costs of this project can be categorized in two types: the ones specific to the project and the infrastructure already available and used for other purposes.

The first category would be entirely composed by the time spent doing the code and writing this document. In order to write the code, the papers describing the two algorithms had to be read and understood and a familiarization with the Wiselib was previously needed. To write this document, other than the time spent doing the tests with the simulator and with the real hardware in the testbeds, most of the time was essentially spent writing the document itself. Although it is hard to say how much time was spent in this process (because I was not able to work on it full time), approximately 6 months with full dedication were employed. As of 2013, the funds received by an FI pre-doctoral scholarship of the Generalitat de Catalunya are 14,400 euros for the first year so, roughly speaking, that would translate to approximately 7,000 euros.

In the second category other than the code and programs already written (the Wiselib itself or the Shawn simulator), there would be the cost of write the code, setup and maintaining the testbeds where the tests were performed. But such cost is really difficult to estimate and anyway it is shared infrastructure.

A scholarship (Becas de colaboración de estudiantes en departamentos universitarios; BOE-A-2009-13443, Orden EDU/2235/2009 ) of 2,700 of euros of the Ministerio de Educación y Cultura was given in order to support the realization of this project.
Chapter 10

Conclusions

In this chapter, a brief summary of the advantages and disadvantages of the two algorithms (and in the case of CBTC, the two recommended values for $\alpha$) is explained. A more detailed explanation of some aspects of the results chapter is also presented.

10.1 CBTC: $\alpha = 5/6$ vs. $\alpha = 2/3$

When comparing the two values for the $\alpha$ parameter, it is straightforward that $5/6$ always performs better than $2/3$.

When all optimizations are applied for both values, $5/6$ ties or performs better than $2/3$ in every metric: there is less distance between nodes, its simulation traffic is lower and it sends less messages. And the same is true when comparing the two values with the basic implementation (with no optimizations) of CBTC, the only exception being in the distances of the nodes. But then again, in the traffic simulation it is shown that it does not matter.

Of course $2/3$ has the advantage that it can perform an optimization (the asymmetric edge removal) that $5/6$ is not able to, but that apparent advantage becomes irrelevant once the final optimization, and the most effective one, is applied.

Last but not least, this optimization does not come for free. First of all, it introduces a significant complexity to a code that otherwise is pretty simple. Secondly, it requires messages to be sent and thus delaying the execution of the code. In the other two optimizations, only CPU is required to perform them so the bottleneck that represents the network is not hit. And finally, an additional list of neighbours is required in hardware were typically the memory is so scarce.

10.2 CBTC vs. LSP

The first thing to be pointed out about comparing LSP to CBTC is that it is not a fair comparison. CBTC relies on special technology not commonly available, the Angle of Arrival. That being said, CBTC with all optimizations applied outperforms LSP in the two most important metrics: transmission power and traffic simulation. Not only that, but LSP performs worse than all the CBTC configurations in the traffic simulation test in three of the four density
CHAPTER 10. CONCLUSIONS

scenarios, with the only exception being CBTC with no optimizations in the lowest density case. Surprisingly, because it behaves similarly in number of neighbours \[8.1\] transmission power and distance between nodes \[8.2\] to CBTC with all optimizations. The only possible explanation is that it creates a less efficient network in terms of routing. One possible cause for such inefficiency is the aggressive policy of neighbour dismiss of LSP: it may make sense from a local (the node) point of view, but this may be the cause of global problems. It would be interesting to investigate in that direction.

LSP also sends more messages than CBTC. That is because, although they both use the same mechanism to find their neighbours (broadcasting at every possible transmission level), the majority of nodes in a CBTC network stop this process while LSP checks every possible level. In this regard, it is worth mentioning that this is not specified by the original article\[11\] but otherwise no method is specified. In the article, it is hinted that if a method existed to find out the distances of the neighbours of a node (in terms of transmission power), then just one message would be needed per node. In that case, as can be seen in \[8.5\] and \[8.6\] LSP would be in CBTC levels or better.

In any case, although LSP behaves worse than CBTC, it does not do it so badly. Actually, in terms of number of neighbours, distance between nodes and transmission power, it has numbers similar to those of CBTC with all optimizations applied (for both \(\alpha\) values considered). And in the traffic simulation test, although worse than all CBTC configurations, it is still orders of magnitude better than with no topology control and very close to the CBTC values. Moreover, it does not require any special hardware and thus can be used in virtually any kind of hardware with transmission capabilities. While CBTC, if it does not have access to the Angle of Arrival technology, it has to rely on GPS (that at the same time is also specialized hardware but not as uncommon) that has problems of its own, as outlined in \[9\].

Finally, the last significant difference between both is memory consumption. As explained in \[8.9\] CBTC requires \(O(x)\) in relation to a node’s neighbours, while LSP requires \(O(x^2)\).

10.3 Scientific publications

With the CBTC implementation and other topology control algorithms implemented with the Wiselib, an article was published: “Topology control algorithms in WISELIB” \[1\] in 2010:


There is another article planned with the results of this master thesis for the end of this year.
Appendix A

Code

```cpp
#include <math.h> // NOTE: required for atan2f

#define PI 3.1415926535

namespace wiselib
{
/** 
 * brief Cbtc topology implementation of \ref topology_concept
 * "Topology Concept"
 */

// CbtcTopology<OsModel, Position, Radio, MAX_NODES>

typedef self_type;
typedef TopologyMessage;
typedef Neighbours;
```
typedef typename OsModel_P::size_t size_type;

typedef typename Radio::node_id_t node_id_t;

typedef typename Radio::size_t size_t;

typedef typename Radio::block_data_t block_data_t;

typedef typename Radio::TxPower Power;

typedef typename Timer::millis_t millis_t;

typedef vector_static<OsModel, node_id_t, MAX_NODES> Neighbors;

///@name Construction / Destruction
///@
CbtcTopology();
~CbtcTopology();
///@

///@name Main Control
///@
void enable( void );
void disable( void );
///@

///@name Methods called by Timer
///@
void timer_elapsed_first_phase( void *userdata );
void timer_elapsed_second_phase( void *userdata );
///@

///@name Methods called by RadioModel
///@
void receive( node_id_t from, size_t len, block_data_t *data );
///@

inline void set_startup_time( millis_t startup_time )
{ startup_time_ = startup_time; };

inline void set_work_period_1( millis_t work_period )
{ work_period_1 = work_period; };

inline void set_work_period_2( millis_t work_period )
{ work_period_2 = work_period; };

inline void set_alpha(double angle)
{ alpha = angle; };

Neighbors &topology(){
    N.clear();
    size_type i;
    for ( i = 0; i < neighbours.size(); ++i ){
        N.push_back(neighbours[i].id);
    }
    return N;
}

#ifdef DEBUG_TOPOLOGY_CBTC
void init( Position &pos, Radio& radio, Timer& timer,
           Debug& debug )
{    position_ = &pos;
    radio_ = &radio;
#endif


```cpp
timer_ = &timer;
dbg_ = &debug;
}
#else

void init( Position &pos, Radio& radio, Timer& timer) {
  position_ = &pos;
  radio_ = &radio;
  timer_ = &timer;
}
#endif

#endif

dvoid destruct() {
}

private:

Radio& radio() {
  return *radio_; }

Position& position() {
  return *position_; }

Timer& timer() {
  return *timer_; }

#endif DEBUG_TOPOLOGY_CBTC

Debug& debug() {
  return *debug_; }
#endif

Position * position_;  
Radio * radio_;  
Timer * timer_;  
#endif DEBUG_TOPOLOGY_CBTC

Debug * debug_;  
#endif

/** \brief Message IDs */
enum CbtcTopologyMsgIds {
  CbtcMsgIdHello = 200, ///< HELLO broadcasting
  CbtcMsgIdAck = 201, ///< HELLO acking
  CbtcMsgIdAsymmetric = 202, ///< Asymmetric removal
  CbtcMsgIdNDP = 203, ///< Neighbour Discovery Protol beacon
};

millis_t startup_time_;  
millis_t work_period_1;  
millis_t work_period_2;  
double alpha;

TopologyMessage helloMessage;  
TopologyMessage ackMessage;  
TopologyMessage asymmetricMessage;  
TopologyMessage NDPMessage;  

Neighbours neighbours;  

Power selfpower;  

Neighbors N;  // Topology
```

APPENDIX A. CODE

80
true just_started;
bool first_phase_done;
bool boundary_node;
bool enabled;

void generate_topology();
inline bool check_gap();
inline void shrinkback();
inline void pairwise();
inline void pairwise_one_node(size_type index, int power);

};

// ---------------------------------------------------------------
// ---------------------------------------------------------------
// ---------------------------------------------------------------

}.

// ---------------------------------------------------------------
// ---------------------------------------------------------------
// ---------------------------------------------------------------

template<typename OsModel_P,
typename Localization_P,
typename Radio_P,
uint16_t MAX_NODES>
CbtcTopology()
: startup_time_ ( 2000 ),
work_period_1 ( 5000 ),
work_period_2 (15000),
alpha(2.617993), // (5/6) * pi radians
// alpha(2.094395), // (2/3) * pi radians
helloMessage( CbtcMsgIdHello ),
ackMessage( CbtcMsgIdAck ),
asymmetricMessage( CbtcMsgIdAsymmetric ),
NDPMessage(CbtcMsgIdNDP),
enabled(false)
{
}

// ---------------------------------------------------------------
// ---------------------------------------------------------------
// ---------------------------------------------------------------

template<typename OsModel_P,
typename Localization_P,
typename Radio_P,
uint16_t MAX_NODES>
void
enable( void )
{
    enabled=true;
    just_started = true;
    selfpower = Power::MIN;
    first_phase_done = false;
    boundary_node = false;
neighbours.set_id( radio().id() );
radio().enable_radio();
#endif
radio().template reg_recv_callback<self_type, &self_type::receive>( this );
timer().template set_timer<self_type, &self_type::timer_elapsed_first_phase>(startup_time_, this, 0 );
}

// --------------------------------------------------------------
template<typename OsModel_P,
typeName Localization_P,
typeName Radio_P,
uint16_t MAX_NODES>
disable( void ) {
    enabled=false;
    #ifdef DEBUG_TOPOLOGY_CBTC
debug().debug("%i: Called CbtcTopology::disable\n",radio().id());
    #endif
}

// ---------------------------------------------------------------
template<typename OsModel_P,
typeName Localization_P,
typeName Radio_P,
uint16_t MAX_NODES>
timer_elapsed_first_phase( void* userdata ) {
    if(!enabled)
        return;
    if(first_phase_done)
        return;
    if(selfpower == Power::MAX) {
        generate_topology();
        return;
    }
    if(just_started)
        just_started = false;
    else
        selfpower++;
    #ifdef DEBUG_TOPOLOGY_CBTC
debug().debug("%i:CBTC,1stPhase,timer:,%i::Power::%i\n",radio().id(), selfpower.to_ratio() );
    #endif
    helloMessage.set_power(selfpower.to_ratio());
    helloMessage.set_position(position()().x(), position()().y());
radio().set_power(selfpower);
radio().send(
radio().BROADCAST_ADDRESS,
TopologyMessage::HELLO_SIZE,
(uint8_t*)&helloMessage);
timer().template set_timer<self_type,
&self_type::timer_elapsed_first_phase>(work_period_1, this, 0);
}

// ---------------------------------------------------------------

template<typename OsModel_P,
typename Localization_P,
typename Radio_P,
uint16_t MAX_NODES>
void
timer_elapsed_second_phase( void* userdata )
{
  if(!enabled)
    return;

  #ifdef DEBUG_TOPOLOGY_CBTC
    debug().debug( "%i: CBTC 2nd Phase timer: Power: %i
", radio().id() );
  #endif

  if(!first_phase_done)
    return;

  if(neighbours.ndp_update() && check_gap()) {
    first_phase_done = false;
    timer().template set_timer<self_type,
      &self_type::timer_elapsed_first_phase>
      (work_period_1, this, 0);
    #ifdef DEBUG_TOPOLOGY_CBTC
      debug().debug("%04x: Back to first phase\n", radio().id() );
    #endif
    return;
  }

  NDPMessage.set_power(selfpower.to_ratio());
  NDPMessage.set_position(position()().x(), position()().y());
  radio().send(  
    radio().BROADCAST_ADDRESS,
    TopologyMessage::NDP_SIZE,
    (uint8_t*)&NDPMessage  
  );

  timer().template set_timer<self_type,
    &self_type::timer_elapsed_second_phase>
    (work_period_2, this, 0 );
}

// ---------------------------------------------------------------

template<typename OsModel_P,
typename Localization_P,
typename Radio_P,
uint16_t MAX_NODES>
```c
    receive( node_id_t from, size_t len, block_data_t *data )
{
    if (!enabled)
        return;
    TopologyMessage *msg = (TopologyMessage *)data;
    uint8_t msg_id = msg->msg_id();
    int p = msg->power();
    Power power;
    power.from_ratio(p);
    double angle;
    bool check_for_gap;
    if ( from == radio().id() )
        return;
    angle = atan2f( msg->position_y() - position()().y(),
                   msg->position_x() - position()().x() );
    if ( msg_id == CbtcMsgIdHello ) {
        #ifdef DEBUG_TOPOLOGY_CBTC
        #ifdef DEBUG_TOPOLOGY_CBTC_V
        debug().debug( "%i:␣Received␣HELLO␣from␣%i␣with␣power:␣%i
", radio().id(), from, p );
        #endif
        #endif
        neighbours.add_update_neighbour(from, p, angle, true);
        ackMessage.set_power(p);
        ackMessage.set_position(position()().x(), position()().y());
        radio().set_power( power );
        radio().send( from,
                      TopologyMessage::ACK_SIZE,
                      (uint8_t *)&ackMessage
                      );
        radio().set_power( selfpower );
    } else if ( msg_id == CbtcMsgIdAck ) {
        #ifdef DEBUG_TOPOLOGY_CBTC
        #ifdef DEBUG_TOPOLOGY_CBTC_V
        debug().debug( "%i:␣Received␣ACK␣from␣%i␣x:␣%f␣y:␣%f
", radio().id(), from, msg->position_x(),
                       msg->position_y() );
        #endif
        #endif
        neighbours.add_update_neighbour(from, p, angle, false);
        if (!first_phase_done)
            generate_topology();
    } else if( msg_id == CbtcMsgIdAsymmetric ) {
        #ifdef DEBUG_TOPOLOGY_CBTC
        #ifdef DEBUG_TOPOLOGY_CBTC_V
        debug().debug( "%i:␣Received␣ASYMMETRIC␣from␣%i
", radio().id(), from );
        #endif
        #endif
    }
    neighbours.add_update_neighbour(from, p, angle, false);
    if (!first_phase_done)
        generate_topology();
} else if( msg_id == CbtcMsgIdAsymmetric ) {
    #ifdef DEBUG_TOPOLOGY_CBTC
    #ifdef DEBUG_TOPOLOGY_CBTC_V
    debug().debug( "%i:␣Received␣ASYMMETRIC␣from␣%i
", radio().id(), from );
    #endif
    #endif
    #endif
    #endif
```

if(!first_phase_done)
    neighbours.add_asymmetric_to_remove(from);
} else if( msg_id == CbtcMsgIdNDP) {
    #ifdef DEBUG_TOPOLOGY_CBTC
    #ifdef DEBUG_TOPOLOGY_CBTC_V
    debug().debug( "%i:␣Received␣NDP␣from␣%i\n", radio().id(), from );
    #endif
    #endif
    if(!first_phase_done)
        return;

    check_for_gap = neighbours.add_update_ndp(from, p, angle);
    if(check_for_gap && check_gap()) {
        first_phase_done = false;
        timer().template set_timer<self_type,
            &self_type::timer_elapsed_first_phase>( work_period_1, this, 0 );
        return;
    }

    shrinkback();
}

// --------------------------------------------------------------

if(first_phase_done)
    return;

check_for_gap = neighbours.add_update_ndp(from, p, angle);
if(check_for_gap && check_gap()) {
    first_phase_done = false;
    timer().template set_timer<self_type,
        &self_type::timer_elapsed_first_phase>( work_period_1, this, 0 );
    return;
}

shrinkback();

// --------------------------------------------------------------

{
    if(first_phase_done)
        return;

    gap_exists = check_gap();

    if(selfpower < Power::MAX && gap_exists)
        return;

    first_phase_done = true;

    if(gap_exists)
        boundary_node = true;

    neighbours.first_phase_done = true;
    neighbours.copy_to_NDP();

    #ifdef DEBUG_TOPOLOGY_CBTC
    debug().debug( "%i:␣First␣phase␣done␣and␣boundary␣node\n", radio().id(), boundary_node ?"B":"Not\b" );
    neighbours.print_basic();
    #endif

    //#Optimizations
477 //Shrink-back operation
478 if(boundary_node)
479 shrinkback();
480
481 #ifdef DEBUG_TOPOLOGY_CBTC
482 neighbours.print_optimization("Nodes[Shrinkback]:");
483 #endif
484
485 //Asymmetric edge removal
486 if(alpha <= 2 * PI / 3) {
487 //Nodes told to be deleted
488 for( i = 0; i < neighbours.ATR.size(); ++i){
489 neighbours.delete_by_id(neighbours.ATR[i]);
490 }
491 neighbours.ATR.clear();
492
493 //Tell nodes to delete self
494 for( i = 0; i < neighbours.size(); ++i ){
495 if(neighbours[i].asymmetric){
496 radio().send(
497 neighbours[i].id,
498 TopologyMessage::ASYMMETRIC_SIZE,
499 (uint8_t*)&asymmetricMessage
500 );
501 neighbours.delete_by_index(i);
502 i--;}
503 }
504 } else {
505 for( i = 0; i < neighbours.size(); ++i ){
506 neighbours[i].asymmetric = false;
507 }
508 }
509 }
510
511 #ifdef DEBUG_TOPOLOGY_CBTC
512 neighbours.print_optimization("Nodes[Asymmetric]:");
513 #endif
514
515 //Calculate the power to beacon
516 if(!boundary_node) {
517 power = (Power::MIN).to_ratio();
518 for( i = 0; i < neighbours.size(); ++i ){
519 if(neighbours[i].power > power)
520 power = neighbours[i].power;
521 }
522 selfpower = Power::from_ratio(power);
523 }
524
525 //Pair-wise edge removal
526 pairwise();
527
528 #ifdef DEBUG_TOPOLOGY_CBTC
529 neighbours.print_optimization("Nodes[Pairwise]:");
530 #endif
531
532 TopologyBase<OsModel>::notify_listeners();
533
534 timer().template set_timer<self_type,
535 &self_type::timer_elapsed_second_phase>
536 ( work_period_2, this, 0 );
537 }
// ---------------------------------------------------------------
template<typename OsModel_P,
         typename Localization_P,
         typename Radio_P,
         uint16_t MAXNODES>
inline bool
check_gap(){
    size_type first, i, j;
    if(neighbours.size() < 2)
        return true;
    for (first = 0; first < neighbours.size(); ++first)
        if (!neighbours[first].asymmetric)
            break;
    if(first >= neighbours.size() - 1)
        return true;
    j = first;
    for (i = first + 1; i < neighbours.size(); ++i){
        if (!neighbours[i].asymmetric)
            if(neighbours[i].angle - neighbours[j].angle > alpha)
                return true;
            j = i;
    }
    return PI * 2.0 +
            neighbours[first].angle - neighbours[j].angle > alpha;
}

// ---------------------------------------------------------------
template<typename OsModel_P,
         typename Localization_P,
         typename Radio_P,
         uint16_t MAXNODES>
inline void
shrinkback(){
    if(neighbours.size() < 3)
        return;
    int p_threshold;
    size_type i, j, k;
    for (p_threshold = (Power::MAX).to_ratio();
         p_threshold > (Power::MIN).to_ratio();
         p_threshold--){
        for (i = 0; i < neighbours.size(); ++i){
            if(neighbours[i].power < p_threshold)
                break;
        }
        //There is no one or only one
        if (i >= neighbours.size() - 1)
            return;
for (; j < neighbours.size(); ++j)
    if (neighbours[j].power < p_threshold) {
        if (j - k > 1 && neighbours[j].angle -
            neighbours[k].angle > alpha)
            return;
        k = j;
    }

    // There is only one
    if (k == i)
        return;

    if (!(i == 0 && k == neighbours.size() - 1) &&
        PI * 2 + neighbours[i].angle - neighbours[k].angle > alpha){
        return;
    }

    // delete same level
    neighbours.delete_by_power(p_threshold);
}

// ---------------------------------------------------------------
template<typename OsModel_P,
          typename Localization_P,
          typename Radio_P,
          uint16_t MAX_NODES>
inline void
pairwise()
{
    int i, j, fst_below, last_current, last_below;
    int p_threshold;
    double angle;
    for (p_threshold = (Power::MAX).to_ratio();
         p_threshold > (Power::MIN).to_ratio();
         p_threshold--){
        for (fst_below = 0;
             fst_below < (int)neighbours.size();
             ++fst_below )
            if (neighbours[fst_below].power < p_threshold)
                break;

        if(fst_below == (int)neighbours.size())
            return;

        i = fst_below + 1;
        last_current = -1;
        last_below = fst_below;
        for (; i < (int)neighbours.size(); ++i ){
            // Node on the power level to be deleted
            if(neighbours[i].power == p_threshold) {
                if(last_current == -1) {
                    if(neighbours[i].angle -
neighbours[last_below].angle > PI / 3)
last_current = i;
}
// Otherwise node with a power below power to be deleted
else {
    if (last_current != -1) {
        for(j = last_current; j < i; j++) {
            if (neighbours[i].angle -
                neighbours[j].angle > PI / 3)
                break;
        }
        if (j < i)
            return;
        else
            last_current = -1;
    }
    last_below = i;
}

// There is just one nodes with power below power to be
// deleted if (last_below == fst_below){
pairwise_one_node(fst_below, p_threshold);
return;
}

// If all nodes at the end of the vector with power to be
// deleted can be deleted
if (last_current == -1) {
    angle = neighbours[last_below].angle - 2 * PI;
    for(i = 0; i < fst_below; i++){
        if (neighbours[i].angle - angle > PI / 3)
            break;
    }
    if (angle != neighbours[last_current].angle)
        return;
    i = 0;
}

// Otherwise, if there are some nodes with power to be
// deleted after the last node with power below
else {
    angle = neighbours[fst_below].angle + 2 * PI;
    for(i = last_current; i < (int)neighbours.size(); i++){
        if (angle - neighbours[i].angle > PI / 3)
            return;
    }
    for(; i < fst_below; i++){
        if (neighbours[fst_below].angle - neighbours[i].angle
            > PI / 3)
            return;
    }
    neighbours.delete_by_power(p_threshold);
}

// ---------------------------------------------------------------
template<typename OsModel_P,
typeiname Localization_P,
typeiname Radio_P,
pairwise_one_node(size_type index, int power)
{
    size_type i;

double angle1 = neighbours[index].angle;
double angle2 = neighbours[index].angle - PI * 2;
for (i = 0; i < index; ++i){
    if(angle1 - neighbours[i].angle < PI / 3)
        continue;
    else if (neighbours[i].angle - angle2 < PI / 3)
        continue;
    else
        return;
}

angle1 = neighbours[index].angle;
angle2 = neighbours[index].angle + PI * 2;
for (i = index + 1; i < neighbours.size(); ++i){
    if(neighbours[i].angle - angle1 < PI / 3)
        continue;
    else if (angle2 - neighbours[i].angle < PI / 3)
        continue;
    else
        return;
}

neighbours.delete_by_power(power);
}

#endif
inline CbtcTopologyMessage( uint8_t id );

inline uint8_t msg_id()
{
    return read<OsModel, block_data_t, uint8_t>( buffer );
}

inline void set_msg_id( uint8_t id )
{
    write<OsModel, block_data_t, uint8_t>( buffer, id );
}

inline int power()
{
    return (int)read<OsModel, block_data_t, int>( buffer + POWER_POS);
}

inline void set_power( int power ){
    write<OsModel, block_data_t, int>(buffer + POWER_POS, power);
}

inline double position_x() {
    return read<OsModel, block_data_t, double>(buffer + POSITIONX_POS);
}

inline double position_y() {
    return read<OsModel, block_data_t, double>(buffer + POSITIONY_POS);
}

inline void set_position( double x, double y )
{
    write<OsModel, block_data_t, double>(buffer + POSITIONX_POS, x);
    write<OsModel, block_data_t, double>(buffer + POSITIONY_POS, y);
}

private:
enum data_positions
{
    MSG_ID_POS = 0,
    POWER_POS = MSG_ID_POS + sizeof(uint8_t),
    POSITIONX_POS = (POWER_POS + sizeof(int)),
    POSITIONY_POS = (POSITIONX_POS + sizeof(double))
};

uint8_t buffer[sizeof(uint8_t) + sizeof(int) +
sizeof(double) * 2];
APPENDIX A. CODE

template<typename OsModel_P, typename Radio_P>
CbtcTopologyMessage<OsModel_P, Radio_P>::
CbtcTopologyMessage( uint8_t id )
{
    set_msg_id( id );
}
}
#endif

namespace wiselib
{
    template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
    class CbtcTopologyNeighbours
    {
        public:
            typedef OsModel_P OsModel;
            typedef Radio_P Radio;
            #ifdef DEBUG_TOPOLOGY_CBTC
            typedef typename OsModel::Debug Debug;
            #endif
        
        typedef typename OsModel_P::size_t size_type;
        
        typedef typename Radio::node_id_t node_id_t;
        
        typedef struct triplet_t{
            double angle;
            node_id_t id;
            int power;
            bool asymmetric;
        } TIPA_t;
        
        typedef struct ndp_struct{
            node_id_t id;
            int power;
            bool first_time_seen;
            uint8_t ndp_counter;
        } ndp_t;
        
        typedef vector_static<OsModel, TIPA_t, MAX_NODES> Nodes;
        Nodes N;
        
        typedef normal_iterator<OsModel, TIPA_t*, Nodes> Niter;
        
        //Asymmetric Nodes to be removed
        vector_static<OsModel, node_id_t, MAX_NODES> ATR;
        vector_static<OsModel, ndp_t, MAX_NODES> NDP;
        
        bool first_phase_done;
        
        CbtcTopologyNeighbours();
        
        inline void set_id(node_id_t id) {id_ = id; }
        
        inline size_type size(){ return N.size(); }
        
        inline TIPA_t& operator[](size_type n) { return N[n];}
    }
void add_update_neighbour(
    node_id_t id, int p, double angle, bool asymmetric);
inline void delete_by_id(node_id_t id);
inline void delete_by_index(size_type index);
bool add_update_ndp(node_id_t id, int p, double angle);
bool npd_update();
inline void delete_by_power(int p){
    size_type i;
    for ( i = 0; i < N.size(); ++i ){
        if(N[i].power == p){
            delete_by_index(i);
            i--;
        }
    }
}
inline void add_asymmetric_to_remove(node_id_t from){
    size_type i;
    for(i = 0; i < ATR.size(); i++)
        if(ATR[i] == from)
            break;
    if(i == ATR.size())
        ATR.push_back(from);
}
inline void copy_to_NDP();
#endif DEBUG_TOPOLOGY_CBTC
inline void print_basic();
inline void print_optimization(const char *s);
#endif DEBUG_TOPOLOGY_CBTC
private:
    node_id_t id_;
if(first_phase_done){
  //Add it to NDP or update it if it's already there
  for ( i = 0; i < NDP.size(); ++i ){
    if(NDP[i].id == id){
      NDP[i].first_time_seen = true;
      if(NDP[i].power > p) NDP[i].power = p;
    }
  }
  if(i == NDP.size()){
    ndp_t ndp;
    ndp.id = N[i].id;
    ndp.power = N[i].power;
    ndp.ndp_counter = 0;
    ndp.first_time_seen = true;
    NDP.push_back(ndp);
  }
}

//Remove from ATR if it's there and if now is symmetric
if(!first_phase_done and !asymmetric){
  for ( i = 0; i < ATR.size(); ++i )
    if(ATR[i] == id)
      break;
  if(i < ATR.size()){
    for(; i < ATR.size() - 1; i++)
      ATR[i] = ATR[i+1];
    ATR.pop_back();
  }
}

//Update it in N
//If we are in second phase just update don't add it to N if
//it's not there
for ( i = 0; i < N.size(); ++i ){
  if (N[i].id == id) {
    if (N[i].power > p ) N[i].power = p;
    N[i].angle = angle;
    N[i].asymmetric = N[i].asymmetric && asymmetric;
    break;
  }
}

if(first_phase_done or i < N.size())
  return;

//Add it to N (just if we are in first phase)
TIPA_t t;
t.id = id;
t.power = p;
t.angle = angle;
t.asymmetric = asymmetric;
for ( i = 0; i < N.size(); ++i ){
  if (angle < N[i].angle) {
    N.push_back(t);
    for(j = N.size() - 1; j > i; j--){
      N[j] = N[j-1];
    }
  }
}
APPENDIX A. CODE

176  N[i] = t;
177  return;
178 }
179 }
180 }
181 N.push_back(t);
182 }
183 
184 // -------------------------------------------------------------
185 template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
186 inline void
187 CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
188 delete_by_id(node_id_t id){
189  size_type i;
190  
191 for( i = 0; i < N.size(); i++)
192     if(N[i].id == id)
193         break;
194  
195 for(; i < N.size() - 1; i++)
196     N[i] = N[i+1];
197 
198 N.pop_back();
199 }
200 
201 // -------------------------------------------------------------
202 template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
203 inline void
204 CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
205 delete_by_index(size_type index){
206  size_type i;
207  
208 for(i = index; i < N.size() - 1; i++)
209     N[i] = N[i+1];
210 
211 N.pop_back();
212 }
213 
214 // -------------------------------------------------------------
215 template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
216 inline void
217 CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
218 copy_to_NDP(){
219  size_type i;
220  ndp_t ndp;
221  for(i = 0; i < N.size(); i++){
222     ndp.id = N[i].id;
223     ndp.power = N[i].power;
224     ndp.ndp_counter = 0;
225     ndp.first_time_seen = true;
226     NDP.push_back(ndp);
227  }
228 }
229 
230 // -------------------------------------------------------------
231 template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
232 bool
233 CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
234 add_update_ndp(node_id_t id, int p, double angle){
235  size_type i, j;
236  ndp_t ndp;
bool has_to_check_gap = false;

// Update it in NDP
for (i = 0; i < NDP.size(); i++) {
    if (id == NDP[i].id) {
        NDP[i].ndp_counter += 1;
        break;
    }
}

// If it's not in NDP add it
if (i == NDP.size()) {
    ndp.id = id;
    ndp.power = p;
    ndp.ndp_counter = 0;
    ndp.first_time_seen = true;
    NDP.push_back(ndp);
}

// Update it in N
for (j = 0; j < N.size(); j++) {
    if (id == N[j].id) {
        if (N[j].angle != angle)
            has_to_check_gap = true;
        N[j].angle = angle;
        break;
    }
}

// If it's not in N add it sorted
if (j == N.size()) {
    TIPA_t t;
    t.id = id;
    t.power = NDP[i].power;
    t.angle = angle;
    t.asymmetric = false;

    N.iter it;
    for (it = N.begin(); it != N.end(); it++) {
        if (angle < (*it).angle) {
            N.insert(it, t);
            break;
        }
    }
}

return has_to_check_gap;

// -------------------------------------------------------------
template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
bool CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
ndp_update(){
    size_type i, j;
    bool check_for_gap = false;
    node_id_t id;
}
for (i = 0; i < NDP.size(); i++){
  if(NDP[i].first_time_seen){
    NDP[i].first_time_seen = false;
    NDP[i].ndp_counter = 0;
  } else {
    if(NDP[i].ndp_counter) {
      NDP[i].ndp_counter = 0;
    } else {
      id = NDP[i].id;
      for(j = i; j < NDP.size() - 1; j++) {
        NDP[j] = NDP[j+1];
      }
      NDP.pop_back();
      for(j = 0; j < N.size(); j++) {
        if(N[j].id == id) {
          check_for_gap = true;
          delete_by_index(j);
          break;
        }
      }
    }
  }
}

return check_for_gap;

#ifdef DEBUG_TOPOLOGY_CBTC
// -------------------------------------------------------------
template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
inline void
CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
print_basic(){
  size_type i;
  char nodes[MAX_NODES*7+1];
  int n = 0;
  for ( i = 0; i < N.size(); ++i )
    n += snprintf(nodes + n, 7, "%5i \t", N[i].id );
  debug().debug("%i: Nodes[Firstphase]:\t\%s\n", id_, nodes );
  n = 0;
  for ( i = 0; i < N.size(); ++i )
    n += sprintf(nodes + n, "%5.2f \t", N[i].angle );
  debug().debug("%i: Nodes[Angles]:\t\%s\n", id_, nodes );
  n = 0;
  for ( i = 0; i < N.size(); ++i )
    n += sprintf(nodes + n, "%5i \t", N[i].power );
  debug().debug("%i: Nodes[Powers]:\t\%s\n", id_, nodes );
  n = 0;
  for ( i = 0; i < N.size(); ++i )
    n += sprintf(nodes + n, "%s\t", N[i].asymmetric? "\t\t\tyes" : "\t\t\tno" );
  debug().debug("%i: Nodes[asymmetry]:\t\%s\n", id_, nodes );
}
// -------------------------------------------------------------
#endif
template<typename OsModel_P, typename Radio_P, uint16_t MAX_NODES>
inline void
CbtcTopologyNeighbours<OsModel_P, Radio_P, MAX_NODES>::
print_optimization(const char *s){
    size_type i;
    char nodes[MAX_NODES*7+1];
    int n = 0;
    for ( i = 0; i < N.size(); ++i )
        n += snprintf(nodes + n, 7, "%5i\t", N[i].id );
    debug().debug("%i:␣%s␣%s
", id_, s, nodes );
}
#endif
#endif

namespace wiselib
{
    /** \brief LSP topology implementation
        * of \ref topology_concept "Topology Concept"
        * \ingroup topology_concept
        */
    template<typename OsModel_P,
        uint16_t MAX_NEIGHBORS = 32,
        typename Radio_P=typename OsModel_P::Radio,
        typename Timer_P=typename OsModel_P::Timer>
    class LSPTopology : public TopologyBase<OsModel_P>
    {
        public:
            typedef OsModel_P OsModel;
            typedef Radio_P Radio;
            typedef Timer_P Timer;
        #ifdef DEBUG_TOPOLOGY_LSP
        typedef typename OsModel::Debug Debug Debug;
        #endif
    }
    typedef LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P,
        Timer_P> self_type;
    typedef LSPTopologyMessage<OsModel, Radio> LSPMessage;
    typedef typename Radio::node_id_t node_id_t;
    typedef typename Radio::size_t size_t;
    typedef typename Radio::block_data_t block_data_t;
    typedef typename Radio::TxPower Power;
    typedef typename Timer::millis_t millis_t;
typedef vector_static<OsModel, node_id_t, MAX_NEIGHBORS> Neighbors;

struct arc {
    int u; // vertex origin
    int v; // vertex destination
double p; // arc's weight
}

arc (int u, int v, double p) : u(u), v(v), p(p) { }
arc () : u(-1), v(-1), p(0) {}

bool operator< (const arc& b) const {
    return p < b.p;
}

};

///@name Construction / Destruction
///@
LSPTopology();
~LSPTopology();
///@}

///@name Main Control
///@
void enable( void );
void disable( void );
///@}

///@name Methods called by Timer
///@
void timer1( void *userdata );
void timer2( void *userdata );
///@}

///@name Methods called by RadioModel
///@
void receive( node_id_t from, size_t len,
block_data_t *data );
///@}

inline void set_startup_time( millis_t startup_time )
{ startup_time_ = startup_time; }

inline void set_work_period( millis_t work_period )
{ work_period_ = work_period; }

inline void set_wait_time( millis_t wait_time )
{ wait_time_ = wait_time; }

Neighbors &topology()
{ return N; }

#define DEBUG_TOPOLOGY_LSP

void init(Radio& radio, Timer& timer, Debug& debug ) {
    radio_ = &radio;
timer_ = &timer;
debug_ = &debug;
}

#else

void init(Radio& radio, Timer& timer) {
APPENDIX A. CODE

106     radio_ = &radio;
107     timer_ = &timer;
108 }
109 #endif
110
111 void destruct() {
112 }
113
114 private:
115
116 Radio& radio()
117 { return *radio_; }
118
119 Timer& timer()
120 { return *timer_; }
121
122 #ifdef DEBUG_TOPOLOGY_LSP
123 Debug& debug()
124 { return *debug_; }
125 #endif
126
127 Radio * radio_;
128 Timer * timer_;
129 #ifdef DEBUG_TOPOLOGY_LSP
130 Debug * debug_;
131 #endif
132
133 /** \brief Message IDs */
134 +/
135 enum LSPTopologyMsgIds {
136 LSPMsgIdHello = 200, ///< hello broadcasting
137 LSPMsgIdAck = 201, ///< hello acking
138 LSPMsgIdInfo = 202, ///< info about links
139);
140
141 // ---------------------------------------------------------
142 // This should be in the external interface file
143 enum PowerValues {
144 POWER_NULL_VALUE = -1, ///< Tx power null value
145};
146
147 //@name Data
148 //@{
149 Neighbors N; // Topology
150 //@}
151
152 millis_t startup_time_; 
153 millis_t work_period_; 
154 millis_t wait_time_; 
155
156 LSPMessage helloMessage; 
157 LSPMessage ackMessage; 
158 LSPMessage infoMessage; 
159
160 Power selfpower; 
161
162 int callback_id; 
163 bool enabled; 
164 bool just_started; 
165
166 int NMatrix[MAX_NEIGHBORS][MAX_NEIGHBORS]; 
167 Neighbors NMapping;
vector_static<OsModel, bool, MAX_NEIGHBORS> S;
vector_static<OsModel, int, MAX_NEIGHBORS> P;
vector_static<OsModel, int, MAX_NEIGHBORS> D;
priority_queue<OsModel, arc, MAX_NEIGHBORS * MAX_NEIGHBORS> PQ;

void add_neighbor(node_id_t from, node_id_t to, int power);
void remove_asymmetric_links();
void dijkstra();

void enable( void ) {
  just_started = true;
  selfpower = Power::MIN;
  memset(NMatrix, POWER_NULL_VALUE, sizeof(int) * MAX_NEIGHBORS * MAX_NEIGHBORS);
  NMapping.push_back(radio().id());
  radio().enable_radio();
  #ifdef DEBUG_TOPOLOGY_LSP
  debug().debug("%04x\x00:LSPTopology\x00:Enable\n", radio().id());
  #endif
}

/*LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::LSPTopology():
startup_time_ ( 2000 ),
work_period_ ( 5000 ),
wait_time_ ( 10000 ),
helloMessage ( LSPmsgIdHello ),
ackMessage ( LSPmsgIdAck ),
infoMessage ( LSPmsgIdInfo )
*/
void enable( void ) {
  just_started = true;
  selfpower = Power::MIN;
  memset(NMatrix, POWER_NULL_VALUE, sizeof(int) * MAX_NEIGHBORS * MAX_NEIGHBORS);
  NMapping.push_back(radio().id());
  radio().enable_radio();
  #ifdef DEBUG_TOPOLOGY_LSP
  debug().debug("%04x\x00:LSPTopology\x00:Enable\n", radio().id());
  #endif
}
# endif

callback_id = radio().template reg_recv_callback<self_type, &self_type::receive>( this );

timer().template set_timer<self_type, &self_type::timer1>( startup_time_, this, 0 );

enabled = true;

// -------------------------------------------------------------

template<typename OsModel_P, uint16_t MAX_NEIGHBORS, typename Radio_P, typename Timer_P>
void LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::disable( void ) {
    enabled = false;
    #ifdef DEBUG_TOPOLOGY_LSP
    debug().debug("%04x: Called LSPTopology::disable\n", radio().id() );
    #endif
    radio().unreg_recv_callback( callback_id );
}

// -------------------------------------------------------------

template<typename OsModel_P, uint16_t MAX_NEIGHBORS, typename Radio_P, typename Timer_P>
void LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::timer1( void* userdata ) {
    if(!enabled)
        return;
    #ifdef DEBUG_TOPOLOGY_LSP
    if(enabled)
        debug("%04x: Executing Timer1 'LSPTopology'\n", radio().id() );
    #endif
    if(selfpower >= Power::MAX) {
        for(unsigned int i = 1; i < MAX_NEIGHBORS; i++){
            if(NMatrix[0][i] == POWER_NULL_VALUE)
                continue;

            infoMessage.set_other_id(NMapping[i]);
            infoMessage.set_power(NMatrix[0][i]);
            radio().send( Radio::BROADCAST_ADDRESS, LSPMessage::INFO_SIZE, (uint8_t*)&infoMessage );
        }
    }
    #ifdef DEBUG_TOPOLOGY_LSP
    if(enabled)
        debug("%04x: Sent INFO: Power: %i\n", radio().id(), NMatrix[0][i]);
    #endif
    #ifdef DEBUG_TOPOLOGY_LSP_V
    debug("%04x: Sent INFO: Power: %i\n", radio().id(), NMatrix[0][i]);
    #endif
}
APPENDIX A. CODE

```cpp
291 #endif
292 }
293   timer().template set_timer<self_type, &self_type::timer2>( wait_time_, this, 0 );
294   return;
295 }
296 if(just_started)
297   just_started = false;
298 else
299   selfpower = Power::from_ratio( 
300     selfpower.to_ratio() + 100); 
301   helloMessage.set_power(selfpower.to_ratio()); 
302   radio().set_power(selfpower); 
303   radio().send( 
304     Radio::BROADCAST_ADDRESS, 
305     LSPMessage::HELLO_SIZE, 
306     (uint8_t*)&helloMessage 
307   );
308 #ifdef DEBUG_TOPOLOGY_LSP
309 #ifdef DEBUG_TOPOLOGY_LSP_V
310   debug().debug("%04x:␣Sent:\nHELLO:\nPower:%i\n", 
311     radio().id(), selfpower.to_ratio() );
312 #endif
313 #endif
314   timer().template set_timer<self_type, &self_type::timer1>( 
315     work_period_, this, 0 );
316 // -------------------------------------------------------------
317 template<typename OsModel_P,
318   uint16_t MAX_NEIGHBORS,
319   typename Radio_P,
320   typename Timer_P>
321 void
322 LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::
323 timer2( void* userdata )
324 {
325   if(!enabled)
326     return;
327 #ifdef DEBUG_TOPOLOGY_LSP
328   debug().debug("%04x:␣Executing␣Timer2␣'LSPTopology'
", 
329     radio().id() );
330 #endif
331   remove_asymmetric_links();
332   dijsktra();
333   int p = (Power::MIN).to_ratio();
334   for(unsigned int i = 0; i < P.size(); i++){
335     if(P[i] == 0) {
336       N.push_back(NMapping[i]);
337       if(NMatrix[0][i] > p)
338         p = NMatrix[0][i];
339     }
340   }
341   radio().set_power(Power::from_ratio(p));
342 #ifdef DEBUG_TOPOLOGY_LSP
343   int n = 0;
344   char nodes[MAX_NEIGHBORS*7+1];
345   #endif
```
nodes[0] = '\0';
for (unsigned int i = 0; i < N.size(); ++i )
    n += sprintf(nodes + n, "%04x", N[i] );
debug().debug("%04x:\n Neighbours:\n", radio().id(), nodes );
debug().debug("%04x:\n Power:\n", radio().id(), p);
#endif
TopologyBase<OsModel>::notify_listeners();

// -------------------------------------------------------------
template<typename OsModel_P,
    uint16_t MAX_NEIGHBORS,
    typename Radio_P,
    typename Timer_P>
void
LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::
receive( node_id_t from, size_t len, block_data_t *data )
{
    if ( !enabled || from == radio().id() )
        return;

    uint8_t msg_id = *data;
    LSPMessage *msg = (LSPMessage *)data;
    int p = msg->power();

    if ( msg_id == LSPMsgIdHello ){
        #ifdef DEBUG_TOPOLOGY_LSP
            #ifdef DEBUG_TOPOLOGY_LSP_V
                debug().debug("%04x: Received HELLO msg from %04x. Power %i\n", radio().id(), from, p );
            #endif
        #endif
        ackMessage.set_power(p);
        radio().set_power(Power::MAX);
        radio().send(
            from,
            LSPMessage::ACK_SIZE,
            (uint8_t*)&ackMessage
        );
        #ifdef DEBUG_TOPOLOGY_LSP
            #ifdef DEBUG_TOPOLOGY_LSP_V
                debug().debug("%04x: Sent ACK Power %i\n", radio().id(), p );
            #endif
        #endif
        #endif
        #endif
        radio().set_power(selfpower);
    } else if ( msg_id == LSPMsgIdAck ){
        #ifdef DEBUG_TOPOLOGY_LSP
            #ifdef DEBUG_TOPOLOGY_LSP_V
                debug().debug("%04x: Received ACK msg from %04x. Power %i\n", radio().id(), from, p );
            #endif
        #endif
        #endif
        #endif
        add_neighbor(radio().id(), from, p);
    } else if ( msg_id == LSPMsgIdInfo ){
        #ifdef DEBUG_TOPOLOGY_LSP
            #ifdef DEBUG_TOPOLOGY_LSP_V
                debug().debug("%04x: Received INFO msg from %04x. Power %i\n", radio().id(), from, p );
            #endif
        #endif
        #endif
        #endif
        #endif
        // -------------------------------------------------------------
debug().debug(
  "%04x:INFO:msg from %04x:Power %i\n",
  radio().id(), from, p);
#endif
#endif
add_neighbor(from, msg->other_id(), p);
}
}

// -------------------------------------------------------------

template<typename OsModel_P, 
  uint16_t MAX_NEIGHBORS, 
  typename Radio_P, 
  typename Timer_P>
void 
LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::
add_neighbor(node_id_t from, node_id_t to, int power)
{
  unsigned int i, j;
  for(i = 0; i < NMapping.size(); i++)
    if(NMapping[i] == from)
      break;
  if(i == NMapping.size()){
    if(NMapping.size() >= NMapping.capacity()) {
      #ifdef DEBUG_TOPOLOGY_LSP
        debug().debug(
          "%04x:ERROR:out of memory\n", radio().id());
      #endif
      return;
    }
    NMapping.push_back(from);
  }
  for(j = 0; j < NMapping.size(); j++)
    if(NMapping[j] == to)
      break;
  if(j == NMapping.size()){
    if(NMapping.size() >= NMapping.capacity()) {
      #ifdef DEBUG_TOPOLOGY_LSP
        debug().debug(
          "%04x:ERROR:out of memory\n", radio().id());
      #endif
      return;
    }
    NMapping.push_back(to);
  }
  if(NMatrix[i][j] > power or NMatrix[i][j] == POWER_NULL_VALUE)
    NMatrix[i][j] = power;
}

// -------------------------------------------------------------

template<typename OsModel_P, 
  uint16_t MAX_NEIGHBORS, 
  typename Radio_P, 
  typename Timer_P>
inline void 
LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::
remove_asymmetric_links()
APPENDIX A. CODE

{  
    #ifdef DEBUG_TOPOLOGY_LSP
    #ifdef DEBUG_TOPOLOGY_LSP_V
    #ifdef DEBUG_TOPOLOGY_LSP_VV
    int p;
    printf("%04x: Collected Information\n", radio().id());
    printf("%04x:\n", radio().id());
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x\", NMapping[i]);
    printf("\n");
    
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x:\n", radio().id(), NMapping[i]);
    for(unsigned int j=0; j<NMapping.size(); j++)
        p = NMatrix[i][j];
    
    p = (p == POWER_NULL_VALUE ? POWER_NULL_VALUE : (p * p) / 10000);
    printf("\n");
    
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x: Remove asymmetric links\n", radio().id());
    printf("%04x:\n", radio().id());
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x\", NMapping[i]);
    printf("\n");
    
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x:\n", radio().id(), NMapping[i]);
    for(unsigned int j=0; j<NMapping.size(); j++)
        p = NMatrix[i][j];
    
    p = (p == POWER_NULL_VALUE ? POWER_NULL_VALUE : (p * p) / 10000);
    printf("\n");
    
    #endif
    #endif
    #endif

    for(unsigned int i = 0; i < NMapping.size(); i++) {
        for(unsigned int j = i + 1; j<NMapping.size(); j++)
            if(NMatrix[i][j] == POWER_NULL_VALUE)
                NMatrix[j][i] = POWER_NULL_VALUE;
            else if(NMatrix[j][i] == POWER_NULL_VALUE)
                NMatrix[i][j] = POWER_NULL_VALUE;
            else {
                NMatrix[i][j] = (NMatrix[i][j] + NMatrix[j][i]) / 2;
            }
    }

    #ifdef DEBUG_TOPOLOGY_LSP
    #ifdef DEBUG_TOPOLOGY_LSP_V
    #ifdef DEBUG_TOPOLOGY_LSP_VV
    printf("%04x: Remove asymmetric links\n", radio().id());
    printf("%04x:\n", radio().id());
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x\", NMapping[i]);
    printf("\n");
    
    for(unsigned int i=0; i<NMapping.size(); i++)
        printf("%04x:\n", radio().id(), NMapping[i]);
    for(unsigned int j=0; j<NMapping.size(); j++)
        p = NMatrix[i][j];
    
    p = (p == POWER_NULL_VALUE ? POWER_NULL_VALUE : (p * p) / 10000);
    printf("\n");
    
    #endif
    #endif

    #endif
}
APPENDIX A. CODE

540  #endif
541  }
542  // -------------------------------------------------------------
543  template<typename OsModel_P,
544          uint16_t MAX_NEIGHBORS,
545          typename Radio_P,
546          typename Timer_P>
547  inline void
548  LSPTopology<OsModel_P, MAX_NEIGHBORS, Radio_P, Timer_P>::
549  dijsktra()
550  {
551      unsigned int n = NMapping.size();
552      int p;
553      for(unsigned int i = 0; i < n; i++) {
554          S.push_back(false);
555          P.push_back(-1);
556          D.push_back(10000001);
557      }
558      D[0] = 0;
559      PQ.push(arc(-1,0,0));
560      while (not PQ.empty()) {
561          arc a = PQ.pop();
562          unsigned int v = a.v;
563          if (not S[v]) {
564              S[v] = true;
565              for(unsigned int i = 0; i < n; i++) {
566                  p = NMatrix[v][i];
567                  if (not S[i]) {
568                      p = p * p;
569                      if (D[i] > D[v] + p) {
570                          D[i] = D[v] + p;
571                          P[i] = v;
572                          PQ.push(arc(v,i,D[i]));
573                      }
574                  }
575              }
576              if (not S[i]) {
577                  p = p * p;
578                  if (D[i] > D[v] + p) {
579                      D[i] = D[v] + p;
580                      P[i] = v;
581                      PQ.push(arc(v,i,D[i]));
582              }
583          }
584      }
585  #ifdef DEBUG_TOPOLOGY_LSP
586  #ifdef DEBUG_TOPOLOGY_LSP_V
587  #ifdef DEBUG_TOPOLOGY_LSP_VV
588      printf("%04x:␣", radio().id());
589      for(unsigned int i=0; i < n; i++) {
590          printf("%04x␣", NMapping[i]);
591      } printf("\n");
592      printf("%04x:␣", radio().id());
593      for(unsigned int i=0; i < n; i++) {
594          printf("%4i␣", P[i]);
595      } printf("\n");
596      printf("%04x:␣", radio().id());
597      for(unsigned int i=0; i < n; i++) {
598          printf("%4i␣", D[i]);
599      }
600  #endif
601  #endif
602  #endif
603  #endif
```c
602         printf("\%4i \n", D[i] / 10000);
603     }
604     printf("\n");
605 #endif
606 #endif
607 #endif
608 }
609 }
610 #endif

algorithms/topology/lsp/lsp_topology_message.h
1 #ifndef __ALGORITHMS_TOPOLOGY_LSP_TOPOLOGY_MESSAGE_H__
2 #define __ALGORITHMS_TOPOLOGY_LSP_TOPOLOGY_MESSAGE_H__
3 #include "util/serialization/simple_types.h"
4 namespace wiselib
5 {
6     template< typename OsModel_P,
7             typename Radio_P>
8     class LSPTopologyMessage
9     {
10         public:
11         typedef OsModel_P OsModel;
12         typedef Radio_P Radio;
13         typedef typename Radio::block_data_t block_data_t;
14         typedef typename Radio::node_id_t node_id_t;
15
16         uint8_t const static HELLO_SIZE = sizeof(uint8_t) + sizeof(int);
17         uint8_t const static ACK_SIZE = sizeof(uint8_t) + sizeof(int);
18         uint8_t const static INFO_SIZE = sizeof(uint8_t) + sizeof(int)
19             + sizeof(node_id_t);
20
21         // ----------------------------------------------------------
22         inline LSPTopologyMessage( uint8_t id );
23         // ----------------------------------------------------------
24         inline uint8_t msg_id() { return read<OsModel, block_data_t, uint8_t>( buffer ); };
25         // ----------------------------------------------------------
26         inline void set_msg_id( uint8_t id ) { write<OsModel, block_data_t, uint8_t>( buffer, id ); };
27         // ----------------------------------------------------------
28         inline int power() { return (int)read<OsModel, block_data_t, int> ( buffer + POWER_POS); }
29         // ----------------------------------------------------------
30         inline void set_power( int power ){ write<OsModel, block_data_t, int> (buffer + POWER_POS, power); }
31         // ----------------------------------------------------------
32         inline node_id_t other_id() { return read<OsModel, block_data_t, node_id_t> (buffer + OTHER_ID_POS); }
```
inline void set_other_id( node_id_t oid )
{
    write<OsModel, block_data_t, node_id_t>(buffer + OTHER_ID_POS, oid);
}

private:

enum data_positions
{
    MSG_ID_POS = 0,
    POWER_POS = MSG_ID_POS + sizeof(uint8_t),
    OTHER_ID_POS = (POWER_POS + sizeof(int))
};

uint8_t buffer[sizeof(uint8_t) + sizeof(int) +
sizeof(node_id_t)];

// ------------------------------------------------------------------
// ------------------------------------------------------------------
// ------------------------------------------------------------------

template<typename OsModel_P, typename Radio_P>
LSPTopologyMessage<OsModel_P, Radio_P>::
LSPTopologyMessage( uint8_t id )
{
    set_msg_id( id );
}

#endif
Appendix B

Results tables
## APPENDIX B. RESULTS TABLES

Table B.1: Neighbours per node

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Avg (%)</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>1</td>
<td>17</td>
<td>8.392</td>
<td>100.000</td>
<td>3.337</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>1</td>
<td>14</td>
<td>6.069</td>
<td>72.328</td>
<td>2.085</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>1</td>
<td>14</td>
<td>5.406</td>
<td>64.425</td>
<td>2.146</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>14</td>
<td>4.213</td>
<td>50.201</td>
<td>2.223</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>1</td>
<td>14</td>
<td>7.027</td>
<td>83.743</td>
<td>2.469</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>1</td>
<td>14</td>
<td>6.126</td>
<td>73.006</td>
<td>2.223</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>1</td>
<td>13</td>
<td>5.728</td>
<td>68.264</td>
<td>2.476</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>13</td>
<td>4.120</td>
<td>49.097</td>
<td>2.158</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>8</td>
<td>3.484</td>
<td>41.520</td>
<td>1.398</td>
</tr>
<tr>
<td><strong>Medium Low density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>8</td>
<td>63</td>
<td>34.517</td>
<td>100.000</td>
<td>10.874</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>3</td>
<td>35</td>
<td>10.445</td>
<td>30.260</td>
<td>6.314</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>2</td>
<td>34</td>
<td>8.978</td>
<td>26.010</td>
<td>4.871</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>21</td>
<td>5.844</td>
<td>16.931</td>
<td>2.788</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>4</td>
<td>43</td>
<td>13.703</td>
<td>39.698</td>
<td>6.792</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>3</td>
<td>39</td>
<td>12.454</td>
<td>32.578</td>
<td>5.176</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>3</td>
<td>36</td>
<td>10.201</td>
<td>29.555</td>
<td>5.230</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>21</td>
<td>5.808</td>
<td>16.826</td>
<td>2.740</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>14</td>
<td>4.753</td>
<td>13.770</td>
<td>1.635</td>
</tr>
<tr>
<td><strong>Medium High density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>13</td>
<td>101</td>
<td>59.372</td>
<td>100.000</td>
<td>17.407</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>3</td>
<td>55</td>
<td>12.427</td>
<td>20.931</td>
<td>10.605</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>2</td>
<td>51</td>
<td>10.249</td>
<td>17.263</td>
<td>7.355</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>23</td>
<td>6.540</td>
<td>11.015</td>
<td>2.849</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>4</td>
<td>62</td>
<td>16.503</td>
<td>27.795</td>
<td>11.943</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>2</td>
<td>54</td>
<td>12.747</td>
<td>21.469</td>
<td>7.601</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>2</td>
<td>52</td>
<td>11.276</td>
<td>18.992</td>
<td>7.883</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>23</td>
<td>6.241</td>
<td>10.511</td>
<td>2.760</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>15</td>
<td>5.445</td>
<td>9.171</td>
<td>1.869</td>
</tr>
<tr>
<td><strong>High density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>20</td>
<td>145</td>
<td>86.326</td>
<td>100.000</td>
<td>25.423</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>3</td>
<td>74</td>
<td>13.975</td>
<td>16.188</td>
<td>14.555</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>2</td>
<td>70</td>
<td>11.140</td>
<td>12.904</td>
<td>9.722</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>25</td>
<td>7.152</td>
<td>8.284</td>
<td>3.092</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>4</td>
<td>80</td>
<td>18.832</td>
<td>21.815</td>
<td>16.573</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>3</td>
<td>70</td>
<td>13.908</td>
<td>16.111</td>
<td>9.949</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>3</td>
<td>70</td>
<td>11.932</td>
<td>13.822</td>
<td>10.327</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>31</td>
<td>6.612</td>
<td>7.659</td>
<td>2.910</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>16</td>
<td>5.976</td>
<td>6.923</td>
<td>1.949</td>
</tr>
</tbody>
</table>
Table B.2: Distances for every two pair of nodes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Max</th>
<th>Low density</th>
<th>Medium Low density</th>
<th>Medium High density</th>
<th>High density</th>
</tr>
</thead>
<tbody>
<tr>
<td>No TC</td>
<td>6</td>
<td>2.004</td>
<td>1.000</td>
<td>0.930</td>
<td>4</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>7</td>
<td>2.444</td>
<td>1.219</td>
<td>1.172</td>
<td>8</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>7</td>
<td>2.582</td>
<td>1.288</td>
<td>1.239</td>
<td>9</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>11</td>
<td>3.052</td>
<td>1.523</td>
<td>1.573</td>
<td>13</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>7</td>
<td>2.196</td>
<td>1.095</td>
<td>1.016</td>
<td>7</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>7</td>
<td>2.355</td>
<td>1.175</td>
<td>1.089</td>
<td>8</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>7</td>
<td>2.513</td>
<td>1.254</td>
<td>1.207</td>
<td>8</td>
</tr>
<tr>
<td>LSP</td>
<td>11</td>
<td>3.345</td>
<td>1.669</td>
<td>1.726</td>
<td>16</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>7</td>
<td>2.196</td>
<td>1.095</td>
<td>1.016</td>
<td>8</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>7</td>
<td>2.355</td>
<td>1.175</td>
<td>1.089</td>
<td>9</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>7</td>
<td>2.513</td>
<td>1.254</td>
<td>1.207</td>
<td>9</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>11</td>
<td>4.694</td>
<td>2.568</td>
<td>1.960</td>
<td>11</td>
</tr>
</tbody>
</table>
# APPENDIX B. RESULTS TABLES

Table B.3: Transmission Power

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>100</td>
<td>100</td>
<td>100.000</td>
<td>0</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>9</td>
<td>100</td>
<td>67.112</td>
<td>25.879</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>1</td>
<td>100</td>
<td>55.451</td>
<td>24.650</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>100</td>
<td>39.743</td>
<td>22.288</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>16</td>
<td>100</td>
<td>77.533</td>
<td>23.545</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>9</td>
<td>100</td>
<td>62.741</td>
<td>24.135</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>4</td>
<td>100</td>
<td>59.076</td>
<td>21.152</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>100</td>
<td>39.238</td>
<td>21.647</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>100</td>
<td>46.312</td>
<td>25.188</td>
</tr>
<tr>
<td><strong>Medium Low density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>100</td>
<td>100</td>
<td>100.000</td>
<td>0</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>1</td>
<td>100</td>
<td>34.572</td>
<td>32.607</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>1</td>
<td>100</td>
<td>26.166</td>
<td>22.713</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>49</td>
<td>13.758</td>
<td>8.795</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>1</td>
<td>100</td>
<td>45.084</td>
<td>34.824</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>1</td>
<td>100</td>
<td>31.813</td>
<td>23.334</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>1</td>
<td>100</td>
<td>31.188</td>
<td>21.710</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>49</td>
<td>14.390</td>
<td>9.156</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>64</td>
<td>18.560</td>
<td>10.417</td>
</tr>
<tr>
<td><strong>Medium High density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>100</td>
<td>100</td>
<td>100.000</td>
<td>0</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>1</td>
<td>100</td>
<td>24.977</td>
<td>31.631</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>1</td>
<td>100</td>
<td>17.720</td>
<td>20.291</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>49</td>
<td>8.927</td>
<td>5.352</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>1</td>
<td>100</td>
<td>33.229</td>
<td>35.459</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>1</td>
<td>100</td>
<td>21.311</td>
<td>20.844</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>1</td>
<td>100</td>
<td>20.839</td>
<td>19.207</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>49</td>
<td>9.326</td>
<td>5.640</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>64</td>
<td>12.476</td>
<td>6.895</td>
</tr>
<tr>
<td><strong>High density</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>100</td>
<td>100</td>
<td>100.000</td>
<td>0</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>1</td>
<td>100</td>
<td>20.473</td>
<td>30.829</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>1</td>
<td>100</td>
<td>13.722</td>
<td>18.767</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>1</td>
<td>49</td>
<td>6.850</td>
<td>4.121</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>1</td>
<td>100</td>
<td>27.737</td>
<td>35.190</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>1</td>
<td>100</td>
<td>16.506</td>
<td>19.183</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>1</td>
<td>100</td>
<td>16.191</td>
<td>17.923</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>1</td>
<td>49</td>
<td>7.107</td>
<td>4.383</td>
</tr>
<tr>
<td>LSP</td>
<td>1</td>
<td>36</td>
<td>9.398</td>
<td>5.078</td>
</tr>
</tbody>
</table>
## Table B.4: Traffic simulation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg</th>
<th>Avg (%)</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low density</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>2003368.211</td>
<td>100.000</td>
<td>1991524.702</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>150892.947</td>
<td>0.753</td>
<td>21354.637</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>130990.316</td>
<td>0.654</td>
<td>21575.891</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>110622.895</td>
<td>0.552</td>
<td>17058.559</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>157350.211</td>
<td>0.785</td>
<td>22970.370</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>134742.368</td>
<td>0.673</td>
<td>22262.194</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>137806.000</td>
<td>0.688</td>
<td>20667.553</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>111143.895</td>
<td>0.556</td>
<td>16565.616</td>
</tr>
<tr>
<td>LSP</td>
<td>141153.105</td>
<td>0.705</td>
<td>24525.536</td>
</tr>
<tr>
<td><strong>Medium Low density</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>1856900.000</td>
<td>100.000</td>
<td>578557.689</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>69673.200</td>
<td>0.375</td>
<td>4981.845</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>65306.050</td>
<td>0.352</td>
<td>4325.702</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>54637.850</td>
<td>0.294</td>
<td>5608.683</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>81357.900</td>
<td>0.438</td>
<td>6203.000</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>72582.050</td>
<td>0.391</td>
<td>4739.228</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>75089.200</td>
<td>0.404</td>
<td>5334.423</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>55866.900</td>
<td>0.301</td>
<td>5466.495</td>
</tr>
<tr>
<td>LSP</td>
<td>83408.700</td>
<td>0.449</td>
<td>6265.396</td>
</tr>
<tr>
<td><strong>Medium High density</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>1848950.000</td>
<td>100.000</td>
<td>433272.143</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>49855.650</td>
<td>0.270</td>
<td>2576.512</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>46880.500</td>
<td>0.254</td>
<td>2719.868</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>40695.000</td>
<td>0.220</td>
<td>2581.605</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>59393.300</td>
<td>0.321</td>
<td>2927.35</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>53055.250</td>
<td>0.287</td>
<td>2368.298</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>55315.900</td>
<td>0.299</td>
<td>2584.994</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>43900.800</td>
<td>0.237</td>
<td>3300.109</td>
</tr>
<tr>
<td>LSP</td>
<td>62113.950</td>
<td>0.336</td>
<td>4597.602</td>
</tr>
<tr>
<td><strong>High density</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No TC</td>
<td>1822200.000</td>
<td>100.000</td>
<td>387538.385</td>
</tr>
<tr>
<td>CBTC Basic 5/6</td>
<td>39376.650</td>
<td>0.216</td>
<td>1994.566</td>
</tr>
<tr>
<td>CBTC Shrinkback 5/6</td>
<td>36847.150</td>
<td>0.202</td>
<td>1965.562</td>
</tr>
<tr>
<td>CBTC Pairwise 5/6</td>
<td>34956.300</td>
<td>0.192</td>
<td>1516.450</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>47570.800</td>
<td>0.261</td>
<td>2274.685</td>
</tr>
<tr>
<td>CBTC Shrinkback 2/3</td>
<td>42074.850</td>
<td>0.231</td>
<td>2206.326</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>44056.400</td>
<td>0.242</td>
<td>2180.343</td>
</tr>
<tr>
<td>CBTC Pairwise 2/3</td>
<td>36986.800</td>
<td>0.203</td>
<td>1640.279</td>
</tr>
<tr>
<td>LSP</td>
<td>52818.400</td>
<td>0.290</td>
<td>3963.389</td>
</tr>
</tbody>
</table>
Table B.5: Number of messages sent

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Low density</th>
<th>Medium Low density</th>
<th>Medium High density</th>
<th>High density</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Std. Dev.</td>
<td>Avg</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>CBTC 5/6</td>
<td>29.368</td>
<td>2.795</td>
<td>39.579</td>
<td>3.066</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>35.421</td>
<td>2.561</td>
<td>54.368</td>
<td>3.375</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>35.789</td>
<td>2.567</td>
<td>55.316</td>
<td>3.540</td>
</tr>
<tr>
<td>LSP</td>
<td>51.895</td>
<td>4.128</td>
<td>166.450</td>
<td>8.703</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>44.737</td>
<td>3.338</td>
<td>153.789</td>
<td>8.799</td>
</tr>
<tr>
<td>CBTC 5/6</td>
<td>39.579</td>
<td>3.066</td>
<td>44.105</td>
<td>3.972</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>54.368</td>
<td>3.375</td>
<td>60.300</td>
<td>5.120</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>55.316</td>
<td>3.540</td>
<td>61.684</td>
<td>5.120</td>
</tr>
<tr>
<td>LSP</td>
<td>166.450</td>
<td>8.703</td>
<td>276.474</td>
<td>7.639</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>153.789</td>
<td>8.799</td>
<td>263.474</td>
<td>7.639</td>
</tr>
<tr>
<td>CBTC 5/6</td>
<td>44.105</td>
<td>3.972</td>
<td>47.895</td>
<td>3.932</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>60.300</td>
<td>5.120</td>
<td>68.158</td>
<td>4.671</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>61.684</td>
<td>5.120</td>
<td>69.947</td>
<td>4.740</td>
</tr>
<tr>
<td>LSP</td>
<td>276.474</td>
<td>7.639</td>
<td>383.947</td>
<td>10.308</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>263.474</td>
<td>7.639</td>
<td>370.947</td>
<td>10.308</td>
</tr>
</tbody>
</table>
### Table B.6: Estimated cost of messages sent

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low density</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBTC 5/6</td>
<td>12078.316</td>
<td>1324.951</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>14921.737</td>
<td>1237.228</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>14990.211</td>
<td>1256.747</td>
</tr>
<tr>
<td>CBTC 5/6 (No position)</td>
<td>7246.842</td>
<td>794.918</td>
</tr>
<tr>
<td>CBTC Basic 2/3 (No position)</td>
<td>8952.789</td>
<td>742.220</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3 (No position)</td>
<td>9021.368</td>
<td>762.114</td>
</tr>
<tr>
<td>LSP</td>
<td>11008.684</td>
<td>970.412</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>9266.579</td>
<td>813.065</td>
</tr>
<tr>
<td><strong>Medium Low density</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBTC 5/6 (No position)</td>
<td>17905.474</td>
<td>1498.813</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>25083.053</td>
<td>1713.042</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>25132.211</td>
<td>1718.306</td>
</tr>
<tr>
<td>CBTC 5/6 (No position)</td>
<td>10743.053</td>
<td>899.387</td>
</tr>
<tr>
<td>CBTC Basic 2/3 (No position)</td>
<td>15049.579</td>
<td>1027.816</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3 (No position)</td>
<td>15098.789</td>
<td>1033.056</td>
</tr>
<tr>
<td>LSP</td>
<td>33845.850</td>
<td>1746.646</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>31028.053</td>
<td>1762.654</td>
</tr>
<tr>
<td><strong>Medium High density</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBTC 5/6</td>
<td>20538.000</td>
<td>1995.859</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>28371.105</td>
<td>2549.839</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>28414.368</td>
<td>2552.225</td>
</tr>
<tr>
<td>CBTC 5/6 (No position)</td>
<td>12322.421</td>
<td>1197.424</td>
</tr>
<tr>
<td>CBTC Basic 2/3 (No position)</td>
<td>17050.050</td>
<td>1495.999</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3 (No position)</td>
<td>17065.842</td>
<td>1532.085</td>
</tr>
<tr>
<td>LSP</td>
<td>54986.316</td>
<td>1581.851</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>52086.316</td>
<td>1581.851</td>
</tr>
<tr>
<td><strong>High density</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBTC 5/6</td>
<td>22695.947</td>
<td>1884.073</td>
</tr>
<tr>
<td>CBTC Basic 2/3</td>
<td>32579.158</td>
<td>2336.608</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3</td>
<td>32622.000</td>
<td>2341.825</td>
</tr>
<tr>
<td>CBTC 5/6 (No position)</td>
<td>13671.316</td>
<td>1130.478</td>
</tr>
<tr>
<td>CBTC Basic 2/3 (No position)</td>
<td>19547.316</td>
<td>1402.003</td>
</tr>
<tr>
<td>CBTC Asymmetric 2/3 (No position)</td>
<td>19590.316</td>
<td>1407.477</td>
</tr>
<tr>
<td>LSP</td>
<td>75385.474</td>
<td>2072.442</td>
</tr>
<tr>
<td>LSP (1 info message)</td>
<td>72485.474</td>
<td>2072.442</td>
</tr>
</tbody>
</table>
Bibliography


