Model-based Cognitive Communications for Low-power Wireless Networks

Indika Sanjeewa Abeywickrama Dhanapala
Department of Electrical and Electronic Engineering, Munster Technological University, Cork, Ireland, i.s.a.dhanapala@mycit.ie

Follow this and additional works at: https://sword.cit.ie/allthe

Part of the Electrical and Electronics Commons

Recommended Citation

This Doctoral Thesis is brought to you for free and open access by the Dissertations and Theses at SWORD - South West Open Research Deposit. It has been accepted for inclusion in Theses by an authorized administrator of SWORD - South West Open Research Deposit. For more information, please contact sword@cit.ie.
Model-based Cognitive Communications for Low-power Wireless Networks

Indika Sanjeewa Abeywickrama Dhanapala

Nimbus Centre for Embedded Systems Research
Department of Electrical & Electronic Engineering
Munster Technological University

Supervised by: Prof. Dirk Pesch
Dr. Ramona Marfievici

This dissertation is submitted for the degree of

Doctor of Philosophy

Munster Technological University January 2022
Declaration

This thesis is entirely the candidate’s own work except where otherwise accredited and has not been submitted for an award at any other institution.

The research presented in this thesis was carried out in compliance with the MTU Code of Good Practice in Research.

Candidate: ___________________________ Date: 10/01/2022

Principal Supervisor: ___________________________ Date: 10/01/2022

Second Supervisor: ___________________________ Date: 10/01/2022
To my family
Acknowledgements

This doctoral thesis is the result of a long journey that began years ago in Bremen after finishing my master studies at the University of Bremen. During all these years, I had the privilege to work with amazing people and it is my pleasure to acknowledge them who were instrumental in the completion of my PhD research.

I begin by thanking my supervisor Prof Dirk Pesch for giving me this opportunity to work on my doctoral thesis under his supervision. Starting with the application to the scholarship, the support you gave me to successfully finish this thesis is tremendous. Guidance from the beginning of this research, your insightful comments, and constant moral supports are unforgettable.

Next, I sincerely thank Dr Ramona Marfievici for becoming my co-supervisor. I cannot appreciate enough the motivation and support that you gave me during the tenure of this thesis. There is no doubt that the discussions we (including Dr Sameeraya Palipana) had in the mornings with a coffee in hand and the heated long arguments in the evenings positively contributed to the success of this thesis.

I would like to kindly thank Dr Stephen Brown and Dr Fabrice Theoleyre, who agreed to be the examiners of my thesis. Your insightful comments and feedback have been invaluable to further strengthen my thesis during the final stage of the writing process.

I am grateful to the team in the then United Technologies Research Centre, Cork, for the guidance at the beginning of this research. The support given by Dr Rodolfo de Paz Alborela, Dr Piyush Agrawal, and Dr Phil Haris is not forgotten.

I would like to thank the late Prof Carmelita Görg who was my supervisor during my master studies at the University of Bremen for believing in me and for her tremendous support and guidance that eventually bridged my way to Cork. Being part of her research group was truly a remarkable and unforgettable experience.

Over the past few years, I had the most delightful company of my colleagues, Dr Sameera Palipana, Dr Pablo Corbalán, Dr David Rojas, Roland Katona, Dylan
Smyth, and Dr Victor Cionca. It was a great pleasure working with you and thank you all for supporting me in numerous ways during the last few years.

I thank Dr Bernd-Ludwig Wenning, Dr Andreas Könsgen, Dr Koojana Kuladinithi, Dr Asanga Udugama, and Irangani Könsgen for all the motivation, support and guidance especially at the beginning of this long journey.

I would also like to thank the Nimbus team, especially, Prof John Barrett, Richard Linger, Dr Susan Rea, Dr Alex Vakaloudis, Juan Francisco Martinez, Brian Cahill and Dave O’Leary for supporting me in numerous ways during my difficult times.

A special thank goes to my wife Praveeni Kaluarachchi for her love, patience and invaluable support by numerous means over the last few years. I am grateful for your understanding and spiritual and moral support given.

Finally, it is with reverence and utmost respect that I remember my parents for creating an environment for me to pursue my education with all the hardships and giving me the strength I needed to take the next steps toward my dreams. Without your love, dedication, guidance and support, I wouldn’t have come thus far.

Cork, 10.01.2022

Indika Sanjeewa Abeywickrama Dhanapala

The research leading to this doctoral thesis has been funded by the Irish Research Council in collaboration with the then United Technologies Research Centre, Cork, Ireland. The funding was the result of the award “Enterprise Partnership Scheme Postgraduate Scholarship 2014” under the project ID EPSPG/2014/66 (Cognitive Radio Communication Framework for Wireless Sensor Networks).
Abstract

Our world is increasingly being instrumented with low-power wireless networks that deliver data for diverse application domains such as energy management, health and well-being, security, and other Internet of Things services. As these wireless networks operate in the unlicensed radio spectrum while co-existing with various other communication networks, their communication reliability and energy efficiency are affected by the radio environment. Because of this vulnerability to radio interference, low-power wireless networks may be unable to deliver their application requirements and be inefficiently consuming energy, functioning as a low-dependability network. The situation is exacerbated in indoor environments such as office and residential buildings, wherein collocated wireless devices and electrical appliances impair packet reception, further reducing communication reliability and energy efficiency.

The performance of co-existing low-power wireless networks can be increased with a profound understanding of Radio Frequency (RF) noise in the operating environment and by designing more reliable and energy-efficient communication solutions.

In this thesis, to better understand the sources of Cross-Technology Interference (CTI), a large set of noise traces were collected using real-world scenarios. Because Wi-Fi is the dominant source of indoor RF noise, initially Wi-Fi traces were collected in order to investigate the impact of Wi-Fi activities on the performance of low-power wireless networks. As low-power wireless devices perceive noise from sources other than Wi-Fi and a handful of work has been done in that regard, the focus of the thesis was moved toward the sensor node’s perspectives wherein how the sensor nodes perceive noise from unknown sources is studied. As the analysis of the traces demonstrated the existence of noise patterns, the traces were further exploited to model their statistical distributions. The accuracy of the noise model to capture wireless activities and the performance of the white-space model to accurately predict transmissions opportunities, also known as white-spaces, are the key to the proposed approach. These models motivated the
design of a packet-loss-aware proactive MAC protocol (LUCID) for low-power wireless networks. LUCID was designed for periodic data applications and was evaluated in realistic simulations with varying application data rates and network sizes. LUCID achieves slight performance improvements w.r.t. the state-of-the-art CRYSTAL technique, showing a 1.2% increase in packet delivery ratio, 0.02% decrease in duty-cycle, and 7.4% more energy efficient under bursty indoor radio frequency noise. All these promising results are achieved with a high energy budget required for collecting noise measurements and training the models preferably in a separate identical network. Despite the modest amelioration of the performance of low-power wireless networks, LUCID opens new research directions for further improving the performance of wireless communication networks in general.

In summary, this thesis presents a mechanism to analyse patterns in noise traces, a mechanism to use the noise patterns to predict noise-free opportunities for transmission, and a protocol (LUCID) that uses the predicted transmission opportunities to identify rendezvous points for low-interference communication. Investigations presented in this thesis do help to enhance the performance of low-power wireless networks by LUCID, in which nodes utilise the predicted transmission opportunities in a model-based receiver-aware setting.

Keywords
Low-power Wireless Networks, Receiver-aware Communication, Proactive Medium Access, Models, Interference-aware Communication, Measurements, Prediction
## Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of figures</td>
<td>xii</td>
</tr>
<tr>
<td>List of tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xv</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Terminology</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Goals</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Publications</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Thesis Outline</td>
<td>9</td>
</tr>
<tr>
<td><strong>2 Background and State of the Art</strong></td>
<td>10</td>
</tr>
<tr>
<td>2.1 Noise Detection and Classification</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Modelling of Radio Frequency Noise</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Cognitive Radio Communication Solutions</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Low-power Reliable Data Collection</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Taxonomy of CTI Solutions</td>
<td>19</td>
</tr>
<tr>
<td><strong>3 Collecting and Understanding Radio Frequency Noise</strong></td>
<td>24</td>
</tr>
<tr>
<td>3.1 IEEE 802.11 Noise</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1 Measurement Locations</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2 Hardware and Software Platforms</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3 Wi-Fi Measurement Execution</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Characterising IEEE 802.11 Noise</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 Preliminaries</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2 Estimating Wi-Fi Noise</td>
<td>32</td>
</tr>
</tbody>
</table>
### Table of contents

3.2.3 Aggregated Wi-Fi Noise ........................................... 34
3.2.4 MMPP(2) Model Performance ................................. 37
3.3 Noise from Unknown Source ................................. 41
  3.3.1 Measurement Locations ................................. 41
  3.3.2 Hardware and Software Platforms ..................... 41
  3.3.3 Measurement Execution ................................. 43
3.4 Characterising Noise from Unknown Sources ............... 45
  3.4.1 Metrics ..................................................... 45
  3.4.2 Methodology ............................................. 46
  3.4.3 Noise Patterns ......................................... 48
3.5 Summary ....................................................... 52

4 Estimating Noise from Unknown Sources and Predicting White-spaces .......................... 54
  4.1 Noise/White-space Models ................................. 54
    4.1.1 Gaussian Mixture Model .............................. 55
    4.1.2 Hidden Markov Model .............................. 57
  4.2 Estimating Model Parameters ................................. 60
  4.3 Performance ............................................. 62
    4.3.1 Metrics ............................................. 62
    4.3.2 Noise Estimation ..................................... 63
    4.3.3 White-space Prediction ............................. 67
  4.4 Summary ..................................................... 71

5 Model-based Receiver-aware Communication .................... 73
  5.1 Overview of the Design ...................................... 75
  5.2 Deployment .................................................. 77
    5.2.1 Measure Noise ...................................... 78
    5.2.2 Characterise Noise ................................... 79
    5.2.3 Training Models ..................................... 79
  5.3 Model-based Data Communication ............................ 79
    5.3.1 Network Initialisation ................................ 80
      5.3.1.1 Time Synchronisation ................................ 80
      5.3.1.2 Routing and Model Exchange .................... 83
    5.3.2 Model Parameter Selection ............................ 84
    5.3.3 Receiver-aware Communication ....................... 85
    5.3.4 PDR Monitoring and Feedback Loop .................... 89
Table of contents

5.4 Performance of LUCID

  5.4.1 Proof of Concept Implementation
  5.4.2 Simulation Configurations and Execution
  5.4.3 Performance Evaluation
    5.4.3.1 Model Re-parametrisation
    5.4.3.2 Impact of Slot-length
    5.4.3.3 Comparison of LUCID with CRYSTAL and ContikiMAC
      5.4.3.3.1 5-node Network
      5.4.3.3.2 16-node Network

5.5 Summary

6 Conclusions and Outlook

  6.1 Contributions
  6.2 Lessons Learned
  6.3 Limitations and Further Research Directions

References

Appendix A Snippets of Measured Noise Traces

  A.1 OFFICE Peak
  A.2 OFFICE Off-peak
  A.3 HOME Peak
  A.4 HOME Off-peak

Appendix B Source Code

  B.1 Contiki
    B.1.1 Application
      B.1.1.1 Data Collection Application
      B.1.1.2 Project Configuration File
      B.1.1.3 Makefile
    B.1.2 LUCID
      B.1.2.1 LUCID Code
      B.1.2.2 LUCID Header File
    B.1.3 Noise Measuring Tools
      B.1.3.1 Scanner
      B.1.3.2 Timestamper
  B.2 MATLAB
Table of contents

B.2.1 Starting Script .................................. 254
B.2.2 Cooja Callback Function .......................... 258
B.2.3 Train White Space Model ....................... 263
B.2.4 Pre-compute White Spaces ...................... 266
B.2.5 Obtain Relative Free Slot Index ............... 269

Appendix C JamLab Settings ............................. 272

Appendix D Packet Formats .............................. 302

Appendix E Simulation Log .............................. 303

Appendix F Simulation Results .......................... 307
List of figures

2.1 Principle of ContikiMAC. ........................................ 17
2.2 Operation of CRYSTAL. ........................................ 18
2.3 Taxonomy of CTI solutions. ................................... 20
3.1 Overlapping Wi-Fi and IEEE 802.15.4 channels. .......... 25
3.2 The test-bed for characterising Wi-Fi noise. ............... 27
3.3 Burstiness of Wi-Fi noise. ..................................... 29
3.4 Self-similarity of Wi-Fi aggregated noise. ................... 30
3.5 MMPP(2) noise model. .......................................... 31
3.6 second-order hyperexponential traffic model. ............... 33
3.7 Aggregated Wi-Fi noise model ................................ 35
3.8 System calibration: variation of MMPP(2) metrics with x and k. 37
3.9 Comparison of percentage RMSE. ............................ 39
3.10 Experimentation setup to capture noise from unknown sources. 42
3.11 TMote Sky node placement .................................... 43
3.12 TMote Sky node placement in OFFICE ......................... 44
3.13 False discovery of BUSY periods ............................. 46
3.14 IEEE 802.15.4 PHY frame structure [2]. .................... 47
3.15 Probability distribution function of noise traces from FIRST on channel 13 (left), 18 (center) and 23 (right). .......... 49
3.16 Probability distribution function of noise traces from SECOND at location 1 (left), 2 (center), and 3 (right). ............... 49
3.17 Probability distribution function of noise traces during THIRD from the OFFICE and the HOME. .......................... 50
3.18 Noise patterns in the OFFICE and the HOME for two weeks. .... 51
4.1 AUC vs. number of components in GMM .................... 56
4.2 Prediction mechanism. .......................................... 58
4.3 Bayesian representation of a hidden Markov model. 58
4.4 Hidden Markov model. 60
4.5 Performance of the GMM model with $M = 7$. 64
4.6 Performance of GMM on different channels at the same location (FIRST). 65
4.7 Performance of GMM on channel 18 at varying locations (SECOND). 65
4.8 Performance of GMM on channel 18 for long-term noise (THIRD). 65
4.9 White-space model performance prediction and the PLR of a 1 s (top) and 60 s (middle) data rate application in FIRST and SECOND, hourly variations of noise (bottom). 70
4.10 White-space model performance prediction and the PLR of a 1 s (top) and 60 s (middle) data rate application in THIRD, hourly variations of noise (bottom). 71
5.1 Design overview of LUCID. 75
5.2 Overview of the deployment phase. 78
5.3 Overview of the model-based data communication. 80
5.4 Timeline of LUCID events starting from the deployment. 80
5.5 Timeline of clock synchronisation. 82
5.6 Receiver-aware communication with white-space models. 87
5.7 Flowchart of the implementation of LUCID in Contiki/COOJA. 93
5.8 Protocol stack. 94
5.9 System integration with COOJA and MATLAB. 94
5.10 Simulation scenarios with two RF noise generators. 96
5.11 Network topologies. 101
5.12 Change in PDR, duty-cycle, and energy-efficiency of LUCID with varying slot lengths. 107
5.13 Comparison of PDR, duty-cycle, and energy-efficiency in the 5-node network with the SoA. 109
5.14 Comparison of PDR, duty-cycle, and energy-efficiency in the 16-node network with the SoA. 111
D.1 MAC control packet format. 302
D.2 White space model exchange packet format. 302
## List of tables

2.1  Classified solutions according to the taxonomy in Figure 2.3. . . . 21
3.1  Stationarity test. .................................................. 29
3.2  Characteristics of aggregated Wi-Fi noise .......................... 36
3.3  Comparison of absolute RMSE. .................................. 40
3.4  NCLR comparison of the noise traces from FIRST and SECOND. . 48
4.1  Confusion matrix for channel states. .............................. 62
4.2  Performance of the GMM model with $\mathcal{M} = 7$. ............... 63
4.3  GMM vs. alternative solutions in THIRD. .......................... 66
4.4  Comparison of white-space model prediction performance to alter-
     native solutions. ...................................................... 68
4.5  Comparison of white-space model prediction performance to alter-
     native solutions: confidence intervals. .......................... 68
5.1  Configurations of LUCID. ........................................... 99
5.2  Simulation execution. ............................................... 100
5.3  EMA statistics without model re-parametrisation. ................. 103
5.4  EMA statistics with model re-parametrisation. ...................... 104
F.1  Comparison of PDR with the SoA. ................................. 308
F.2  Comparison of duty-cycle with the SoA. .......................... 309
F.3  Comparison of energy efficiency (in terms of $PDR$/duty-cycle) with
     the SoA. ................................................................. 310
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>AP</td>
<td>Access Point</td>
</tr>
<tr>
<td>AUC</td>
<td>Area Under Curve</td>
</tr>
<tr>
<td>BLE</td>
<td>Bluetooth Low Energy</td>
</tr>
<tr>
<td>CCA</td>
<td>Clear Channel Assessment</td>
</tr>
<tr>
<td>CCC</td>
<td>Common Control Channel</td>
</tr>
<tr>
<td>CDF</td>
<td>Cumulative Distribution Function</td>
</tr>
<tr>
<td>CR</td>
<td>Cognitive Radio</td>
</tr>
<tr>
<td>CRN</td>
<td>Cognitive Radio Network</td>
</tr>
<tr>
<td>CSMA/CA</td>
<td>Carrier-Sense Multiple Access with Collision Avoidance</td>
</tr>
<tr>
<td>CTI</td>
<td>Cross-Technology Interference</td>
</tr>
<tr>
<td>CTP</td>
<td>Collection Tree Protocol</td>
</tr>
<tr>
<td>CTS</td>
<td>Clear To Send</td>
</tr>
<tr>
<td>EM</td>
<td>Expectation Maximization</td>
</tr>
<tr>
<td>EMA</td>
<td>Exponential Moving Average</td>
</tr>
<tr>
<td>FN</td>
<td>False Negative</td>
</tr>
<tr>
<td>FP</td>
<td>False Positive</td>
</tr>
<tr>
<td>FPR</td>
<td>False Positive Rate</td>
</tr>
</tbody>
</table>
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMM</td>
<td>Gaussian Mixture Model</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>IAT</td>
<td>Inter-Arrival Time</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ISM</td>
<td>Industrial, Scientific and Medical</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>LPL</td>
<td>Low-Power Listening</td>
</tr>
<tr>
<td>LPP</td>
<td>Low-Power Probing</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MACA-BI</td>
<td>MACA By Invitation</td>
</tr>
<tr>
<td>MFR</td>
<td>MAC footer</td>
</tr>
<tr>
<td>MHR</td>
<td>MAC header</td>
</tr>
<tr>
<td>ML</td>
<td>Maximum Likelihood</td>
</tr>
<tr>
<td>MMPP</td>
<td>Markov-Modulated Poisson Process</td>
</tr>
<tr>
<td>MMPP(2)</td>
<td>second-order MMPP</td>
</tr>
<tr>
<td>NCLR</td>
<td>Normalised Cross-Likelihood Ratio</td>
</tr>
<tr>
<td>PDF</td>
<td>Probability Density Function</td>
</tr>
<tr>
<td>PDR</td>
<td>Packet Delivery Ratio</td>
</tr>
<tr>
<td>PLR</td>
<td>Packet Loss Ratio</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit</td>
</tr>
<tr>
<td>PTS</td>
<td>Prepare To Send</td>
</tr>
<tr>
<td>RDC</td>
<td>Radio Duty-Cycling</td>
</tr>
<tr>
<td>RI-MAC</td>
<td>Receiver-Initiated MAC</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio-Frequency IDentification</td>
</tr>
<tr>
<td>RMSE</td>
<td>Root Mean Square Error</td>
</tr>
<tr>
<td>RSS</td>
<td>Received Signal Strength</td>
</tr>
<tr>
<td>RSSI</td>
<td>Received Signal Strength Indicator</td>
</tr>
<tr>
<td>RTS</td>
<td>Ready To Send</td>
</tr>
<tr>
<td>SFD</td>
<td>Start Frame Delimiter</td>
</tr>
<tr>
<td>SINR</td>
<td>Signal to Interference Noise Ratio</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>TBNS</td>
<td>Time Between Noise Signals</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time-Division Multiple Access</td>
</tr>
<tr>
<td>TN</td>
<td>True Negative</td>
</tr>
<tr>
<td>TP</td>
<td>True Positive</td>
</tr>
<tr>
<td>TPR</td>
<td>True Positive Rate</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>Wireless Fidelity</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the last decade, the advancement of wireless Internet of Things (IoT) devices has caused the radio spectrum, especially the unlicensed 2.4 GHz ISM band, to be heavily crowded with smart wireless devices that are used in a wide range of application domains, such as smart cities and environments, smart grids, industrial automation, traffic management and logistics, healthcare and assisted living, agriculture and breeding, public safety and remote monitoring [57, 62, 119, 42]. Irrespective of their specific wireless communication technology, these IoT devices, e.g., sensors, actuators, Radio-frequency Identification (RFID) tags, and mobile phones, are part of the lives of millions of people every day. The rapid growth in the computing ability and the continuous decrease in the cost of electronic devices have paved the way for a new era where such devices can monitor and control everything that we use every day via the Internet.

Soon more and more such smart and interconnected devices will join the already crowded radio spectrum, which brings new challenges, in particular for low-power resource-constrained wireless devices to access the radio medium, to efficiently utilise the scarce frequency spectrum to fulfil the stringent requirements of the ever-growing applications in the IoT domain.

1.1 Terminology

Before proceeding with the contents, it is important to clear the reader’s mind with the terminology used in this thesis for dependability, radio frequency interference, noise, sensor node’s perspective, white-space, and Wi-Fi.

It is worth noting that according to the literature, dependability of systems comprises of multiple attributes, such as availability, reliability, safety, confiden-
1.2 Motivation

One of the major aims of the IoT paradigm is to access data about targeted objects and environments without human interaction as human-entered data has been seen as inaccurate and less cost-effective [90] especially in hard-to-reach places. Therefore, IoT devices, such as low-power wireless devices, are tasked with accurately acquiring required data for applications while maintaining the lifespan of the devices as high as possible to keep the human interaction at its minimum. To this end, high communication reliability and energy efficiency of devices are key requirements of IoT applications, leading to high dependability networks.

A low-power wireless network, such as one based on IEEE 802.15.4 networks, is a collection of wirelessly-interconnected, resource-constrained, distributed data sources that provide information about the environment in which they have been...
deployed. Upholding the communication reliability and energy efficiency of such a resource-constrained network is essential to fulfil its application requirements.

The original motivating use case of the work presented in this thesis is a design of a reliable in-building fire detection system. The fire detectors/sensor nodes in the fire detection system should periodically check for fire, e.g., by sampling the temperature of the building wherein they are deployed. Periodically generated data is conveyed to a network coordinator that interacts with a server. If a fire is detected, the sampling frequency of the detectors increase, which essentially generates a high amount of data. It is important to note that the radio environment in which the fire detectors communicate is high in noise which is mainly due to fire and other in-building communication systems that share the same frequency band with the fire detectors.

As it is well-known, when located in each other’s proximity, wireless communication devices can interfere with each other due to the broadcast nature of wireless transmissions. This phenomenon is called Cross-Technology Interference (CTI) which happens due to the inability of heterogeneous wireless communication devices to coordinate their transmissions in time and frequency. CTI is a major problem, especially in the 2.4 GHz unlicensed radio spectrum, as the said radio band is shared by many heterogeneous devices. Some of those devices operating in the 2.4 GHz unlicensed radio spectrum have high power transmitters, such as IEEE 802.11/Wi-Fi and IEEE 802.15.1/Bluetooth, and high power noise emissions, such as microwave ovens. All these high power uncoordinated emissions, in particular, affects the low-power devices [165, 70], such as low-power IEEE 802.15.4 networks, resulting in increased channel contentions and increased packet losses, which degrade their performance in terms of communication reliability and energy efficiency. Ultimately, CTI under-utilises the precious frequency spectrum as it consumes resources without being used for effective communications.

The research community has primarily approached mitigating CTI in IEEE 802.15.4 networks by focusing on the emission sources. Thus, this thesis has initially focused on IEEE 802.11 noise in the low-power wireless networks as Wi-Fi noise dominates the ISM bands with high transmission power (20 dBm). Nonetheless, transmissions of a sensor node could experience interference from multiple emission sources at any given time instance. Thus, focusing on only one source of noise could lead to under-performed networks. There are only a handful of works investigating this with the sensor node’s perspective [23, 74, 129], but they look only at one source of noise (i.e., Wi-Fi or IEEE 802.15.1/Bluetooth or
microwave ovens) affecting the sensor node. The subsequent study of the thesis has moved toward the sensor node’s perspectives to characterise noise from unknown sources rather than individual ones.

1.3 Thesis Goals

The objectives of the research presented here are to improve the performance of IEEE 802.15.4 networks in terms of reliability and energy efficiency and the efficiency of the spectrum usage while adapting to varying noise conditions.

High communication reliability and high energy efficiency make IEEE 802.15.4 networks suitable for the next generation of applications, especially in the IoT domain. In this thesis, a high dependability network is achieved through the design of a novel, decentralised, implicit noise-aware and receiver-aware, proactive medium access control mechanism, which analyses the noise in the environment in advance and estimates noise patterns to predict transmission opportunities for low-power wireless sensor nodes.

The predicted transmission opportunities, i.e., temporary vacant channel spaces in time also known as *temporal white-spaces*, are then utilised by the sensor nodes. Since the proposed technique schedules packet transmissions when noise is not predicted, it decreases packet corruptions, maximising the channel efficiency in the precious unlicensed frequency band.

The radio environment, in particular, the unlicensed band, is a highly dynamic setting as it is crowded by a massive number of devices using different technologies to access the wireless medium [19, 68, 71, 96, 23, 129]. If the transmission schedules of the sensor nodes do not adapt to noise in the radio environment, packet receptions will be interfered with, leading to packet corruptions and data losses. Therefore, adapting the communication algorithm and its parameters in line with dynamic noise is essential to improve the performance of low-power wireless networks.

1.4 Contributions

This section presents the key contributions of this thesis.

1. **A mechanism to analyse patterns in noise traces.** Devices operating in the 2.4 GHz ISM band can interfere with IEEE 802.15.4 networks. This could be from Wi-Fi, IEEE 802.15.1/Bluetooth, or any other device operating in
the aforementioned band generating noise. Thus, a sensor node deployed in
the same band perceives noise differently depending on the source of noise
as discussed in [71, 68, 171]. Therefore, by considering the source of noise
individually, efficient interference mitigation techniques can be developed,
e.g., [23, 129]. Even though these solutions perform well in the presence
of noise that they are tailored for, there is no countermeasure for the other
sources of noise. A robust interference mitigation technique should tackle
noise from all possible sources. There is only a handful of research done to
mitigate the effects of noise from unknown sources in IEEE 802.15.4 networks,
such as work done in [74].

Understanding noise patterns in the deployed environment is essential to mit-
igate the effects of noise from unknown sources. Therefore, to identify noise
patterns, the radio environment is sampled for collecting noise measurement
traces. These traces are characterised by the mean Time Between Noise
Signals (TBNS) and the number of identified noise signals within a selected
time period. The two-dimensional distribution of the aforementioned two
metrics and a tool called “Normalised Cross-Likelihood Ratio (NCLR)” are
the key to identify noise patterns.

2. A mechanism to use noise patterns to predict noise-free opportu-
nities for transmission. One of the goals of this thesis is to predict
transmission opportunities for low-power wireless devices. In this thesis,
a noise model is used to estimate noise patterns and a white-space model
is used to predict noise-free opportunities for nodes. These models are
parametrised based on the noise conditions in the deployed environment.
The parametrisation of the models is done with the help of knowledge about
the noise patterns. The models are trained to have different parameter sets
that are suitable for varying noise conditions and the most suited parameter
set is chosen based on the performance of the wireless network.

3. A protocol that utilises the prediction mechanism to identify ren-
dezvous points for low-interference communication. The backbone of
this thesis is the model-based receiver-aware MAC protocol (LUCID). The
receiver-initiated communication concept for wireless ad hoc networks came
with the work presented in MACA By Invitation (MACA-BI) [155] in 1997,
wherein the receiver polls one of its neighbours asking if it has a data packet
to send. In 2008, [154] proposed a receiver-initiated asynchronous duty-cycle
MAC protocol for finding the rendezvous point between the sender and the receiver. However, these methods cannot guarantee collision-free transmissions in the network with hidden terminals. In this work, the collected noise traces are exploited to build noise/white-space models that estimates noise patterns and predicts transmission opportunities, also known as white-spaces, for low-power wireless nodes. The predicted white-spaces from the white-space model is the key to finding the rendezvous points between a transmitter and a receiver. To this end, the receiver’s white-space model is used by the transmitter to decide the time slot during which it transmits packets. As the transmitter transmits packets in a round-robin fashion in the dedicated sub-slot within the selected time slot, the technique can minimise the collision of packets with other nodes that transmit in the same time slot. Therefore, LUCID can effectively tackle the hidden terminal problem as well.

LUCID uses a hybrid time slotted radio medium access mechanism for scheduling transmissions. This technique uses good properties of both centralised and decentralised scheduling paradigms, wherein tight time synchronisation is coordinated by a central entity, the network coordinator, while the actual transmission scheduling is done locally in the nodes. As it uses time slots, time synchronisation is a vital part of the proposed interference mitigation solution. This is achieved through updating clock drifts of individual nodes in the low-power wireless network upon receiving periodic time synchronisation packets flooded by the network coordinator. The transmission timing of the individual node is decided by the node itself locally with the use of its receiver’s white-space model, which predicts white-spaces at the receiver. The transmission opportunities predicted by the receiver’s white-space model are the key to deciding the transmission timing w.r.t. time slots of individual nodes.

In the presence of noise, low-power IEEE 802.15.4 nodes need to adapt to changing noise patterns and adjust their transmission schedules in order to avoid interfering transmissions and maximise the performance of their wireless network. As a model-based solution is proposed to tackle interference, the model parameters need to be tuned in line with the dynamics of noise to minimise transmission corruptions and losses, which eventually improves the energy efficiency of the wireless network. Therefore, to have packet loss awareness and adaptation to packet loss dynamics, continuous monitoring of data communication reliability is done to identify change points. Detection of
such change points triggers the re-parametrisation of the white-space model to align with the new noise patterns.

It is noteworthy that LUCID is designed and evaluated for periodic data applications. It is not realisable with currently available off-the-shelf low-power IEEE 802.15.4 nodes due to the CPU and memory requirements of the protocol for a) analysing the noise patterns, and b) predicting future noise-free windows. Nonetheless, LUCID can be implemented with modern processors, such as ARM Cortex CPUs, which have enough memory and CPU power.

4. **A large set of real-world indoor noise traces.** To properly parametrise the noise model that can estimate noise patterns, it is essential to accurately understand the radio environment. This is the foundation of the solution proposed in this thesis to mitigate the effects of noise in IEEE 802.15.4 networks. To this end, a large set of real-world, indoor noise traces are collected by considering the noise source’s as well as the sensor node’s perspectives.

- noise source’s perspectives: as the dominant indoor interferer [124], Wi-Fi creates heavy noise on IEEE 802.15.4 channels with much higher transmission powers than low-power sensor nodes. The situation is exacerbated by the overlapping nature of wide Wi-Fi channels, which generates aggregated disturbances on sensor nodes. In this work, the aggregated Wi-Fi disturbance is measured with the help of the noise traces collected by Wi-Fi dongles in two distinct indoor environments, OFFICE, an office building, and HOME, a shared apartment in a student dormitory. These locations of the experiments are chosen to cover different noise conditions. The outcome of the measurements is a set of timestamps at which Wi-Fi packets are detected.

- sensor node’s perspectives: to gain more understanding about the noise patterns as perceived by sensor nodes, the investigations are moved toward sensor node’s perspectives in which noise traces from unknown sources are collected by TMote Sky [139] sensor nodes. The acquired datasets consist of noise traces from three measurement campaigns in the same two indoor environments mentioned before. Unlike Wi-Fi measurements, these traces do not differentiate the source of noise and constitute noise from unknown sources as seen by sensor nodes. During
the measurement campaigns, timestamps of the start of noise detected by the sensor nodes are collected. The design of the data traces collection is informed by the interest in understanding the noise and its short- and long-term, channel, and location variations. Although the sensor node platform that is used can be deemed obsolete, the measurements are valid for all contemporary radios as the Clear Channel Assessment (CCA) threshold is set to $-77$ dBm in the sensor node.

The datasets expose several insights that may be of value to the broader community. The acquired Wi-Fi traces are characterised by Wi-Fi frame Inter-Arrival Time (IAT), while the characterisation of noise traces from unknown sources is done based on two metrics: mean (TBNS) and the noise signal count within a selected time period. The two metrics help to characterise unknown noise in two dimensions, which is more accurate than characterising it with a single metric. The noise characterisation revealed the distinct properties of unknown noise on different IEEE 802.15.4 channels, locations, and environments. Moreover, the collected datasets can be used to reproduce realistic indoor noise patterns in a precise and repeatable way in simulators and/or testbeds. Existing tools like JamLab [23] can use probability distributions extracted from the collected datasets to generate indoor noise patterns. Furthermore, the datasets can also directly inform CTI mitigation strategies.

## 1.5 Publications

All the contributions of this thesis are published in peer-reviewed publications as enlisted below.


1.6 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 reviews existing work exploited for noise detection and classification, noise modelling and low-power reliable data collection for IEEE 802.15.4 networks. In chapter 3, the procedure followed in this work for collecting real-world noise from unknown sources, the characteristics of the collected noise traces and their patterns are illustrated. Chapter 4 discusses the noise/white-space models that are used in this thesis, how the models are trained with the collected traces, and the performance of the models in estimating noise and predicting white-spaces. The proposed solution for mitigating the effects of radio frequency noise in IEEE 802.15.4 networks, LUCID, is presented in Chapter 5 wherein the prerequisites of the solution, how it works, how the MAC protocol has been integrated into IEEE 802.15.4 devices, and the performance of LUCID is discussed in detail before comparing it with the state-of-the-art. Finally, Chapter 6 concludes the thesis by summarising the performance of LUCID, discussing its limitations, and providing an outlook on future research directions.
Chapter 2

Background and State of the Art

The main focus of this chapter is to review the background of noise detection, identification of the source of noise, accurate noise estimation, feasibility of cognitive radio based techniques to tackle interference in low-power wireless networks, and interference mitigation techniques and systems, tailored especially for low-power wireless networks. Section 2.1 summarises noise detection and classification mechanisms. In Section 2.2, attempts for noise modelling are discussed. Finally, Section 2.4 presents complete systems that propose to increase the performance of low-power wireless networks.

2.1 Noise Detection and Classification

Several works aim to measure noise and understand its impact on low-power wireless networks and classify noise sources [124, 129, 79, 68, 71, 69, 97, 63]. Musaloiu and Terzis [124] use RSSI based features to quantify the noise on all IEEE 802.15.4 channels to select the least affected channel. Noda et al. [129] compute the ratio of channel idle to busy time for assessing channel quality in the presence of noise. SpeckSense [79] classifies RSSI bursts to characterise the channel as periodic, bursty, or a combination of both. SoNIC [68] uses information from corrupted packets along with RSSI for noise source classification. These works succeed in detecting noise and identifying its source. However, it is not clear how the techniques are useful for real-time interference mitigation due to the variety of noise sources, such as IEEE 802.11/Wi-Fi, Bluetooth, and radio emissions from microwave ovens. TIIM [71] makes a step further and extracts features from corrupted packets to quantify the noise conditions instead of identifying the source of the noise. By monitoring channel metrics, such as link quality
indication of corrupted frames, packet losses, CSMA deferrals, and properties of corrupted packets, TIIM defines a metric to estimate the impact of CTI. Thus, the noise condition can be mapped to a specific mitigation technique. Nonetheless, TIIM only recommends countermeasures that can be applied to prevailing noise conditions but does not provide their implementations. CrossZig [69], the follow-up work to TIIM, contains an implementation of an adaptive packet recovery and forward error correction coding technique to address the problem. According to real-time noise level assessment based on a measurement of packet reception ratio, ART [97] proposes a probabilistic mechanism for adaptively enabling CSMA only under severe noise conditions. It facilitates fine-tuning the throughput and packet reception ratio under a time-varying noise, that otherwise cannot be totally avoided by their multi-channel access mechanism. Grimaldi et al. [63] use manifold supervised-learning classifiers for real-time identification of multiple sources of noise, such as Wi-Fi, IEEE 802.15.4, Bluetooth, and Bluetooth Low Energy (BLE), by extracting envelope and spectral features of the underlying noise signals. Their technique can identify statistics of concurrent noise signals in adverse conditions.

All these solutions, however, do not aim to predict the white-spaces through modelling, that is instead the intention of this doctoral thesis.

2.2 Modelling of Radio Frequency Noise

Creating lightweight models of noise is not a trivial task. Several researchers have proposed models for channel occupancy [151, 60, 58, 86, 24, 25, 74] and for emulating noise caused by Wi-Fi and Bluetooth [23]. A two-state semi-Markov model for channel occupancy is defined in [151], and exploited by each node to identify the least affected channel and to switch accordingly. In comparison, the proposed technique in this thesis does not limit itself to noise caused only by Wi-Fi but identifies the white-spaces for a specific channel through modelling noise in the time domain. For modelling Wi-Fi noise, Glaropoulos et al. [60] proposes a semi-Markov model and Geirhofer et al. [58] study their continuous-time Markov chain model, while Laganà et al. [86] enhance this model with the ability to distinguish detected and undetected Wi-Fi activities. This model considers the limited communication range of sensor nodes and uses likelihood maximisation and neural networks for estimating the model’s parameters. Boano et al. [24, 25] define a two-state semi-Markov model for channel occupancy and noise measurements are used to measure the duration of the FREE and BUSY instants and compute
their CDFs. Based on the longest busy period, MAC protocols’ parameters are derived to meet the application requirements. JamLab [23] models and regenerates Wi-Fi/Bluetooth/microwave noise patterns from the sensor node’s perspectives, considering both saturated (always busy) and unsaturated traffic scenarios. A Markov chain model is used for saturated traffic and a probability mass function of empirical data for the non-saturated one. In contrast, the goal in this thesis is not to emulate noise but to predict it, and for this, a Gaussian Mixture Model (GMM) was used to capture the ambient noise conditions.

Beyond Co-existence [74] is closely related to that of this thesis, focusing on a model-based prediction of the length of the immediate white-space when a ZigBee frame is ready to be transmitted in the presence of Wi-Fi noise. Depending on the length of the white-space, the MAC frame is split in order to minimise collision probability. Nevertheless, continuous sampling of the operating channel is required as the model’s parameters are calibrated whenever there is a frame to be transmitted. Moreover, their prediction is short-term in contrast to the work in this thesis which is long-term and provides more information about when to transmit.

2.3 Cognitive Radio Communication Solutions

Cognitive Radio (CR) is a technology envisaged to solve problems in wireless networks emerging due to scarce frequency spectrum and its inefficient allocation/usage [6]. The CR-enabled devices are able to change their transmitter parameters based on interaction with the environment in which they operate [52]. Several works have proposed to enhance the communication in wireless networks using CR [35, 152, 84, 20, 73, 33, 72].

C-MAC [35] exploits a superframe based distributed multi-channel MAC protocol to tackle the dynamics of resource availability due to primary user activities. Here, the coordination is accomplished with a dynamically assigned Common Control Channel (CCC). Su and Zhang [152] use two transceivers, one for conveying control data over the dedicated CCC while the other for data communication. The authors use different sensing policies for finding available idle channels and a time slotted mechanism for coordination between nodes. The node that detects an idle channel informs the other nodes via the CCC with the use of beacons in mini-slots. Two transceivers in a resource-constraint node, such as IEEE 802.15.4 based sensor nodes, would be not energy optimal.
The use of a CCC leads to problems such as single point of failure and channel saturation with an increasing number of users. SYN-MAC [84] avoids the CCC and uses a hybrid MAC protocol wherein the exchange of control signals is done in a time slotted fashion while data transmission is based on random access. SYN-MAC shows better connectivity and higher throughput than CCC based protocols in a congested network. Similar to SYN-MAC, the protocol proposed in this thesis uses time slots not only for signalling but also for data transmissions. COMAC [20] uses a contention-based handshaking mechanism for the exchange of control information. The protocol continuously updates locally available channels through the overheard of control packets. When a pair of nodes wants to initiate a communication, information regarding locally available channels is shared between them, and the receiver selects the channels to communicate based on the Signal to Interference Noise Ratio (SINR) of the channels. In contrast to COMAC, to assess the noise conditions, the proposed solution uses PDR based feedback mechanism.

SCA-MAC [73] exploits the statistics of spectrum usage for decision making on channel access. To this end, for each channel, a list of the last 1000 channel idle durations is maintained. CR-CSMA/CA [33] is another multichannel MAC protocol that extends the traditional RTS/CTS to a three-way-handshake mechanism PTS (Prepare To Send)/RTS/CTS for channel access coordination. In this thesis, channel access coordination is accomplished via a time slotted receiver-aware model-based white-space prediction mechanism.

CR-RDV [72] is a CSMA/CA-based distributed CR rendezvous MAC protocol to overcome channel contention and rendezvous problem in wireless networks, which occurs when multiple devices achieve rendezvous on the same channel. The data channel is selected based on the receiver preferences. To this end, the protocol maintains a list of backup channels to be used during service interruptions and the channel list is integrated into RTS/CTS packets. Rendezvous is achieved in this thesis with the proposed white-space prediction mechanism and receiver-aware communication.

Most of these CR solutions require multichannel spectrum sensing and the use of a global or a local CCC for the exchange of control information. To satisfy those requirements, wireless devices need to spend a considerable amount of energy. Thus, these solution needs more research before adopting them in low-power wireless networks. Nonetheless, similar to CCC in CR, a dedicated time slot for signalling is used in the proposed communication protocol in this thesis.
2.4 Low-power Reliable Data Collection

Wireless sensor networks are tasked with monitoring and recording physical conditions, such as temperature and humidity, in the environments wherein they have been deployed and organising the data collection at a central entity. Once deployed, such networks enable automatic data acquisition of physical phenomena in the operating environments. Generally, the lifetime of such wireless networks is envisioned to exceed several years to minimise human intervention [110, 90]. However, under adverse interference conditions, wireless transmissions of such networks might get corrupted leading to increased packet losses. This amplifies the energy consumption of nodes in the network due to re-transmissions, which eventually reduces the network lifetime. Therefore, to achieve this longevity and high reliability of data collection, it is crucial to maximise the performance of the wireless network.

Several protocols have been proposed to achieve high performance in low-power wireless networks. Solutions such as Splash [43] and Pando [44] exploit constructive interference and channel diversity to obtain high reliability. Because the combination of constructive interference and pipelining jeopardises the reliability of data communication, they further apply transmitter density diversity, opportunistic overhearing, and forward error correction to increase the reliability. They focus more on communication reliability than the energy consumption of low-power wireless networks. ETSCH [157] is another protocol that exploits channel diversity to obtain high reliability. It is a centralised TDMA-based protocol that improves the TSCH protocol [2] with a distributed channel quality estimation technique, which dynamically identifies less busy channels. The communication reliability is boosted by adaptively selecting a subset of low-noise channels for hopping between. Even though ETSCH can achieve high communication reliability, the energy overhead incurred due to the distributed channel quality estimation is significant.

Complete data collection systems to improve the performance of low-power wireless networks have been emerging over the last decade. These solutions include Dozer [30, 29], RedFixHop [83, 49, 48, 47], Sommer et al. [150], Sparkle [169, 170], Lim et al. [100], Al Nahas et al. [8, 7], BigBangBus [50], Ma et al. [109, 108], RedNodeBus [51], ContikiMAC [45], and CRYSTAL Clear [162, 78, 77].

Dozer [30, 29] is a network stack designed for periodic data-gathering with low energy consumption. Idle listening and overhearing of sensor nodes are minimised with the proper coordination of MAC, topology control, and routing. Using a tree-based network structure and a wake-up schedule that relies only on local
synchronisation, the parents schedule transmissions for their children. Dozer focuses much on improving the longevity of low-power wireless networks but pays less attention to communication reliability. The proposed approach in Dozer was demonstrated in a data-gathering application, 120 seconds data sampling interval, using a network of 40 sensor nodes. The mean duty-cycle per node per data period was $3.5E-4\%$ while tolerating a maximum of 98.85% packet delivery ratio, excluding a node that suffered approximately 30% packet losses. In comparison, the proposed solution in this thesis with a 60 seconds data sampling interval achieves $2.4E-4\%$ mean duty-cycle and 92.73% mean packet delivery ratio per node per data period.

RedFixHop [83, 49] exploits flooding and constructive interference for hardware-triggered simultaneous transmissions. The hardware triggered communication mechanism is the key to minimising the delay incurred at the relay nodes, which should be in the sub-microsecond region for achieving constructive interference. Even though they achieve high communication reliability, the re-transmissions triggered by hardware acknowledgements increase the energy consumption of the wireless network. Another downside is that the efficacy of RedFixHop is limited by the transceiver. The CC2420 transceiver requires an additional delay of 12 symbol periods after the last symbol of the incoming frame, which implicitly restricts the payload length to only 1 Byte when using hardware-triggered ACKs to achieve constructive interference (Figure 24 in the CC2420 datasheet [137]). Note that the clock drifts of the nodes should be below 0.5 $\mu$s to achieve constructive interference. Their follow-up work [48, 47] exploits frequency diversity as well, which assures high reliability in adverse conditions, such as frequency-dependent noise or jamming.

Similar to RedFixHop, Sommer et al. [150] applies flooding and frequency diversity for improving the communication reliability of low-power wireless networks. The frequency hopping sequence is randomly generated based on the packet counter and the sequence number. The energy consumption of this technique can be tailored to the available energy budget by adapting the flooding interval and the number of packet repetitions.

Sparkle [169, 170] combines flooding, topology control, and transmission power control in low-power wireless networks for improving transmission reliability and for decreasing energy consumption and latency. This is achieved by finding the most reliable paths between the source and the destination with the help of the capture effect. Authors have shown that energy saving is significant at a slight improvement in reliability.
Transmission schemes wherein constructive interference and the capture effect are exploited for robust communication protocol face an enormous problem when temporal misalignments occur in the concurrent transmissions. Lim et al. [100] propose a mechanism to mitigate the impact coming from such temporal misalignment by using consecutive synchronous transmissions, randomising transmit powers to artificially increase the diversity in signal strengths, and time synchronisation based on linear regression.

Al Nahas et al. [8, 7] propose a distributed data aggregation protocol that is based on synchronous transmissions and the capture effect, wherein upon receiving packets, nodes merge received data with their own and transmit again synchronously. The performance is further enhanced by channel hopping, utilising multiple channels in parallel, and channel blacklisting.

BigBangBus [50] is a wireless bus that exploits flooding, the capture effect, and channel hopping for low-latency reliable packet delivery. The authors do not use any operating system, such as Contiki, in their solution, instead, they access the microcontroller and the radio registers directly. Therefore, the low-level optimisation and high-priority flows are the keys to decreasing the delays incurred in the software-triggered transmissions, while the transmission power randomisation and slot skipping can reduce energy consumption. Hopping over multiple channels in a pseudo-random sequence helps to skip the interference and to have high transmission reliability. As opposed to RedFixHop [83, 49], BigBangBus proposes software-triggered transmissions, which are meant to increase the payload size. However, this only manages to increase payload up to 2 bytes compared to 1 byte in RedFixHop.

Ma et al. [109, 108, 107] propose a concurrent transmission-based protocol that is based on constructive interference and the capture effect. Channel hopping is used to further increase the reliability, while the ranking of nodes and the oriented flooding mechanism save energy by allowing nodes outside the communication path to sleep.

RedNodeBus [51] proposes a real-time wireless bus, which is based on RedFixHop [83, 49] and BigBangBus [50]. RedNodeBus overcomes the restrictions of very short packets in both methods, and exploits frequency-, spatial-, and time-diversities using redundant transmissions. Long preambles used in RedNodeBus eases tight synchronisation requirements and are used to boost the likelihood of the capture effect. Nonetheless, long preambles increase the transmission time and consequently, the energy consumption in every transmission. The synchro-
nisation beacons further increase energy consumption, especially for high data rate applications. However, the scheme manages to reduce a part of the energy consumption by introducing a predetermined time-to-live field, after which the packets are discarded to avoid repetitions.

ContikiMAC [45] proposes an asynchronous wake-up mechanism wherein the sender continuously transmits packets during the wake-up interval while the receiver periodically wakes up and listen for packet transmissions. If a transmission is detected during a wake-up, the receiver is kept on to be able to receive the packet. Upon successfully receiving a unicast transmission, the receiver sends a link layer acknowledgement to the transmitter as depicted in Figure 2.1a. With the acknowledgement received, the transmitter learns the wake-up phase of the receiver, which results in fewer transmissions during subsequent wake-up intervals. The broadcast packets do not result in acknowledgements, as shown in Figure 2.1b, thus the transmitter repeatedly sends packets during the full wake-up interval.
to maximise the packet reception by all its neighbours. ContikiMAC also has a mechanism to detect noise and allow the potential receiver to go back to sleep fast, saving energy. To increase the transmission reliability, re-transmissions are used at the cost of energy efficiency.

![Diagram of CRYSTAL operation](image)

(a) Initiation, transmission, and termination.

![Diagram of CRYSTAL channel hopping](image)

(b) Channel hopping.

Fig. 2.2 Operation of CRYSTAL.

CRYSTAL Clear [162, 78, 77] combines synchronous transmissions with channel hopping and noise detection to tackle strong noise in the operating channel and improve the performance of low-power wireless networks. As depicted in Figure 2.2a, CRYSTAL uses a synchronisation (S) phase at the beginning of each epoch to ensure time synchronisation. Exploiting capture effect and constructive interference, the senders compete for the subsequent transmission (T) slot with concurrent transmissions, and the sink disseminates the information about the sender whose packet was received in the following acknowledgement (A) slot. These TA pairs are repeated until all the senders have successfully transmitted their packets, after which the termination criteria (R consecutive silent pairs) is applied. A silent pair consists of a silent T followed by a NACK sent by the sink. When there is strong RF noise on the channel, the termination criteria may be triggered prematurely
forcing the network to go to sleep, highly affecting the transmission reliability. Noise detection is added to tackle the early termination due to noise wherein the channel is marked as noisy when CCA reads above $-60$ dBm at least 80 times. When this happens the nodes are kept on for a longer period irrespective of the triggered premature termination criteria to schedule extra transmissions in a decentralised way for fighting interference. The CCA threshold used for noise detection can only detect very high noise, such as emissions from microwave ovens, and the technique may keep nodes unnecessarily active, which deteriorates the performance of the low-power wireless network. The performance of the authors early work of CRYSTAL [77] evaluated under emulated Wi-Fi noise generated by JamLab [23] demonstrates its high performance. Nonetheless, in the presence of noise from a microwave oven, the reliability of CRYSTAL falls below 80%. Channel hopping introduced in the subsequent work of CRYSTAL [78, 162] tackles harsh interference by moving to a less noisy channel to escaping persistent noise. As shown in Figure 2.2b, the channel hopping is driven by the S slot which defines the hopping sequence in the epoch and the channels with 7-channel spacing are used in the hopping sequence to escape both Wi-Fi noise and emissions from microwave ovens.

With the design and implementation of Glossy [53], most communication protocols/systems tend to exploit synchronous transmissions, constructive interference, and the capture effect as a way of improving the reliability of low-power wireless networks. Moreover, channel hopping, channel blacklisting, multiple parallel channels, and slot skipping are applied to avoid harsh interference/jamming. Node ranking and oriented flooding, random transmit powers and predetermined time-to-live were used as a measure of minimising energy consumption. In contrast, LUCID as presented in Chapter 5 of this doctoral thesis proposes a model-based receiver-aware data communication concept to improve the performance of low-power wireless networks. Thus, the joint use of all the work done in this thesis will pave the way toward new research directions in the wireless research community in general.

2.5 Taxonomy of CTI Solutions

Although there are many other solutions designed to address CTI in low-power wireless networks, this chapter provides only a brief overview of such attempts made. All these techniques can be categorised into four groups according to their novelty as depicted in Figure 2.3: spectrum sensing, noise source classification,
noise modelling, and interference mitigation. The main four categories are further sub-divided based on the approach they followed. Spectrum sensing can be achieved through energy detection, i.e., RSSI measurements, or any other technique. The second branch of Figure 2.3 focuses on the metrics used for classifying noise sources, while the third branch divides the techniques according to their ability to model noise (here, the techniques that did not focus on a particular noise source fall under \textit{not specific} sub-category). The last branch groups CTI solutions based on their interference mitigation solution. Finally, Table 2.1 lists CTI solutions surveyed in this section and indicates their classification following the taxonomy in Figure 2.3.

![Fig. 2.3 Taxonomy of CTI solutions.](image-url)
Table 2.1 Classified solutions according to the taxonomy in Figure 2.3.

<table>
<thead>
<tr>
<th>Protocol/Technique</th>
<th>Spectrum Sensing</th>
<th>Noise Source Classification</th>
<th>Noise Modelling</th>
<th>Interference Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musaloiu and Terzis [124]</td>
<td>energy detection</td>
<td>none</td>
<td>none</td>
<td>frequency diversity</td>
</tr>
<tr>
<td>ETSCH [157]</td>
<td>energy detection</td>
<td>none</td>
<td>none</td>
<td>frequency diversity</td>
</tr>
<tr>
<td>TSCH [2]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity</td>
</tr>
<tr>
<td>Stabellini et al. [151]</td>
<td>energy detection</td>
<td>none</td>
<td>Wi-Fi</td>
<td>frequency diversity</td>
</tr>
<tr>
<td>ART [97]</td>
<td>other technique</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, other technique</td>
</tr>
<tr>
<td>RedFixHop [83, 49, 48, 47]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, synchronous transmissions, frequency diversity,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sommer et al. [150]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>synchronous transmissions, frequency diversity,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparkle [169, 170]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>synchronous transmissions, other technique,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lim et al. [100]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>synchronous transmissions, other technique,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Al Nahas et al. [8, 7]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>synchronous transmissions, other technique,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BigBangBus [50]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>synchronous transmissions, other technique</td>
</tr>
</tbody>
</table>
Table 2.1 continued from previous page

<table>
<thead>
<tr>
<th>Protocol/Technique</th>
<th>Spectrum Sensing</th>
<th>Noise Source Classification</th>
<th>Noise Modelling</th>
<th>Interference Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ma et al. [109, 108]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, synchronous transmissions, other technique</td>
</tr>
<tr>
<td>RedNodeBus [51]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, synchronous transmissions, other technique</td>
</tr>
<tr>
<td>CRYSAL Clear [162, 78, 77]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, synchronous transmissions, other technique</td>
</tr>
<tr>
<td>Splash [43]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, synchronous transmissions, other technique</td>
</tr>
<tr>
<td>Pando [44]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>frequency diversity, synchronous transmissions, other technique</td>
</tr>
<tr>
<td>TIIM [71]</td>
<td>energy detection, other techniques</td>
<td>RSS-based metrics, properties of corrupted packets, other metrics</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>SoNIC [68]</td>
<td>energy detection</td>
<td>RSS-based metrics</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>SpeckSense [79]</td>
<td>energy detection</td>
<td></td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Grimaldi et al. [63]</td>
<td>energy detection</td>
<td></td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Noda et al. [129]</td>
<td>energy detection</td>
<td></td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>
### Table 2.1 continued from previous page

<table>
<thead>
<tr>
<th>Protocol/Technique</th>
<th>Spectrum Sensing</th>
<th>Noise Source Classification</th>
<th>Noise Modelling</th>
<th>Interference Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JamLab [23]</td>
<td>energy detection</td>
<td>RSS-based metrics</td>
<td>not specific</td>
<td>none</td>
</tr>
<tr>
<td>Laganà et al. [86]</td>
<td>energy detection</td>
<td>none</td>
<td>Wi-Fi</td>
<td>none</td>
</tr>
<tr>
<td>Glaropoulos et al. [60]</td>
<td>other technique</td>
<td>none</td>
<td>Wi-Fi</td>
<td>none</td>
</tr>
<tr>
<td>Geirhofer et al. [58]</td>
<td>energy detection, other techniques</td>
<td>RSS-based metrics, properties of corrupted packets, other metrics</td>
<td>Wi-Fi</td>
<td>none</td>
</tr>
<tr>
<td>CrossZig [69]</td>
<td>energy detection, other techniques</td>
<td>none</td>
<td>none</td>
<td>other technique</td>
</tr>
<tr>
<td>Dozer [30, 29]</td>
<td>other technique</td>
<td>none</td>
<td>none</td>
<td>other technique</td>
</tr>
<tr>
<td>ContikiMAC [45]</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>other technique</td>
</tr>
<tr>
<td>Boano et al. [24, 25]</td>
<td>none</td>
<td>none</td>
<td>not specific</td>
<td>other technique</td>
</tr>
<tr>
<td>Beyond Coexistence [74]</td>
<td>energy detection</td>
<td>none</td>
<td>not specific</td>
<td>predict transmission opportunities</td>
</tr>
</tbody>
</table>
Chapter 3

Collecting and Understanding Radio Frequency Noise

There is a plethora of wireless communication devices based on a range of standards as well as proprietary technologies operating in the 2.4 GHz ISM band, such as low-power wireless networks, IEEE 802.11/Wi-Fi, Bluetooth, microwave ovens, cordless phones, RFIDs, and surveillance cameras, whose radio emissions interfere with each other due to their broadcast nature, widely known as Cross-Technology Interference (CTI). Among these, communication by low-power wireless network devices, especially those based on the IEEE 802.15.4 standard, are significantly affected by IEEE 802.11/Wi-Fi devices as they dominate the 2.4 GHz ISM band with much higher transmission powers (20 dBm) compared to low-power wireless devices (0 dBm). As the objectives in this thesis are to improve the performance of low-power wireless networks and to efficiently utilise the frequency spectrum while adapting to varying noise conditions, at the beginning of this research, Wi-Fi signal trace collection with Wi-Fi dongles, their characterisation, and estimation was the prime focus. Even though Wi-Fi traces paved the path to understanding the behaviour of the dominant Wi-Fi noise, there was no conclusive evidence from the Wi-Fi noise characterisation as to how a sensor node perceives noise. Thus, to gain a better understanding of how a sensor node might get affected by noise, the focus of the thesis was moved from investigating the noise sources to the sensor nodes’ perspectives. This facilitated the collection and characterisation of noise from unknown sources, that is received by sensor nodes rather than differentiating noise sources.

This chapter describes the contributions of this thesis toward the collection of a large set of noise traces [41] and their characterisation. Section 3.1 presents
how the Wi-Fi noise measurements were carried out. The required background information to understand noise characteristics and the procedure for characterising Wi-Fi noise is presented in Section 3.2. Section 3.3 describes how the noise from unknown sources was collected. In Section 3.4, the metrics and methods used for characterising the collected noise traces from unknown sources are illustrated. Identified noise patterns are presented in Section 3.4.3. Finally, Section 3.5 summarises the observations from the noise measurement campaigns and highlights the importance of noise awareness toward accurate noise estimation and interference mitigation.

## 3.1 IEEE 802.11 Noise

As IEEE 802.11 is a higher bandwidth technology than IEEE 802.15.4, e.g., bandwidth ranges between 20 MHz and 40 MHz depending on the specific PHY layer specification, it creates noise on multiple 2 MHz IEEE 802.15.4 channels simultaneously (see Figure 3.1). This effect is called aggregated Wi-Fi signals [92], which unfortunately diminishes the transmission opportunities for IEEE 802.15.4-based sensor networks and degrades their performance in terms of reliability and lifetime. Furthermore, the higher data rate of IEEE 802.11 compared to IEEE 802.15.4, e.g., 1 Mbps to 300 Mbps (depending on PHY layer) versus 250 kbps, creates rapid variations in time and frequency domain, which are hard to detect by the typical IEEE 802.15.4 sensor node receiver hardware.

### 3.1.1 Measurement Locations

The study areas of this thesis are two typical indoor environments: HOME and OFFICE. The former is an office building and the latter is a student dormitory.
The student dormitory consists of multiple three-story buildings dispersed in a large geographical area; however, only one of the buildings wherein the author had been living was considered as the home environment. It is also noteworthy that the dormitory management had configured Wi-Fi routers in such a way that individual buildings had a different Wi-Fi channel assignment.

These two indoor environments exhibit unique radio frequency noise conditions, such as burstiness and self-similarity, due to varying user behaviours in those environments, paving the way to validate the proposed approach under diverse noise conditions. The office is characterised by bursty Wi-Fi noise (as shown in Figure 3.3); however, it does not exhibit self-similarity as much as in the home. This is confirmed by the Hurst parameter [138, 75], showing a value of 0.5 for the office and 0.7 for the home. Note that the definition of self-similarity and how to quantify it with the Hurst parameter is presented in Section 3.2.1. It is noteworthy that even a short trace could sufficiently capture the statistical properties of noise in any given environment if the noise exhibits high self-similarity [93, 74].

### 3.1.2 Hardware and Software Platforms

To investigate the characteristics of Wi-Fi radio emission patterns, a simple test-bed comprised of IEEE 802.11n compliant USB Wi-Fi dongles was used. The dongles were connected to a line-powered USB hub which was mounted on a wall at 1.75 m height. The USB hub is then connected to a USB port of a PC, which controls the experiment and stores Wi-Fi traces. The test-bed is depicted in Figure 3.2.

A Wi-Fi device receives Wi-Fi frames on its operating channel, and each frame has a time of arrival. The time between consecutive Wi-Fi frame arrivals is known as Wi-Fi frame Inter-Arrival Time (IAT), whose distribution is a key statistical property that can be used to identify Wi-Fi radio emission patterns and essentially to accurately estimate them. Thus, it is necessary to acquire timestamped Wi-Fi signals. Therefore, by using the tcpdump tool [158], Wi-Fi signals detected by the USB dongles were dumped into pcap files, from which timestamps can be extracted. Then, the Wi-Fi aggregated signal traces were obtained by merging traces of overlapping Wi-Fi channels, i.e., traces from overlapping channels were appended and sorted in ascending order based on their timestamps. To this end, mergecap and then tshark are used to extract the IAT distribution; both tools are part of the popular network protocol analyser, Wireshark [166].
3.1 IEEE 802.11 Noise

Fig. 3.2 The test-bed for characterising Wi-Fi noise.

3.1.3 Wi-Fi Measurement Execution

In the office environment, there are plenty of Wi-Fi installations, thus IEEE 802.11 transmissions appear on almost all the IEEE 802.11 channels in the 2.4 GHz ISM band. In this research, traces from highly busy four Wi-Fi channels, i.e., 1–4, were simultaneously collected, ensuring each IEEE 802.15.4 overlapped channel, i.e., 11–14, is affected by a different number of Wi-Fi channels, as can be seen in Figure 3.1.

Unlike the office, the home had a specific Wi-Fi channel assignment, as mentioned in Section 3.1.1. With the available Wi-Fi router settings, only IEEE 802.11 channels 7 to 13 showcased a significant amount of IEEE 802.11 transmissions
in comparison with other channels in the HOME. This was a compelling scenario to use IEEE 802.11 channels 7 to 13 in the HOME, as opposed to IEEE 802.11 channels 1 to 4 in the OFFICE, in the noise measurements. As the goal of this setup was to investigate and characterise highly affected channels by noise, it resorted to sniffing IEEE 802.15.4 channels 7 to 13, which overlap with IEEE 802.15.4 channels 20 to 23, each being affected by four IEEE 802.11 channels. Moreover, to induce diversity in noise on measured channels and to mimic a realistic situation in the HOME environment, a continuous trace of video-streaming (not a constant bit rate flow as there is buffering) was generated, using another laptop, on IEEE 802.15.4 channel 11 which only overlaps with IEEE 802.15.4 channels 21, 22, 23, and 24, as depicted in Figure 3.1.

The Wi-Fi noise traces were collected during a busy time on a day, for two hours, from 10:30 AM to 12:30 PM, in the OFFICE and the HOME. These two-hour busy periods in each environment can be considered as representative of the worst-case scenario in terms of noise from the low-power wireless networks’ perspectives. Moreover, the aforementioned Wi-Fi noise measurement settings are favourable to gather noise traces from IEEE 802.11 channels in different environments that exhibit channel noise pattern diversity.

3.2 Characterising IEEE 802.11 Noise

Characteristics of measured Wi-Fi noise defer based on the environment in which the noise measurements were collected. Figure 3.3 illustrates varying Wi-Fi noise characteristics in terms of its burstiness within a randomly selected 500 ms time period. Each vertical line in the figure represents a noise measurement and a noise burst can be recognised with a cluster of such vertical lines. With this notion, as Figure 3.3 clearly demonstrates, the OFFICE has more bursty Wi-Fi traffic than that of the HOME. Wi-Fi has varying frame types, such as management, control, and data, which are different in size. Because of their varying sizes, these frames induce interference on low-power transmissions differently. The longer the frame, the more interference that they can create on low-power wireless transmissions. Especially, the data frames have a longer size in comparison with other management and control frames. In the collected traces, the percentages of data packets are 18% and 1% approximately, respectively for the HOME and the OFFICE. Because there are more Wi-Fi access points deployed in the OFFICE than the HOME, the
3.2 Characterising IEEE 802.11 Noise

Fig. 3.3 Burstiness of Wi-Fi noise. Each vertical line represents a noise measurement on:  
a) IEEE 802.15.4 channel 21 (HOME),  
b) IEEE 802.15.4 channel 14 (OFFICE).

Table 3.1 Stationarity test.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Test statistic</th>
<th>Critical value (5%)</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOME</td>
<td>-20.037</td>
<td>-2.862</td>
<td>0</td>
</tr>
<tr>
<td>OFFICE</td>
<td>-54.050</td>
<td>-2.862</td>
<td>0</td>
</tr>
</tbody>
</table>

Wi-Fi traffic consists of a higher number of control packets in the former than that in the latter.

To test the collected noise traces for the stationarity, the following Null Hypothesis ($H_0$) and the Alternative Hypothesis ($H_A$) are used with the Augmented Dickey-Fuller (ADF) test [55]:

- $H_0$: “the noise trace is non-stationary”;
- $H_A$: “the noise trace is stationary”.

With 5% critical value and 0.05 p-value, traces from both the OFFICE and the HOME reject $H_0$ (i.e., p-value < 0.05 and test statistic < critical value) and take that the traces are stationary according to Table 3.1.

3.2.1 Preliminaries

In this section, statistical preliminaries that are required for a thorough characterisation of Wi-Fi noise and model parameters for the estimation of noise are
3.2 Characterising IEEE 802.11 Noise

presented. The reader who is familiar with generic statistical properties such as mean, standard deviation, and coefficient of variation of a process will only want to skim to the subsections describing self-similarity, Hurst parameter, and Markov-Modulated Poisson Process (MMPP). Note that complex mathematics behind the Hurst parameter and the MMPP model parameters are avoided; nonetheless, the reader is encouraged to refer to resources on these for further understanding.

**Self-similarity.** Several empirical studies of traffic measurements [93, 5, 64, 125, 99] have convincingly demonstrated that network traffic, such as traffic in broadband networks, is self-similar. Intuitively, self-similarity describes the phenomenon in which certain characteristics (i.e., structural patterns and statistical properties) of the traffic are preserved irrespective of scaling in space or time, which can actually be exploited to better infer traffic properties.

To clarify the terminology used in this thesis, the definition of a few basic concepts are briefly summarised here. Let \( X(t) = (X_t : t \geq 0) \) be a stationary process, i.e., its joint distribution across a collection of times \( t_1, \cdots, t_N \) is invariant to time shifting. \( X \) is called *self-similar* if:

\[
X(at) = a^H X(t), a > 0 \tag{3.1}
\]

where the equality refers to equality in distributions, \( a \) is a scaling factor, and the self-similarity parameter \( H \) is called the *Hurst* parameter [138, 75].

The proposed approach takes advantage of this property, depicted in Figure 3.4 for a set of collected traces, to infer noise characteristics without incurring excessive data trace collection. The Hurst parameter is used as a scale to quantify the time invariance of noise traces.

**Statistical properties of noise.** Key statistical metrics typically used to provide insights into network traffic are: *mean* (\( \mu \)), *standard deviation* (\( \sigma \)) and, *coefficient
of variation ($C$) computed as $\sigma/\mu$ [130, 60, 54]. Moreover, the predominant way to quantify self-similarity is through $H$, which is a scalar. $H$ takes on values from 0 to 1, wherein values close to 0, 0.5, and 1 indicating anti-persistent traffic, traffic similar to white noise, and highly self-similar traffic, respectively [132, 135].

Calculating the Hurst parameter is not that straightforward: firstly, it can only be estimated, secondly, although there are several methods to estimate it, such as absolute value, aggregated variance, boxed periodogram, differenced variance, Higuchi, Peng, periodogram, rescaled range, wavelet, and Whittle, they often produce conflicting results due to the bias of the estimator towards under-/over-estimating the Hurst parameter [135]. In this thesis, Peng [133], Periodogram [59], and Boxed-Periodogram [156] methods were used to estimate $H$, as they only have a very small bias and the estimation is reasonably well in comparison with other methods for a sample size as low as 7000 [135]. The selected three estimators could also produce conflicting estimations due to their differences in the bias. Therefore, to minimise the effect coming from the bias of the estimators, the median of the estimated values is considered as the Hurst parameter of the underlying process. Note that the median of the three estimated values is more stable than the mean of them even when one of the estimators fails.

Markov-Modulated Poisson Process. Markov-Modulated Poisson Process (MMPP) has been widely used in many areas, such as economics [14, 127, 65], natural science [159, 85], network theory [143], telecommunications [168], and data traffic modelling [67, 145, 32]. The main reason as to why the MMPP model is popular among the researcher in the aforementioned areas is that it provides a faithful representation of real-world circumstances [15]. Moreover, they can also capture long-range dependency of statistical processes, which is said to be caused by slowly decaying autocorrelation [54, 143, 168].

It is well known that IEEE 802.11 traffic showcases burst and non-burst frames. This phenomenon of IEEE 802.11 frames can be easily fitted into a two-state

![Fig. 3.5 MMPP(2) noise model.](image)
MMPP, known as the second-order MMPP, denoted by MMPP(2), wherein each state of the MMPP(2) models burst or non-burst IEEE 802.11 traffic while the modulation between the two states is controlled by a continuous-time Markov chain. This is depicted in Figure 3.5. The four defining parameters of the MMPP(2) model according to [54] are:

\[
Q = \begin{pmatrix} -r_1 & r_1 \\ r_2 & -r_2 \end{pmatrix}, \quad \Lambda = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad \pi = \frac{1}{r_1 + r_2} \begin{pmatrix} r_2 & r_1 \end{pmatrix}
\]

where \( Q \) represents the infinitesimal generator, \( \Lambda \) is the matrix of the Poisson arrival rates, and \( \pi \) is the initial probability vector of the underlying Markov process.

These parameters are obtained by fitting the empirical Wi-Fi frame IAT distribution to that of the model. How these parameters were computed in the proposed approach is described in Section 3.2.2.

### 3.2.2 Estimating Wi-Fi Noise

After the Wi-Fi noise measurements, the next goal is to characterise the collected Wi-Fi trace. To this extend, the Wi-Fi frame IATs were used, as they can be used as a measure of determining temporary vacant channel spaces, also known as white-spaces, that can be utilised by low-power wireless nodes. Note that small Wi-Fi frame IATs are an indication of heavy and bursty Wi-Fi signals, as they reduce the availability of transmission opportunities for low-power wireless networks. The IATs of Wi-Fi frames were extracted from the measured traces with the `tshark` tool. Then, the characterisation of Wi-Fi noise is captured through calculating mean (\( M_1 \)), coefficient of variation (\( C \)), and the Hurst parameter (\( H \)) of the Wi-Fi frame IAT.

By having a Hurst parameter greater than 0.5, the collected Wi-Fi noise traces demonstrated their self-similarity. This is further evident in Figure 3.4 wherein the statistical properties, such as the mean and the coefficient of variation of Wi-Fi frame IAT, are shown for two time-scales: 20 minutes vs. 30 seconds. Irrespective of the time-scale, these statistical properties exhibit similar values.

As the Wi-Fi traffic showcases self-similarity and due to wide usage for traffic modelling [54, 147, 148, 12, 89], one could use a Markov-Modulated Poisson Process (MMPP), for estimating Wi-Fi noise. As the end-goal of this study is to determine the channel status, FREE or BUSY, the MMPP model with two states, the MMPP(2)
3.2 Characterising IEEE 802.11 Noise

traffic model, is used, which: i) assumes traffic exhibits self-similarity and, ii) uses empirical data for estimating its parameters.

Estimating model parameters of an MMPP(2) directly from the empirically measured Wi-Fi frame IAT distribution is non-trivial. According to [54, 130], an MMPP(2) process can be approximated by a second-order hyperexponential distribution, with parameters $\mu_1$, $\mu_2$ and $p$, for fitting an empirical Wi-Fi frame IAT distribution to the model. As shown in Figure 3.6, $\mu_1$, $\mu_2$ are mean frame arrival rates and $p$ represents the probability at which traffic is generated at a mean rate of $\mu_1$. These parameters can be computed with the already computed mean time between arrival ($M_1$) and the coefficient of variation ($C$) of the Wi-Fi frame IAT distribution through the balanced means method [4], as illustrated in Equation 3.3.

\[ p = \frac{1}{2} \left( 1 + \sqrt{\frac{C^2 - 1}{C^2 + 1}} \right), \quad \mu_1 = \frac{2p}{M_1}, \quad \mu_2 = \frac{2(1-p)}{M_1} \quad (3.3) \]

Then, the parameters of the MMPP(2) model, $\lambda_1$, $\lambda_2$, $r_1$ and $r_2$ can be calculated from the estimated parameters $p$, $\mu_1$, $\mu_2$ and the Hurst parameter $H$ with Equation 3.4.

\[ \lambda_1 = \frac{1}{2} \left[ p(1-\beta)(\mu_1 - \mu_2) + \beta\mu_1 + \mu_2 + \sqrt{\xi} \right] \]
\[ \lambda_2 = \frac{\mu_1\mu_2[\lambda_1 - p(\mu_1 - \mu_2) - \mu_2]}{\lambda_1\mu_1 - \lambda_1p(\mu_1 - \mu_2) - \mu_1\mu_2} \]
\[ r_1 = \frac{(\mu_1 - \lambda_1)(\mu_2 - \lambda_1)}{\lambda_2 - \lambda_1} \]
\[ r_2 = \frac{(\lambda_2 - \mu_1)(\lambda_1 + r_1 - \mu_1)}{\mu_1 - \lambda_1} \quad (3.4) \]

Fig. 3.6 second-order hyperexponential traffic model.
where

\[
\xi = \left[ p(1 - \beta)(\mu_1 - \mu_2) + \beta \mu_1 + \mu_2 \right]^2 - 4\beta \mu_1 \mu_2,
\]

\[
\beta = 2 - 2H
\]

The approximation expressed by Eq. 3.4 can be applied if and only if Wi-Fi frame IAT distribution satisfies both conditions: \(0.5 < H < 1\) and \(C > 1\). In all other cases, when \(\frac{1}{\sqrt{2}} \leq C \leq 1\), a second-order Coxian distribution [4] must be used prior to the second-order hypereponential distribution. In this case, parameters \(p\), \(\mu_1\) and \(\mu_2\) are computed by:

\[
p = \frac{1}{2C^2}, \quad \mu_1 = \frac{2}{M_1} \left( \frac{p}{1 + p} \right), \quad \mu_2 = \frac{2}{M_1}
\]

(3.5)

### 3.2.3 Aggregated Wi-Fi Noise

In Section 3.1.3, it is demonstrated how Wi-Fi noise traces were collected using USB Wi-Fi dongles. Section 3.2.2 presented how Wi-Fi noise can be estimated with an MMPP(2) traffic model. However, from the IEEE 802.15.4 networks’ point of view, what is important to find out is how IEEE 802.15.4 channels are affected by the Wi-Fi noise. Note that a single IEEE 802.15.4 channel is affected by signals from one or more IEEE 802.11 channels, as depicted in Figure 3.1, experiencing aggregated Wi-Fi noise on that channel. There are two ways to estimate aggregated Wi-Fi noise on an IEEE 802.15.4 channel using the MMPP(2) traffic model:

a) **merge MMPP(2) models**: In this method, first, the MMPP(2) models that can estimate Wi-Fi noise on the overlapping IEEE 802.11 channels are parametrised before merging them to build the aggregated Wi-Fi noise model [37]. The models are parametrised with the IAT distribution of the collected empirical traces, as illustrated in Section 3.2.2. The merging of multiple MMPP(2) models is achieved with a technique called Kronecker sum [54].

b) **merge Wi-Fi traces**: Here, the merging of the Wi-Fi traces collected on the overlapping IEEE 802.11 channels is done first to obtain the aggregated Wi-Fi noise trace on the IEEE 802.15.4 channel, which is then used for parameterising the MMPP(2) model [38, 39]. The merging of Wi-Fi noise traces is done with the mergecap tool and the IAT distribution of the aggregated
Wi-Fi noise trace is obtained with the `tshark` tool. Both tools are part of the popular Wireshark network protocol analyser. As the MMPP(2) model is parametrised with the aggregated Wi-Fi noise, it can essentially estimate the aggregated Wi-Fi noise on the corresponding IEEE 802.15.4 channel.

![Traffic model for estimating aggregated Wi-Fi noise from 4 Wi-Fi channels](image)

Fig. 3.7 Traffic model for estimating aggregated Wi-Fi noise from 4 Wi-Fi channels (each state has distinct frame arrival rate $\lambda_i$, $1 \leq i \leq 16$).

Even though merging multiple MMPP(2) models can be easily done with the Kronecker sum, it increases the number of states in the resultant traffic model. Figure 3.7 depicts the aggregated Wi-Fi noise model for estimating noise on an IEEE 802.15.4 channel which overlaps with 4 Wi-Fi channels. For the estimation of Wi-Fi noise without a bias, a single MMPP(2) model requires 3 independent random number generators: two for each of its states to generate Poisson arrivals and one for the transition between the two states. As opposed to this, in the aggregated noise model, with the high number of states and transitions between them, the requirement for the independent random number generators escalates. This creates a high load in terms of memory and computation especially for a resource-constrained sensor node when it wants to estimate Wi-Fi noise as part of its MAC protocol for deciding whether or not to access the channel for communication. Note that the end goal of this thesis is to design an intelligent medium access mechanism that might use noise estimation for low-power wireless devices. Therefore, to estimate aggregated Wi-Fi noise, the second method with
3.2 Characterising IEEE 802.11 Noise

Table 3.2 Characteristics of aggregated Wi-Fi noise on IEEE 802.15.4 channels in the office and the home environments.

<table>
<thead>
<tr>
<th>office</th>
<th></th>
<th></th>
<th></th>
<th>home</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WSN Ch.</td>
<td>M_1 (ms)</td>
<td>C</td>
<td>H</td>
<td>WSN Ch.</td>
<td>M_1 (ms)</td>
<td>C</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>18.6</td>
<td>0.80</td>
<td>0.54</td>
<td>20</td>
<td>141.5</td>
<td>0.90</td>
<td>0.63</td>
</tr>
<tr>
<td>12</td>
<td>18.4</td>
<td>0.80</td>
<td>0.52</td>
<td>21</td>
<td>67.1</td>
<td>0.92</td>
<td>0.69</td>
</tr>
<tr>
<td>13</td>
<td>18.2</td>
<td>0.81</td>
<td>0.55</td>
<td>22</td>
<td>76.0</td>
<td>0.91</td>
<td>0.70</td>
</tr>
<tr>
<td>14</td>
<td>10.7</td>
<td>0.87</td>
<td>0.51</td>
<td>23</td>
<td>82.2</td>
<td>0.92</td>
<td>0.71</td>
</tr>
</tbody>
</table>

the merging traces from the overlapping Wi-Fi channels before parameterising the MMPP(2) model was used in this work.

In a nutshell, to investigate the aggregated Wi-Fi noise in IEEE 802.15.4 networks, the acquired raw real-world Wi-Fi traces from the two environments were processed as follows:

1. **Aggregating**: it aggregates Wi-Fi traces of multiple overlapping Wi-Fi channels as seen by IEEE 802.15.4 networks, i.e., concatenating raw traces and ordering them as a function of their timestamps. The output of this step is a trace of aggregated Wi-Fi noise.

2. **Noise characteristics extraction**: the IAT distribution and its corresponding statistics in terms of mean (M_1), coefficient of variation (C) and Hurst parameter (H) are determined from the aggregated Wi-Fi traces.

3. **Verifying self-similarity property**: the self-similarity of the aggregated Wi-Fi noise traces is determined by checking if the Hurst parameter (H) satisfies the following condition: H ∈ (0.5, 1). Note that the Wi-Fi noise must show self-similarity to estimate it using the MMPP(2) traffic model.

**Aggregated Wi-Fi Noise Characteristics.** Table 3.2 exhibits characteristics of aggregated Wi-Fi noise on IEEE 802.15.4 channels in terms of parameters M_1, C, and H of the Wi-Fi frame IAT distribution. From Table 3.2, one can clearly observe that the mean IAT of the aggregated noise is smaller, 10.7 ms, in comparison with that of the other channels in the OFFICE. The small mean IAT on channel 14 represents high noise on that channel. Moreover, with a Hurst parameter close to 0.5, i.e., 0.51, the noise on channel 14 exhibits less self-similarity. Furthermore, on
channel 20 in the HOME, the mean of IAT is almost double, i.e., 141.5 ms, than on other channels, i.e., 67-82 ms, as can be seen in Table 3.2. The reason for this is that IEEE 802.15.4 channel 20 does not overlap with Wi-Fi channel 11, see Figure 3.1, which continuously carried video-streaming Wi-Fi traffic. With a Hurst parameter considerably higher in the HOME than the OFFICE, the HOME exhibits stronger self-similar noise. It is noteworthy that values of $H$ slightly above 0.5 still reflects self-similar behaviour, but the degree of self-similarity is small on such occasions. The errors in the estimation technique can largely influence the value of $H$, which directly affects the performance of the MMPP(2) model.

3.2.4 MMPP(2) Model Performance

Model calibration. The performance of the MMPP(2) model depends on two parameters: $x$ and $y$. The former is the training duration of the MMPP(2) model and the latter is the duration of the estimated noise by the model. For a low $x$, the statistics from the training traces can not correctly capture the behaviour of the aggregated Wi-Fi noise, as the number of samples in such a small window is limited. The collected logs indicate that a value of $x \leq 1$ second leads to this. A larger $x$ is equivalent with an increase in the length of the training trace, which is prohibited in resource-constrained settings like the ones that the work here targets.

As the aim of this research is to estimate the short-term behaviour of the aggregated noise, $y$ should be as small as possible. Moreover, to accurately model aggregated Wi-Fi noise and to avoid bias of noise estimations by the two states, the MMPP(2) model must generate noise from both its two states at least once. This leads to a definition of a lower bound for $y$ as $y_{lb} = \frac{1}{\tau_1} + \frac{1}{\tau_2}$, where $\frac{1}{\tau_i}$ represents the mean duration of a state in the MMPP(2) model. Therefore, $y$ was configured

![Fig. 3.8 System calibration: variation of MMPP(2) metrics with $x$ and $k$.](image)
3.2 Characterising IEEE 802.11 Noise

during the calibration process to be an integer multiple of its lower bound $y_{lb}$

\( y = y_{lb} \times k, \ k \in \mathbb{Z}^+ \).

When evaluating the performance of noise estimation, it is important to quantify
the deviation of the estimated noise, i.e., forecasts, from the measured noise, i.e.,
observations. The estimation can be considered a good fit when the forecast
errors are minimum. Root Mean Square Error (RMSE) is the square root of the
squared error between the estimated ($\hat{x}$) and the measured ($x$) values as defined in
Equation 3.6. RMSE provides a measure of how good the forecast is with high
reliability [13, 22]. Therefore, to validate the accuracy of the noise estimation,
RMSE between the modelled Wi-Fi frame IAT and that of the testing set is used.

\[
RMSE = \sqrt{\langle \hat{x} - x \rangle^2}
\] (3.6)

As the goal is to reduce the RMSE between the modelled noise and the testing
set, different combinations of $x$ and $y$ were used initially to obtain a better
understanding of their impact on the RMSE. Following this, $y$ was fixed at $k = 1$
in the office and $k = 2$ in the home, and $x$ was varied as an exponential function
starting from 60 seconds to 40 minutes. Overall, all the evaluations, as depicted in
Figure 3.8a and 3.8b, show that a value of 1 for $k$ and 300 seconds for $x$ and 2 for
$k$ and 500 seconds for $x$ provide a minimum RMSE in the office and the home,
respectively. The training dataset (duration of which is $x$) is obtained from the
measured aggregated Wi-Fi noise trace. An MMPP(2) model is trained once for
each channel in the two environments considered.

**Model performance.** To evaluate the performance of the MMPP(2) model in
estimating the aggregated Wi-Fi noise, the MMPP(2) model was trained with the
aggregated Wi-Fi noise traces obtained after merging the collected Wi-Fi traces.
The length of the training set in the home for IEEE 802.15.4 channels 21-23 is
6500 samples, while that of the channel 20 is 3500 samples. The reason for the
fewer number of samples on the IEEE 802.15.4 channel 20 in the home is that
it was not under the Wi-Fi noise generated by the video-streaming traffic. In
the office, for IEEE 802.15.4 channels 11-13, the length of the training set is
16000 samples, while for channel 14, it is 28000 samples. The IEEE 802.15.4
channel 14 in the office overlaps with 4 different Wi-Fi channels, as depicted in
Figure 3.1, thus it is affected by the highest number of Wi-Fi noise generators,
leading to a high number of samples.

The evaluation of the MMPP(2) model is performed on the basis of RMSE
statistic computed on the generated noise trace by the MMPP(2) models versus
3.2 Characterising IEEE 802.11 Noise

an unseen aggregated Wi-Fi noise trace (testing set) obtained from the collected Wi-Fi traces.

Because it estimates short-term Wi-Fi noise and has claimed to accurately capture Wi-Fi signal characteristics, a state-of-the-art Pareto model [74] was also used for the noise estimation performance comparison. The Pareto model is defined by two parameters: $\alpha$ (scale) and $\beta$ (shape). Given the average Wi-Fi frame cluster IAT as $\lambda$, $\alpha$ and $\beta$ hold the following relationship: $\beta = \frac{\lambda}{x-\alpha}$. Here, the value of $\alpha$ defines the minimum length of white-spaces, i.e., time between two Wi-Fi clusters.

Note that in the worst-case with the maximum transmission unit of 133 bytes an IEEE 802.15.4 frame takes 4.256 ms for successful transmission. Therefore, as the end goal of this work is to accurately predict white-space for low-power wireless devices, $\alpha$ in the Pareto model was set to 4.256 ms which is the same value used as the transmission slot duration (there are 2 transmission slots within a sub-slot) in LUCID, as presented in Chapter 5.

Figure 3.9 shows the results of the comparison of the two models in terms of percentage RMSE. A few trends are clearly identifiable. First, the accuracy of the estimation decreases as one goes from the office to the home for both models, which is in line with the quantity of the noise decrease. However, this is not true for MMPP(2) on the IEEE 802.15.4 channel 14 in the office and 20 in the home. Secondly, the accuracy of the estimation of the MMPP(2) model is always higher than the Pareto model. The difference is more marked on channel 11 in the office and channel 20 in the home, Figure 3.9 shows a difference of 6.6 ms and 244.5 ms, respectively. However, channel 14 in the office shows a different behaviour, Pareto delivering the best performance on this channel even if there is more traffic. As presented in Table 3.2, the lower mean IAT in channel 14 in the office implies higher noise on it. Moreover, the noise on that channel exhibits
3.2 Characterising IEEE 802.11 Noise

Table 3.3 Comparison of absolute RMSE.

<table>
<thead>
<tr>
<th>WSN Ch.</th>
<th>office RMSE (ms)</th>
<th>home RMSE (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MMPP(2)</td>
<td>Pareto</td>
</tr>
<tr>
<td>11</td>
<td>1.6</td>
<td>8.1</td>
</tr>
<tr>
<td>12</td>
<td>0.7</td>
<td>5.2</td>
</tr>
<tr>
<td>13</td>
<td>1.4</td>
<td>4.4</td>
</tr>
<tr>
<td>14</td>
<td>4.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>

less self-similarity. Therefore, the characteristics of the traffic on channel 14 are more favorable for the Pareto model, which captures the mean IAT, than for the MMPP(2) which requires all three parameters \((M_1, C \text{ and } H)\) for traffic modelling. Note that of the accuracy the MMPP(2) model depends on the self-similarity of noise which is quantified by \(H\) and channel 14 exhibits the least self-similar noise among all the considered channels. Furthermore, the high mean IAT on channel 20 in the home translates into less noise on that channel compared to other channels, as shown in Table 3.2. As Pareto is a heavy-tailed distribution, it fails to capture the unsaturated traffic on this channel, having an RMSE of 259 ms. On the other hand, the MMPP(2) model performs the best on this channel, i.e., RMSE of 14.5 ms out of all the home channels.

The performance evaluation of the MMPP(2) model confirmed the versatility of the model for modelling both unsaturated and saturated aggregated Wi-Fi noise present on IEEE 802.15.4 channels. Even though Wi-Fi noise dominates the ISM bands with high transmission power (20 dBm), transmissions of low-power wireless devices could also be affected by other types of noise sources other than Wi-Fi. Because the focus of a single noise source to build an interference mitigation technique could lead to under-performed networks, it is essential to further study how low-power wireless devices such as IEEE 802.15.4-based sensor nodes perceive noise from unknown sources rather than individual ones. Moreover, there is only a handful of works investigating noise with the sensor node’s perspective [23, 74, 129]. Therefore, the subsequent study of the thesis has moved toward the sensor node’s perspectives to characterise noise from unknown sources.
3.3 Noise from Unknown Source

In this section, noise is investigated from the sensor nodes’ perspectives rather than from the perspectives of the source generating it. Consider an environment wherein wireless communication devices, such as Wi-Fi, Bluetooth, and IEEE 802.15.4-based sensor nodes, and non-communication devices, such as microwave ovens, that emit noise into the same frequency spectrum, are operating. In such a scenario, the interference that a low-power sensor node experiences is not specific to a particular wireless technology. Thus, the term noise from unknown sources is used throughout this thesis to denote any kind of noise that a sensor node might receive on its operating channel. As a starting point to the noise investigation, traces that capture noise as seen by the IEEE 802.15.4 devices is collected and characterised.

3.3.1 Measurement Locations

As introduced in Section 3.1.1, two typical indoor environments [71, 172], the office and the home, were selected in this study. The former is an office building, while the latter is a student dormitory. These environments exhibit different noise conditions for low-power wireless networks. Both environments consist of multiple sources of radio signals, such as Wi-Fi (APs, smartphones, and laptops), Bluetooth (wireless mice, keyboards, headsets, and stereo speakers located in the building as well as in the nearby buildings), and Microwave ovens (located in the kitchens of both environments), operating in the 2.4 GHz ISM band. The patterns of radio signals are different due to the peculiar behaviours/habits of users in the two environments.

3.3.2 Hardware and Software Platforms

The noise measurements from unknown noise sources were acquired by TMote Sky nodes [139], equipped with the ChipCon 2420 (CC2420) [137] radio chip, which is compliant with the IEEE 802.15.4 standard. Each sensor node has an integrated omni-directional inverted-F microstrip antenna. Three nodes were placed on walls along the hallway in both the office and the home at 1.75 m height and were connected to a USB port of a PC, as shown in Figure 3.10.

To enable noise detection, the Clear Channel Assessment (CCA) and Start of Frame Delimiter (SFD) pins of the CC2420 transceiver were used, leveraging the experience from [74]. In this case, when a signal above the CCA threshold
3.3 Noise from Unknown Source

Fig. 3.10 Experimentation setup to capture noise from unknown sources.

(i.e., −77 dBm, the default value used for the CC2420 transceiver) is detected, the CCA pin goes active indicating a busy channel, while the SFD pin going active indicates the start of an incoming IEEE 802.15.4 frame. Note that a channel is considered to be busy when its probability of a successful transmission is low due to noise on the channel, otherwise, the channel is considered to be free. The hardware interrupts associated with the two pins were captured, with CCA going active while SFD stays inactive indicating the presence of a noise signal. When this occurs, an interrupt signal which consists of one character is sent from the node to the PC, it is timestamped with milliseconds granularity and stored for
3.3 Noise from Unknown Source

The main findings reported in this thesis were gathered in the following three experimental campaigns: FIRST, SECOND and THIRD.

The FIRST and the SECOND were two short-term campaigns and were performed in the OFFICE. During the FIRST, three nodes were deployed at the same location between two Wi-Fi APs (Access Points) detecting the noise on IEEE 802.15.4...
Fig. 3.12 TMote Sky node placement in the OFFICE during the three measurement campaigns.
channels 13, 18 and 23, as depicted in Figure 3.11 and Figure 3.12a. For the second, the three nodes and the APs were interleaved and used channel 18 only (see Figure 3.12b). These choices paved the way to explore noise traces from IEEE 802.15.4 channels overlapping with different Wi-Fi channels and different characteristics of Wi-Fi traffic. The collection of traces was executed for 24 hours, during a working day of the week, from 1:00 PM and 4:00 PM in the office and the home respectively.

For the long-term third campaign, a single node was used to collect traces on channel 18 from both the office and the home. As the goal was to assess the noise for the long-term, the measurement campaign was run for two weeks, during September 12-26, 2017.

Note that a low number of SFD triggers (0 in home environment) was anticipated especially in office which did not have many WSN deployments. Appendix A presents snippets of measured noise traces from all measurement campaigns.

3.4 Characterising Noise from Unknown Sources

3.4.1 Metrics

The acquired noise traces are a set of timestamps when noise signals are detected by the IEEE 802.15.4 devices. These noise traces are characterised in terms of mean Time Between Noise Signals (TBNS) and the number of noise signals per slot statistics. Note that if there is no or single noise signal within a slot, the mean TBNS is considered to be the length of the slot. A slot is a fixed time period, e.g., 100 ms or 50 ms, and is further divided into sub-slots with 8.512 ms length in which data transmissions can be done, as discussed in Section 3.4.2. Although the mean TBNS is an informative statistical property of noise bursts, if used on its own to characterise the traffic within a slot, leads to an increase in the false discovery rate of busy periods. Figure 3.13 illustrates two free slots with and without burst of noise signals. For example, three noise signals can appear in a burst (e.g., small mean TBNS of 3 ms) or scattered (e.g., large mean TBNS of 20 ms). Note that small TBNS reflects that the noise signals are appearing continuously, thus representing a busy slot. Therefore, if the TBNS is used alone without the noise signal count, the slot is incorrectly identified as busy during bursty noise signals (Figure 3.13a), while for scattered noise the slot is correctly identified as free. To
reduce false discovery of busy slots due to noise bursts, both mean TBNS and signal count are used together.

3.4.2 Methodology

The goal of this thesis is to predict white-spaces for low-power wireless networks in the presence of noise in order to improve their performance in terms of communication reliability and energy efficiency. In general, a white-space is a temporarily vacant channel time, which can be used for communication by any co-existing wireless communication device operating in the same frequency band. In this thesis, a white-space is considered to be the duration of time in which an IEEE 802.15.4 frame and its ACK can be fully transmitted. In the current implementation of LUCID, ACKs are not used during data communication. Thus, the ACK space is also used by data packets especially when forwarding packets. In order to characterise noise and predict white-spaces with the help of the two metrics, mean TBNS and the signal count, the time is divided into slots and the slots are further divided into sub-slots. The length of a sub-slot is 8.512 ms ( = \( \frac{2 \times 133 \times 8}{250} \) ms) which is the time required for a 133 bytes IEEE 802.15.4 frame, as depicted in Figure 3.14, and its ACK transmitted at a rate of 250 Kbps [2]. As defined in Equation 3.7, the slots with least noise signals (i.e., at least one sub-slot is noise free so that a node can transmit without the packet being interfered) are identified as noise-free slots or simply FREE slots. The upper boundary of the length of white-spaces is the length of the time slot itself. All the time slots that are not FREE are marked as BUSY.

\[
    Channel = \begin{cases} 
        \text{FREE, if mean TBNS} > TH_{TBNS} \text{ and count} < TH_{count} \\
        \text{BUSY, otherwise} 
    \end{cases} \tag{3.7}
\]
3.4 Characterising Noise from Unknown Sources

Note that the length of a slot is considered to be 100 ms only for the noise characterisation. In the proposed MAC protocol in Chapter 5, a different value is experimentally determined with the help of the MAC protocol and used to evaluate its performance. The slot length used for noise characterisation is empirically determined that while higher values of the slot length result in high prediction accuracy, they reduce the energy efficiency of the nodes. Moreover, a node has to turn on its radio for the duration of a slot for receiving frames from its neighbours. Although a long free slot greatly contributes to high communication reliability especially at forwarding nodes, it certainly degrades the energy efficiency of the whole network depending on the network size. For a small sensor network with a few nodes, the transmission opportunities available within a short time slot could be sufficient. Short time slots also reduce the idle listening of non-forwarding nodes that could otherwise increase the energy consumption of the network. The longer the duration of a slot, the higher the energy consumption of a node becomes especially due to idle listening for packet reception. Therefore, 100 ms is a good trade-off for both. More information on the impact on the slot length toward the performance of low-power wireless networks is presented in Chapter 5.

To classify time slots into free or busy, two thresholds in relation with the mean TBNS and number of noise signals within a time slot are used: $TH_{TBNS}$ and $TH_{count}$. The value of the former is the lower boundary of a white-space, i.e., $TH_{TBNS} = 8.512$ ms, while the latter is $TH_{count} = 11$. The intuition behind the value of the $TH_{count}$ is as follows. One can divide a time slot into equal sub-slots with the length of its lower boundary 8.512 ms. These sub-slots are essentially the portions of white-space wherein actual frame transmissions can be done. In a 100 ms slot, there are 11 full sub-slots. If at least one of the sub-slots are free from noise the whole slot is marked as free. As explained in Section 3.4.1, the number of noise signals within a slot is the key to reduce the false discovery rate of the busy periods. Therefore, $TH_{count}$ along with the $TH_{TBNS}$ can be used to decide the state of the channel as defined in Equation 3.7.

---

<table>
<thead>
<tr>
<th>Bytes</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>127</td>
</tr>
<tr>
<td>Preamble Sequence</td>
<td>SFD</td>
<td>PHY Header</td>
<td>MAC Protocol Data Unit (MPDU)</td>
</tr>
</tbody>
</table>

Synchronization Header

PHY Service Data Unit (PSDU)

Fig. 3.14 IEEE 802.15.4 PHY frame structure [2].
In the example from Figure 3.13, the combined use of the two thresholds, correctly identifies the slot as **free** for both cases.

### 3.4.3 Noise Patterns

Once the acquired noise traces are characterised in terms of mean TBNS and noise signal count per slot, the two-dimensional Probability Distribution Function (PDF) of the two metrics is computed. The PDF of the mean TBNS and the signal count is crucial for identifying noise patterns. To this end, the PDF of each traces is computed on an hourly basis. By comparing hourly PDFs with that of the peak-hour, peak and off-peak hours are classified. To compare two probability distributions, the Normalised Cross-Likelihood Ratio ($NCLR$) [141, 91] metric was used.

**Normalised Cross-Likelihood Ratio.** The $NCLR$ is a measure of distance between two distributions and was first used in speech recognition to detect dissimilarities in speaker models [141]. Given two-dimensional probability distributions, $D_1$ and $D_2$, with the elements $X_i$ and $X_j$, and the number of elements $N_i$ and $N_j$ along the dimensions $i$ and $j$, the $NCLR$ is computed as in Equation 3.8.

$$NCLR(D_1, D_2) = \frac{1}{N_i} \log\frac{L(X_i|D_1)}{L(X_i|D_2)} + \frac{1}{N_j} \log\frac{L(X_j|D_2)}{L(X_j|D_1)} \quad (3.8)$$

Here, $\log\frac{L(X_i|D_1)}{L(X_i|D_2)}$ is the cross-likelihood ratio of the two distributions along the dimension $i$. Note that an $NCLR$ value close to zero indicates highly similar distributions.

The classification of peak and off-peak hours with $NCLR$ in turn are useful for training the traffic model appropriately. A threshold value of $0.5$ for $NCLR$ is used to distinguish among the two groups.

The characterisation of the noise trace is two-fold. The first is to compute the PDF of the noise traces w.r.t. their mean TBNS and noise signal count, as shown

Table 3.4 NCLR comparison of the noise traces from **FIRST** and **SECOND**.

<table>
<thead>
<tr>
<th>Channel</th>
<th>FIRST</th>
<th>SECOND</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0.78</td>
<td>0.12</td>
</tr>
<tr>
<td>23</td>
<td>0.22</td>
<td>0.88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0.78</td>
<td>0.12</td>
<td>0.88</td>
</tr>
<tr>
<td>18</td>
<td>0.22</td>
<td>0.88</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>0.78</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
3.4 Characterising Noise from Unknown Sources

Fig. 3.15 Probability distribution function of noise traces from FIRST on channel 13 (left), 18 (center) and 23 (right).

Fig. 3.16 Probability distribution function of noise traces from SECOND at location 1 (left), 2 (center), and 3 (right).

in Figure 3.15, 3.16, and 3.17. The other is to compute the NCLR for traces from the same campaign, from different channels (FIRST), time locations (SECOND) or weeks (THIRD).

- **first.** Through the PDF lens (see Figure 3.15), it appears that the noise on channel 13 and 23 are similar. This is further confirmed by the low value of $NCLR = 0.22$ in Table 3.4. On the other hand, the noise on channel 18 is different. It can be conjectured that this is a combined effect of the IEEE 802.15.4 channels overlapping with different Wi-Fi channels and being affected by different Access Points (APs). In the office building environment, the APs’ channel allocation is dynamic. In other words, an AP can autonomously switch to a different channel if its current operating channel has been experiencing heavy noise. This greatly affects noise characteristics on the IEEE 802.15.4 channels that are being overlapped with the said IEEE 802.11 channel.

- **second.** The geographical location of the nodes induces different trends in their PDFs, in this case location 1 and 2 show similar behaviour, as shown in Figure 3.16. This can also be seen in Table 3.4 with $NCLR = 0.12$, and explained by different noise characteristics induced by the position of the
3.4 Characterising Noise from Unknown Sources

Nodes in the proximity of two different APs (i.e., location 1 and 2 are close to AP1, while 3 is close to AP2).

- **third.** Figure 3.17 shows the results from the third campaign. A few trends are clearly identifiable. First, the quantity of the traffic increases as one progresses from the office to the home. The trend is more marked during the second week. Secondly, the traffic in the home is more bursty than the office, the PDFs show high probabilities in the bursty zone, mostly due to video streaming done by students in the home (i.e., student dormitory). Next, noise characteristics induced by day and night variations are investigated. Figure 3.18 shows the NCLR obtained from the comparison of one-hour peak trace with each one-hour noise trace for both environments. In the office, the one-hour peak trace represents the busiest traffic period during the day, while for the home it is during the night. In the office, Figure 3.18a and 3.18b, the patterns in the noise distribution over time of day and week-ends can be easily identified. The regions with high NCLR match with outside of office hours (7:00-22:00) time and the week-ends (19:00 Saturday-7:00 Monday) when there is no activity in the office building, therefore less noise. Moreover, an increase in the noise, with NCLR decreasing close to zero, can be observed during the busiest office hours, 10:00-11:00 and
3.4 Characterising Noise from Unknown Sources

Fig. 3.18 Noise patterns in the office and the home for two weeks.
13:00-15:00. Interestingly, Thursday night of the first week and the week-end days of the second week, show an increase in noise, which may be ascribed to a set of experiments run in the OFFICE building. In the HOME, the variations over time appear to be somewhat dependent on night (i.e., off-peak between 23:00 and 9:00) and day variations, but are not as clearly marked as in the OFFICE. Also, in the HOME the range of variations between FREE and BUSY periods is more dramatic, while the BUSY periods are smoother (i.e., longer bursty noise periods) than in the OFFICE. These are the effects of the HOME users and their devices (Wi-Fi/Bluetooth/microwave ovens) with no strict access time policies.

The observations from the experimental campaigns show that the environment in which the low-power wireless nodes are immersed, the location where the nodes are placed, and the channel used, have an impact on how the noise is perceived. Moreover, these observations directly inform modelling decisions, suggesting that at least two model configurations accounting for the peak and off-peak noise patterns should be adopted.

### 3.5 Summary

This chapter presented the characterisation of noise in low-power wireless networks in two ways: from the Wi-Fi perspective and the sensor node’s perspective. For the former, Wi-Fi traces were collected in two distinct indoor environments. The traces were combined to analyse the aggregated Wi-Fi disturbance on IEEE 802.15.4 channels. To this end, Wi-Fi frame IAT was used. It also validated the versatility of the MMPP$(2)$ model to estimate both saturated and unsaturated aggregated Wi-Fi noise.

However, to gain more understanding of the noise as seen by IEEE 802.15.4 devices, the investigation was moved to the sensor nodes’ perspectives, which laid the foundation for collecting noise from unknown sources in the same indoor environments with different settings. The characterisation of the noise traces, collected in three measurement campaigns, using the mean TBNS and noise signal count metrics, revealed the diversity of noise on different IEEE 802.15.4 channels, geographical locations, and environments. Moreover, characterisation of the traces w.r.t. the NCLR was used to identify noise patterns in the two indoor environments and suggested to use at least two model configurations for estimating noise from unknown sources accounting for the peak and off-peak noise conditions.
3.5 Summary

It is also noteworthy that Wi-Fi dongles perceive noise with more time granularity than sensor nodes due to their sensitivity difference, which is visible through the collected noise traces. Although the sensor nodes’ ability to perceive noise from all Wi-Fi frames individually is low, their sensitivity is adequate to detect noise, not only limited to Wi-Fi, that affects their transmissions.

The next chapter presents how the identified noise patterns are used to train the proposed noise model appropriately to estimate the noise from unknown sources and predict white-spaces with a white-space model for low-power wireless networks.
This chapter illustrates how the identified noise patterns in Section 3.4 are exploited for accurate estimation of noise regardless of its source of origination, and predict white-spaces for IEEE 802.15.4 networks. Section 4.1 presents the models (noise model and white-space model) that were used for these purposes along with the motivation behind the choice of the models. Proper training of the noise/white-space models is a prerequisite for achieving good prediction performance; the procedure that was followed for proper model training is explained in Section 4.2. The performance of the model is discussed in Section 4.3. Finally, Section 4.4 presents a summary of noise estimation and white-space prediction techniques that were proposed in this work and most importantly how these techniques can be used in a real IEEE 802.15.4 network.

4.1 Noise/White-space Models

The aggregated Wi-Fi noise presented in Section 3.2.3 has bursty and non-bursty behaviour w.r.t. the Wi-Fi frame IAT. This characteristic is readily captured in the two states of the MMPP(2) model. Unlike the aggregated Wi-Fi noise, the noise measured by the IEEE 802.15.4 sensor nodes was characterised by not only the mean TBNS but also the noise signal count per slot as explained in Section 3.4. The two-dimensional distribution of mean TBNS and the noise signal
count do not follow any existing bivariate standard distribution functions, such as the bell-shaped standard normal distribution. Therefore, to estimate noise from unknown sources, a Gaussian Mixture Model (GMM) was used. The intuition behind the choice of the model is that by tuning its parameters, a GMM can model an arbitrarily-shaped distribution smoothly [140]. Note that the term “noise model” is used in the following text to mean the GMM.

### 4.1.1 Gaussian Mixture Model

Before delving further into GMMs, let’s recall that there are two types of Gaussian distributions, i.e., normal distributions: univariate and multivariate. A univariate Gaussian distribution is one-dimensional and is parametrised with the two parameters, mean ($\mu$) and variance ($\sigma^2$). The Probability Distribution Function (PDF) of a random variable $X$ with a univariate Gaussian distribution can be defined as in Equation 4.1.

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$ (4.1)

On the other hand, a multivariate Gaussian distribution has more than a single dimension, thus to capture linear relationships between different dimensions, a multivariate Gaussian distribution takes covariance ($\Sigma$) into account. Similar to a univariate Gaussian distribution, Equation 4.2 defines the PDF of a multivariate Gaussian distribution.

$$p(x|\mu, \Sigma) = \frac{1}{\sqrt{2\pi\Sigma}}e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$ (4.2)

For a $Q$-dimensional random variable $X$, the $\Sigma$ can be defined as follows: $\Sigma_{i,j} = E[(X_i - \mu_i)(X_j - \mu_j)]$, where $\mu_i$ and $\mu_j$ are the means of $X$ in the dimension $i$ and $j$, respectively, where $1 \leq i, j \leq Q$.

A GMM is a superposition of multiple Gaussian distributions that collectively can model an arbitrarily-shaped distribution. A GMM can be formed of a mixture of either univariate or multivariate Gaussian distributions. Therefore, similar to Gaussian distributions, there are two types of GMMs as well: univariate and multivariate. Such a GMM consists of the following defining parameters:

- number of components, $\mathcal{M}$
- number of dimensions, $Q$
- component means, $\mu$
4.1 Noise/White-space Models

- components covariances, $\Sigma$
- weights of mixture components, $W$.

Equation 4.3 mathematically defines the PDF of a GMM, wherein $p(x|\mu_k, \Sigma_k)$ is the PDF of the $k^{th}$ Gaussian component, as defined in Equation 4.2, and $W_k$ is the weight of Gaussian component $k$.

$$p(x|\mu, \Sigma, W) = \sum_{k=1}^{M} W_k p(x|\mu_k, \Sigma_k) \quad (4.3)$$

The choice of the number of components ($M$) affects the estimation accuracy. The higher the number of components of a GMM, the higher the estimation accuracy of the model becomes [98]. However, with the increasing number of components, the memory load of the GMM also escalates. Each component $\mathcal{M}$ has $Q$ dimensions given by the number of features used to characterise the distribution. $\mu$ and $\Sigma$ define the mean and the covariance of each component. $W$ is a stochastic matrix that determines the weight at which each Gaussian component should model data.

Parameter selection. In this work, the number of components of the GMM was empirically determined. To establish the number of components for the GMM model, the noise estimated by the GMM model was compared to that of the collected traces during the measurement campaigns of this work. The comparison was performed with the Area Under Curve ($AUC$) metric, which can quantify the performance of the GMM at distinguishing between the FREE and the BUSY periods. Here, the optimal operating point for the model was determined while varying the number of components, and minimising the False Positive Rate.
(FPR), and maximising the True Positive Rate (TPR). The higher the AUC of the estimated noise, the better the estimate becomes. Figure 4.1 depicts the AUC of the GMM model w.r.t. the varying number of Gaussian components for different IEEE 802.15.4 channels. By looking at the figure, one can conclude that 7 components are good enough to estimate noise without increasing the model memory load further. Therefore, $M = 7$ was used throughout this work. Moreover, as the noise is characterised by the mean TBNS and the noise signal count per slot, a two-dimensional GMM was used, i.e., $Q = 2$. The other parameters, i.e., $\mu$, $\Sigma$ and $W$ are computed during the training phase of the model, which is illustrated in Section 4.2. Furthermore, the noise pattern analysis in Section 3.4.3 revealed the use of at least two noise model configurations for estimating noise from unknown sources in the indoor environments the HOME and the OFFICE. Therefore, in this thesis, a GMM with two different parameter sets, one for peak and the other for off-peak periods are used.

4.1.2 Hidden Markov Model

As discussed in Section 4.1.1, the GMMs are used for estimating noise from unknown sources as perceived by the IEEE 802.15.4 devices. This section explains how the output of the GMM model (noise model) is exploited to predict transmission opportunities for low-power wireless networks. To this end, a Hidden Markov Model (HMM) was used, as it can be easily combined with a GMM.

A hidden Markov model can be seen as a tool for obtaining the probability distribution of a sequence of observations and has been widely used in many application fields, such as speech recognition [149, 161], activity recognition [163, 9], gesture tracking [104, 111], gene tracking [126, 146], health and well-being [112, 120, 113], and wireless communications [142, 128, 21, 34, 117, 164, 46].

The term “white-space model” will be used in the following text to mean the combined model of GMM and HMM. How the prediction mechanism works is depicted in Figure 4.2. The noise model estimates noise and produces a sequence of observations for the HMM as a collection of TBNS and the number of noise signals on a slot basis. These observations are used to uncover the hidden state of the HMM for each time slot, which essentially produces a sequence of FREE and BUSY channel states. These channel states are the predictions for each time slot. In a nutshell, white-space model predicts white-spaces by taking estimated noise from the noise model as input and computing the hidden state of the HMM.
4.1 Noise/White-space Models

Fig. 4.2 Prediction mechanism.

Fig. 4.3 Bayesian representation of a hidden Markov model.
4.1 Noise/White-space Models

Figure 4.3 illustrates a sequence of observations of a HMM and their corresponding states of the underlying stochastic process. In other words, an observation $O_t$ at time $t$ is produced by a stochastic process, but the state $S_t$ of the process at time $t$ cannot be directly observed because it is hidden. The hidden stochastic process satisfies the first-order Markov property, i.e., the state $S_t$ solely depends on the previous state $S_{t-1}$ at time $t - 1$.

For clear illustration, the notation from [134] is adopted to indicate the complete parameter set of the HMM model:

- hidden (unobserved) states $S$
- number of states $N$
- observations $O$
- number of observations $M$
- state transition probability matrix $A$
- observation probability matrix $B$
- initial state probabilities $\pi$

A HMM consists of a set of states denoted by $S = \{1, 2, \ldots, N\}$ where $N$ is the total number of states in the HMM. These states often have some relation to the phenomena being modelled. For example, if a HMM is used for modelling a wireless channel, each state may represent the two different regimes of the wireless channel: FREE and BUSY. The observations $O = \{1, 2, \ldots, M\}$, where $M$ is the total number of observations, are a set of events that are observed by the HMM when it is in each of its states. Because a HMM consists of many states, there has to be a mechanism that defines the transition between these states. To this end, the state transition probability matrix $A$, a square matrix with dimension $N \times N$, is used. Each element of $A$, $a_{ij}$, describes the probability of transition from state $S_{t-1,i}$ to $S_{t,j}$ at one time step where $i, j \in \{1, \ldots, N\}$. The observation probability matrix or emission probabilities $B$ is an $N \times M$ matrix whose elements $b_{jk}$ describe the probability of observing $O_{t,k}$ given the state $S_{t,j}$, where $k \in \{1, \ldots, M\}$. This is written as, $b_{jk} = P(O_{t,k} | S_{t,j})$. When the HMM is parametrised and trained and is ready to execute its functionality, it starts in a randomly selected state. This first state is chosen with the initial state probability matrix $\pi$, which has dimension of $1 \times N$. It is worth noting that $A$, $B$ and $\pi$ are row stochastic matrices, as each
4.2 Estimating Model Parameters

Section 4.1 illustrated how the model parameters are initialised. This Section explains how to further tune the model and estimate their parameters for obtaining optimum performance.

GMM. After choosing $M = 7$ and $Q = 2$, the remaining defining parameters of the GMM, $\mu$, $\Sigma$ and $W$ need to be determined before noise can be estimated. Maximum Likelihood (ML) estimation is a technique used for computing model parameters of multivariate Gaussian models, however, it is not useful with mixture models as there is no closed-form solution. Thus, the remaining parameters of the
4.2 Estimating Model Parameters

GMM are estimated with the Expectation Maximisation (EM) algorithm, which is an iterative form of ML estimation [36].

Before using this technique to estimate model parameters of the GMM, \( \mu, \Sigma \) and \( W \) need to be initialised. This is accomplished with the k-means clustering algorithm [103] with \( M = 7 \) clusters for each dimension \( Q = 2 \). Moreover, the complexity of the EM algorithm increases based on the type of the covariance matrix \( \Sigma \); these types include full and diagonal. The latter approach, which is the most used in the literature, was adopted in this work as it requires less number of samples for training and approximates the full covariance using a linear combination of diagonal covariances. The EM algorithm was run until it converged to \( 1e^{-6} \).

**HMM.** The HMM that was chosen here, has two states, \( N = 2 \), and two observations, \( M = 2 \), as described in Section 4.1.2. Before computing the remaining parameters of the HMM, i.e., \( A, B, \) and \( \pi \) need to be initialised. This is done as follows. The model parameters \( A \) and \( B \) are initialised using uniformly distributed probability matrices while \( \pi \) is initialised for the data set under consideration, as shown in Equation 4.4. To initialise \( \pi \), the training data is labelled as FREE or BUSY with the help of the two thresholds, \( TH_{TBNS} \) and \( TH_{count} \), introduced in Section 3.4.2. \( n_{\text{FREE}} \) and \( n_{\text{BUSY}} \) are the number of FREE and BUSY labels in the training data set, respectively.

\[
A = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \\
B = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \\
\pi = \frac{1}{(n_{\text{BUSY}} + n_{\text{FREE}})} \begin{bmatrix} n_{\text{BUSY}} \\ n_{\text{FREE}} \end{bmatrix} \tag{4.4}
\]

After the initialisation of the HMM parameters, they are recomputed using the *Baum-Welch* algorithm [134]. The aforementioned algorithm, also known as the forward and backward algorithm, is a special case of EM algorithm that is used for estimating HMM parameters. The same convergence threshold, \( 1e^{-6} \), that was used in the GMM training phase was used in the *Baum-Welch* algorithm as well.

The training duration of the HMM determines the accurate estimation of the model parameters and, consequently, the white-space prediction. Therefore, a 1-hour training set was used as it provides statistical relevance, due to self-similarity of noise 3.2.3, for the channel behaviour. Note that in this thesis, an HMM with
two parameter sets are used for peak and off-peak periods. The training traces for peak and off-peak periods are obtained by computing the mean \( NCLR \) for the traces and picking the 1-hour trace closest to this mean.

## 4.3 Performance

### 4.3.1 Metrics

It is important to note that the measured noise trace is considered to be the ground truth when evaluating the performance of the models. The noise model and the white-space model are used to estimate noise and predict white-spaces, respectively. To evaluate the performance of the models, the ground truth is used, wherein the predicted trace by the model is compared against the ground truth trace on a slot basis.

To assess the performance of noise estimation defined in Section 4.3.2, two metrics were considered: accuracy and False Positive Rate (FPR). The former is a measure of the prediction ability of the model, while the latter provides an assessment of the packet loss of the low-power wireless network when the prediction mechanism is being used. In this work, False Positives (FPs) are more critical than False Negatives (FNs); the former is an incorrect prediction of a busy slot as free, thus, utilising such a slot by a sensor node could increase the number of lost packets.

The accuracy and FPR of the estimated noise trace are quantitatively evaluated w.r.t. the the ground truth trace. The output of the GMM model is a trace characterised in terms of the mean TBNS and the noise signal count per slot. To perform the comparison, mean TBNS and number of noise signals of both traces (estimated and ground truth) were translated into a channel state, busy and free, using the \( TH_{TBNS} \) and \( TH_{count} \) thresholds during each time slot. From

<table>
<thead>
<tr>
<th></th>
<th>Predicted</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BUSY</td>
<td>FREE</td>
</tr>
<tr>
<td>True Negative</td>
<td>False Positive</td>
<td>True Positive</td>
</tr>
<tr>
<td></td>
<td>(TN)</td>
<td>(FP)</td>
</tr>
<tr>
<td>False Negative</td>
<td>BUSY</td>
<td>FREE</td>
</tr>
</tbody>
</table>
Table 4.2 Performance of the GMM model with $\mathcal{M} = 7$.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>TBNS (ms)</th>
<th>#Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trace GMM</td>
<td>Trace GMM</td>
</tr>
<tr>
<td>Mean</td>
<td>9.4 10.6</td>
<td>12.9 9.9</td>
</tr>
<tr>
<td>Std.</td>
<td>7.7 5.9</td>
<td>9.3 5.8</td>
</tr>
</tbody>
</table>

this, the confusion matrix of the two channel state sequence is derived along with the metrics.

All metrics are derived from the elements of the confusion matrix in Table 4.1, as shown in Equation 4.5.

\[
\text{accuracy} = \frac{TP + TN}{TP + FP + TN + FN}
\]

\[
\text{FPR} = \frac{FP}{FP + TN}
\]  (4.5)

Moreover, to evaluate the performance of the white-space prediction, the Packet Loss Ratio (PLR) is used in addition to the accuracy in Section 4.3.3. These metrics are computed after comparing the predicted channel state by the HMM, i.e., busy and free, against the ground truth channel state.

### 4.3.2 Noise Estimation

The evaluation of the noise estimation is divided into two parts. First, the performance of the GMM model is assessed with different noise characteristics. For this, traces from all campaigns were used. Second, the performance of the proposed approach is compared with the state-of-the-art, a Pareto model [74], and with the previous approach based on an MMPP(2) model [38] to estimate the aggregated Wi-Fi disturbance on IEEE 802.15.4 channels described in Section 3.2.2. For the comparison, traces from the third campaign were used.

**GMM Performance.** The performance of the GMM is dependent on its ability to accurately estimate the PDF of the underlying empirical data. One can expect high estimation performance when the two PDFs are close enough. To examine GMM performance, a sample trace extracted from the measurements on channel 23 in the first is used. Figure 4.5 compares the two-dimensional PDFs of the empirical data and the corresponding modelled noise using the 7 components in the GMM. The corresponding statistics are summarised in Table 4.2. According to the table,
4.3 Performance

the PDF of the GMM is close enough to the empirical data as it can estimate the mean and the standard deviation (std.) of the TBNS and the noise signal count accurately, and this is further evident in Figure 4.5. The critical region in the two-dimensional PDF is the region near the two thresholds, 8.512 ms (TBNS) and 11 (signal count). The GMM model is able to estimate this region quite well (see Figure 4.5b), which is the reason for the high performance of the model as discussed below. Note that, typically, in the context of classification and machine learning, a model is considered to be good when its accuracy is above 90% [27].

To make a thorough investigation of the GMM performance, on different channels, at varying locations, and in multiple environments, confusion matrix based metrics were used. The results are shown in Figure 4.6, Figure 4.7, and Figure 4.8.

One can see that in the Office, during the 24 hours of the First and the Second campaigns, Figures 4.6 and Figures 4.7, the accuracy of the noise estimation is high (more than 93.2%), except from 10AM to 3PM in location 3 when the accuracy decreases as low as 82.8% and FPR increases up to 43.4%. It is conjectured that this behaviour is induced by the increase in the noise signal count during those hours, as shown in Figures 4.9.

During the Third measurement campaign, as shown in Figures 4.8, the accuracy of the estimation is high in both environments over the course of two weeks, with over 98%. Moreover, it is evident that the GMM model can estimate the behaviour of the noise in the Office better than the Home. This can be explained with reasons similar to those for the other campaigns. In the Home, as depicted in Figures 4.10, the noise is more bursty. In the Office, the estimation accuracy is
4.3 Performance

Fig. 4.6 Performance of GMM on different channels at the same location (FIRST).

Fig. 4.7 Performance of GMM on channel 18 at varying locations (SECOND).

Fig. 4.8 Performance of GMM on channel 18 for long-term noise (THIRD).
4.3 Performance

Table 4.3 GMM vs. alternative solutions in THIRD.

<table>
<thead>
<tr>
<th>Period</th>
<th>Metric</th>
<th>office</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GMM</td>
<td>MMPP</td>
</tr>
<tr>
<td>Weekday</td>
<td>Day</td>
<td>99.82</td>
<td>86.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.08</td>
<td>98.19</td>
</tr>
<tr>
<td>Night</td>
<td>Accuracy</td>
<td>99.98</td>
<td>87.95</td>
</tr>
<tr>
<td></td>
<td>FPR</td>
<td>0</td>
<td>99.34</td>
</tr>
<tr>
<td>Weekend</td>
<td>Accuracy</td>
<td>99.96</td>
<td>97.17</td>
</tr>
<tr>
<td></td>
<td>FPR</td>
<td>0</td>
<td>98.86</td>
</tr>
</tbody>
</table>

Table 4.3 shows the results w.r.t. the ground truth in the office and the home. It can be noted that the GMM approach achieves the best results, highest accuracy and lowest FPR, compared to the alternative approaches across all combinations of environments, channels, locations and time intervals. However, GMM is slightly worse in the home than the office, due to the more bursty noise in the home.

stable at 100%, except for a few occasions. Even those deviations are negligible as they are less than 1% in both weeks. On the contrary, the home exhibits much more frequent variations in accuracy and FPR, but the accuracy does not decrease below 98%.

Comparison with state-of-the-art. Now, it is time to compare the performance of the GMM modelling approach for estimating the noise from unknown sources with state-of-the-art, Pareto and MMPP(2). Table 4.3 shows the results w.r.t. the accuracy and FPR in estimating the actual noise for both environments in THIRD. For each different period in the life of the noise trace, a two-hour test trace was chosen. The three periods in Table 4.3 correspond to peak (day) and off-peak (night, week-end), the two off-peaks exhibiting different characteristics (i.e., different NCLR values).

The Pareto-based approach relies on the self-similarity property of the noise, meaning characteristics of the noise are preserved irrespective of scaling in time. Therefore, to ensure a fair comparison with Pareto, one has to resort to at most a two-hour test trace in which the traffic exhibits self-similarity. The approach assesses the state of the channel upon the arrival of an application packet, therefore, the white-space prediction probability is conditioned by this state.

Moreover, the performance of the MMPP(2) model depends on the training duration $x$ and the modelling duration factor $k$. Here, the MMPP(2) model was calibrated for maximising the AUC value, and used $k = 1$, $x = 240, 180, 300$ s in the office and $x = 420, 240, 540$ s in the home, for day, night and week-end.

It can be noted that the GMM approach achieves the best results, highest accuracy and lowest FPR, compared to the alternative approaches across all combinations of environments, channels, locations and time intervals. However, GMM is slightly worse in the home than the office, due to the more bursty noise in the home.
Nevertheless, the accuracy does not decrease below 99.42% and the FPR is lower than 1.16%. On the other hand, through the lens of both metrics, Pareto performs better in the home than the office. Although Pareto’s accuracy does not go over 32.41%, its FPR is at 5.99% during the busiest traffic periods in the home, arguably due to its distribution, i.e., Pareto models high bursty traffic. Notably, MMPP(2) is better than Pareto in correctly identifying the two states of the channel, BUSY and FREE, translated into higher accuracy. The FREE state is better identified by the MMPP(2) than the BUSY state, translated into high FPR. High accuracy and high FPR in MMPP(2) can be attributed to high amount of FREE slots in the trace than BUSY slots and MMPP(2) mainly estimating the FREE slots.

4.3.3 White-space Prediction

This section presents a performance evaluation of the GMM-HMM modelling approach that provides predictions of white-spaces. For the evaluation, an application that uses the GMM-HMM model to make decisions on when to transmit is considered. For clarity in the following text, the term “white-space model” is used instead of GMM-HMM. The performance of the proposed technique in this thesis is compared with two $p$-persistent channel access methods for $p = \{0.5, 1\}$ (i.e., transmission attempt is with probabilities 0.5 or 1 when the channel is sensed as idle) [40], and with the previous approach based on the combination of HMM model with MMPP(2) [38] that was illustrated in Section 3.2.2.

The white-space prediction works as follows; when a packet is ready to be transmitted, the white-space model produces a sequence of BUSY and FREE channel states, from which the closest FREE slot is selected. Then, the state of the chosen slot is compared against the corresponding slot of the ground truth. A packet is marked as lost if the predicted state is FREE while the ground truth shows BUSY. Note that there is still a probability of losing a packet in a FREE slot, which is however ignored in this analysis. Note that the white-space model predicts the closest FREE slot before the next scheduled packet transmission, and if the model is unable to predict a FREE slot, the current packet is considered as lost.

The performance of the white-space model is evaluated across the entire set of traces collected, i.e., FIRST, SECOND, and THIRD. Moreover, the impact on the reliability of a staple data collection application is investigated ensuring variability in the transmission interval, e.g. 1 s and 60 s, representing a high and a low data rate application. To create events in the 0.5-persistent random access method, a uniformly distributed random number generator was used. Furthermore, for
### Table 4.4 Comparison of white-space model prediction performance to alternative solutions.

<table>
<thead>
<tr>
<th>Period</th>
<th>Metric</th>
<th>OFFICE</th>
<th>HOME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GMM-HMM</td>
<td>MMPP</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
<td>Std. dev.</td>
<td>Mean</td>
</tr>
<tr>
<td>Weekday</td>
<td>Day</td>
<td>Accuracy</td>
<td>97.71</td>
</tr>
<tr>
<td></td>
<td>PLR</td>
<td>2.29</td>
<td>0.78</td>
</tr>
<tr>
<td>Night</td>
<td>Accuracy</td>
<td>99.72</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>PLR</td>
<td>0.28</td>
<td>0.32</td>
</tr>
<tr>
<td>Weekend</td>
<td>Accuracy</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>PLR</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Table 4.5 Comparison of white-space model prediction performance to alternative solutions: confidence intervals.

<table>
<thead>
<tr>
<th>Period</th>
<th>Metric</th>
<th>OFFICE</th>
<th>HOME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GMM-HMM</td>
<td>MMPP</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
<td>95% Confidence Interval</td>
<td>Mean</td>
</tr>
<tr>
<td>Weekday</td>
<td>Day</td>
<td>Accuracy</td>
<td>97.30</td>
</tr>
<tr>
<td></td>
<td>PLR</td>
<td>1.88</td>
<td>2.70</td>
</tr>
<tr>
<td>Night</td>
<td>Accuracy</td>
<td>99.51</td>
<td>99.93</td>
</tr>
<tr>
<td></td>
<td>PLR</td>
<td>0.07</td>
<td>0.49</td>
</tr>
<tr>
<td>Weekend</td>
<td>Accuracy</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>PLR</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
selecting the MMPP(2) parameters and the training duration for the MMPP(2)-HMM, the procedure presented in Section 3.2.4 was followed. It is noteworthy that for the training of GMM and HMM a 1-hour trace is used and for that of MMPP(2) a trace with a length of \(x\) seconds is used. The value of \(x\) is dependent on the environment and noise conditions and is heuristically determined as illustrated in Section 3.2.4. \(x\) is not randomly selected. The trace with a length of \(x\) contains the most recent ground truth samples in relation to the current prediction time. This training procedure is followed throughout the thesis.

**Comparison of performance.** To compare the performance of the models, noise measurements from the third campaign were used as they provide noise measurements in two different environments with day/night and weekday/weekend variations. Table 4.4 and Table 4.5 show a comparison of the results w.r.t. the ground truth. It is important to note that the models generate distinct predictions every time they are used. This is because the state of the random number generator changes whenever a prediction is made by the model. Thus, the prediction accuracy of the models changes over time.

First, across both environments and independent of the time of day/week, the accuracy places the proposed approach in this thesis ahead of the other methods and slightly (i.e., 3.06%) below the 1-persistent method in the home during weekdays. In the office, the proposed method is better than 1-persistent as it accurately predicts free states, increasing accuracy and decreasing PLR. Looking at the Table 4.4, one can notice that accuracy and PLR, for both white-space model and 1-persistent, sum up to 100%, because half of the confusion matrix (i.e., TNs and FNs) is always zero as the analysis is done based on the predicted free slots.

Secondly, in the office, the proposed approach performs worst during the day, with an accuracy of 97.71%, due to the high traffic (i.e., low mean TBNS, high number of noise signals). The accuracy decreases more as one progresses from the office to the home, going as low as 89.44%. This can be explained with arguments similar to those for the GMM performance, the home exhibits more bursty noise and the model does not deal well with short-term channel variations.

Thirdly, through the lens of PLR, the environment induces different trends: in the office, the proposed method performs slightly better than all the others (i.e., maximum PLR is 2.29%), while in the home it is the worst with PLR values as high as 10.56%. The previous approach based on an MMPP(2) model has low accuracy and as such is unable to accurately capture the characteristics of the noise. Although the 1-persistent random access method has a high accuracy in
4.3 Performance

Fig. 4.9 White-space model performance prediction and the \( PR \) of a 1 s (top) and 60 s (middle) data rate application in FIRST and SECOND, hourly variations of noise (bottom).

both environments, it fails in the HOME, always predicting the channel as FREE while in fact it is BUSY. As expected, the 0.5-persistent random access method has around 50% accuracy in both environments at all times.

**White-space model insights.** In order to further evaluate the proposed white-space model, attention is turned to hourly variations of the white-space model accuracy and application \( PR \) across all traces from the three measurement campaigns, as shown in Figures 4.9 and Figures 4.10.

In the OFFICE environment, on channel 23, location 3 and week 1, with the application at low data rate (60 s), the white-space model produces high accuracy (\( \geq 97.1\% \)) while the application has low \( PR \) (\( \leq 2.9\% \)). This is also valid with the high data rate (1 s) application, except on Thursday (minimum accuracy 56.7%) and Monday (minimum accuracy 66.7%) in week 1, and on channel 23 (minimum accuracy 65%) and location 3 (minimum accuracy 58.3%). When one looks at the HOME, the average accuracy, \( 88.3 \pm 16.8\% \), is lower than that in the OFFICE, e.g. \( 98.3 \pm 1.4\% \). The white-space model with the low data rate application in the HOME performs in a similar manner as in the OFFICE, with a minimum accuracy of 56.7%. However, the performance of the white-space model with the high data rate application in the HOME is worse than in the OFFICE. These observations can be explained as follows:
4.4 Summary

This chapter has demonstrated how to exploit noise patterns identified in Chapter 3 for an accurate estimation of the noise from unknown sources. The estimated noise

![Graphs showing accuracy and PLR over time for office and home environments.](image)

(a) The office.  
(b) The home.

Fig. 4.10 White-space model performance prediction and the PLR of a 1 s (top) and 60 s (middle) data rate application in THIRD, hourly variations of noise (bottom).

- the number of hourly noise signals has an impact on the accuracy of the prediction, e.g., across all campaigns, the model exhibits high accuracy when the number of noise signals is \( \leq 2 \times 10^5 \);

- variations of the noise signal count has an impact on the accuracy of the prediction, e.g., sudden and dramatic variations decrease the accuracy;

- sub-optimality of the HMM model, which is good in characterising the distributions and channel state transitions but has difficulties in capturing the variations of the noise in time.

The impact of the amount and the variations of the noise signal count is more marked in the home than in the office. This translates into lower accuracy and higher PLR in the home. Additionally, during the periods with the most severe noise, the number of available free slots is limited, getting as low as 20%, resulting in limited transmission opportunities.

4.4 Summary

This chapter has demonstrated how to exploit noise patterns identified in Chapter 3 for an accurate estimation of the noise from unknown sources. The estimated noise
was then used to predict white-spaces for transmission opportunities in low-power wireless networks.

For the estimation of noise from unknown sources, a Gaussian Mixture Model (GMM) was used as it can estimate non-standard distributions with properly selected parameters. Two-dimensional noise properties, i.e., Time Between Noise Signals (TBNS) and the noise signal count per slot, and a GMM with two different parameter sets, for peak and off-peak noise conditions, was utilised as described in Chapter 3. The performance of the GMM was evaluated in varying noise conditions, which demonstrated its high accuracy in predicting both peak and off-peak noise periods in comparison with state-of-the-art methods. Because the GMM exhibited high noise estimation accuracy, further steps were taken toward white-space prediction with a Hidden Markov Model (HMM), which can be combined easily with the GMM. Similar to GMM, an HMM with two different settings was used for white-space prediction, i.e., for peak and off-peak noise. The performance of the white-space model was evaluated with an application running at varying transmission intervals, 1 second, and 60 seconds, representing high and low data rate applications, respectively. The results demonstrated high accuracy, i.e., more than 97.7%, and low Packet Loss Ratio (PLR), i.e., less than 2.3%, in the office with varying noise characteristics as opposed to the home. It is noteworthy that the amount of hourly noise signals and their variations has an impact on the prediction accuracy. The model exhibits high accuracy when the number of noise signals is \( \leq 2 \times 10^5 \) and sudden and dramatic variations decrease the accuracy. Moreover, the HMM has difficulties in capturing time variations of the noise due to its sub-optimal nature. However, as a solution to the sub-optimality of the HMM, the model can be adapted to sudden changes in the noise by continuous monitoring of the performance for its re-parametrisation.

As the white-space model showcased its capability to estimate noise and predict white-spaces with high accuracy in the scenarios considered, one can embed the proposed technique in the IEEE 802.15.4 protocol stack in order to increase the reliability of low-power wireless networks. Chapter 5 discusses the steps behind this to achieve a high performance low-power wireless network with LUCID.
Chapter 5

Model-based Receiver-aware Communication

Radio frequency noise in a shared medium is an immense problem for the coexistence of collocated dissimilar wireless networks, especially for low-power, resource-constrained networks such as IEEE 802.15.4 based networks. There are many reactive techniques to address the coexistence problem wherein the medium is checked before transmissions. However, there is still a possibility that the packets may get corrupted during the transmission leading to packet losses [71]. Due to this reason, the energy consumption of devices increases because of re-transmissions to maintain communication reliability.

The Medium Access Control (MAC) layer plays a key role in all wireless networks. It is mainly responsible for the arbitration of access to the shared medium, controlling when the radio is switched on or off, and providing a reliable link between two or more peer MAC entities [2]. The MAC may also make use of duty cycling for reducing energy consumption on each node. Duty-cycling incurs an additional problem of finding a rendezvous points between a sender and a receiver, in which both nodes are in the radio on state and the communication link can be established. There are three approaches to solve this problem: synchronous, asynchronous, and semi-synchronous [31, 153, 11].

In the synchronous paradigm, all nodes synchronise to an active/sleep schedule either with a centralised or a decentralised scheduling mechanism. Even though this technique reduces idle listening and collisions at the same time and is resilient to schedule misalignment, it incurs an additional energy overhead for control messages. Moreover, synchronous scheduling techniques inherently have reduced flexibility and scalability and increased algorithmic complexity [31]. Asynchronous
schemes on the other hand do not require clock synchronisation of nodes as they wake up independently of each other. There are two main asynchronous techniques for finding the rendezvous point: sender-initiated and receiver-initiated. In the former, the sender transmits a preamble to indicate its presence to the receiver which occasionally wakes up into the radio-on state. When the receiver detects channel activities, it just remains active and receives first the preamble and then the payload. In contrast to this, in the receiver-initiated technique, the receiver transmits periodic short beacons that are used for synchronisation by the sender. The receiver-initiation paradigm was introduced in RICER [101] and made popular by RI-MAC [154]. Semi-synchronous schemes explore the best of both synchronous and asynchronous paradigms, wherein neighbours are grouped into synchronised clusters and the clusters interact with each other asynchronously. Electing the optimal cluster-head, intra-cluster coordination, and efficient inter-cluster traffic relaying are challenges in the semi-synchronous techniques.

The white-space prediction mechanism introduced in Section 4.1.2 was designed as a technique to tackle CTI. However, later, it turns out that the same mechanism can be easily embedded into a MAC protocol for finding rendezvous points between a sender and a receiver. This chapter presents a novel model-based receiver-aware proactive MAC protocol called LUCID that exploits the proposed white-space prediction mechanism introduced in Chapter 4 to address the co-existing problem in a shared medium. As the same white-space model is used on both sides of the communication channel, the white-space model is able not only to predict transmission opportunities at the sender but also to synchronise receiver’s communication events with that of the sender to find rendezvous points. The added functionality of the white-space prediction mechanism considerably alleviated the design of the novel MAC protocol to address CTI.

The rest of the chapter is organised as follows. A complete overview of LUCID is presented in Section 5.1, while Section 5.2 explains the prerequisites for the proposed MAC protocol. Section 5.3 illustrates how the white-space prediction mechanism is embedded in the IEEE 802.15.4 protocol stack. Practical issues encountered during the design of the technique and how they were overcome by are revealed in Section 5.4.1. The performance of LUCID is presented and discussed in Section 5.4. Finally, Section 5.5 presents a summary of the proposed technique.
5.1 Overview of the Design

LUCID, as depicted in Figure 5.1, consists of two phases: deployment and model-based data communication. Since the proposed solution is a model-based technique, the models in use need to be trained before they can be used.

In LUCID, before going operational to execute the application tasks, the nodes in the network must acquire an accurate understanding of the noise in the surrounding environment. During this phase, noise traces are collected in order to assess noise characteristics in the deployed environment. This process helps the nodes to perform three key functions depicted in Figure 5.2: measure and characterise noise, and train white-space models. Measuring noise is a costly task in terms of energy as the nodes should continuously sample the radio environment to gather representative noise traces. Therefore, as a solution, a separate network of nodes can be used to accomplish noise measurements. If the physical locations of the nodes in the secondary network do not match with that of the main network (with a reasonable degree of physical accuracy), the collected noise measurement would not represent the radio environment around the nodes in the main network that executes the application tasks. Thus, care should be taken when placing the nodes as the two networks must be identical in terms of device types, configurations (e.g., CCA threshold) and their locations. The collected traces can be used to evaluate the performance of the wireless network if the noise measurements are good enough to represent the radio environment around the main sensor network. The output of the deployment phase is a set of trained models that can estimate noise and predict white-spaces for low-power devices. Section 5.2 illustrates the functionality of the deployment phase in detail.

At the end of the deployment phase, the model-based data communication can start its operation, as shown in Figure 5.1 and Figure 5.3. Here, sensor nodes go first through an initialisation phase in which a) time synchronisation, b) acquisition of routing information, and c) exchange of white-space models are accomplished. As the sensor nodes need to wake up on a locally determined schedule set by the white-space model and the senders transmit data in line with the said schedule, the proposed solution uses a slot based mechanism with short time slots for medium

![Fig. 5.1 Design overview of LUCID.](image-url)
access. Therefore, a tight synchronisation of clocks in the sensor network plays a vital part for the time slots to be aligned. Periodic time synchronisation allows all the nodes in the network to update their clock drifts w.r.t. that of the network coordinator. The knowledge about its next hop is an essential piece of information for a node. This knowledge is acquired during the network initialisation phase from the routing protocol running on the node, and the proposed solution relies on the underlying protocols that perform neighbour discovery and routing. It is important to note that routing information plays a vital part in the proposed technique as a white-space model is used to predict transmission opportunities at the receiver and the receiver’s model should be available at the sender.

Because the radio environment has a dynamic nature, the varying routing metrics force the routing protocol to change the routes to have high communication reliability, leading to a new routing structure. Even though LUCID is highly dependent on the underlying routing protocol to discover the next hop, the alterations in the network topology do not affect the solution as all sensor nodes have access to all their neighbouring white-space models (each with 834 bytes), and when the next hop changes the correct white-space model is readily available in the node. Nonetheless, keeping the white-space models of all the neighbours increases the memory footprint of nodes. Therefore, as a countermeasure to decrease the memory footprint, periodic or trigger-based routing updates and white-space model sharing can be implemented; however, investigation of this is not a part of this doctoral thesis.

During the network initialisation phase, nodes also exchange white-space models that were computed during the deployment phase. A node broadcasts its models in a round-robin fashion to confirm all the nodes received their neighbours’ white-space models which are crucial for data transmission.

After exchanging models, nodes go into radio duty cycling to save energy consumption. A node maintains three radio states related to communication: receive-active, transmit-active and sleep. Each node sleeps according to the duty-cycle predictions from the white-space model. In particular, a node goes into the receive-active state when it may receive communications, while it goes to the transmit-active state to transmit data in a receiver-aware manner. In the following description, a periodic data communication application is assumed as this is one of the most common use-cases for a wireless sensor network and LUCID is designed for periodic data. Whenever a packet is ready to be transmitted, determined by the data period timer, a node wakes up and utilises its own transmit sub-slot
within a free receive slot of the receiver predicted by the white-space model of the receiver, which was shared by the neighbour during the network initialisation. For successful communication, both ends of the communication channel synchronise their radio state. By sharing the models, the receiver and the transmitter can separately (locally) calculate the same schedule of free slots for the receiver. It is worth noting that each node goes into its receive-active state, i.e., switches on its radio, to receive packets from its neighbours, following the predictions of its own white-space model. The term “own white-space model” is used here and in the following text to mean the receiver’s white-space model. In the circumstances where the own white-space model doesn’t predict a free receive slot, the receiver considers the very next time slot following the first time slot of the data period as free. Therefore, the rendezvous is readily achieved when the transmitter utilises the predictions from the neighbour’s white-space model. The packet transmission is done irrespective of the presence of noise at the sender.

Because the white-space models need to be adapted to the dynamic noise conditions, while the application is running, the network coordinator periodically scrutinises the network-wide Packet Delivery Ratio ($PDR$). If the moving average of $PDR$ crosses below a predefined threshold, the network coordinator triggers a command to re-parametrise the model. Dissemination of the command is accomplished by flooding a re-parametrisation control packet throughout the network during the dedicated very first time slot of the data period. Upon receiving the re-parametrisation control packet, a node switches to the second parameter set (not immediately as discussed in Section 5.3.4) of the white-space model which was computed during the deployment phase, i.e., from peak to off-peak model, and vice versa. Section 5.3 describes LUCID in more detail.

## 5.2 Deployment

Once deployed, an IoT network, such as an IEEE 802.15.4 low-power wireless network, works autonomously to execute the application tasks that it is programmed to do. However, in LUCID, before going operational, the sensor nodes in the network must undergo an additional step to measure the noise in the surrounding radio environment. During this phase, noise traces are collected in order to assess noise characteristics in the deployed environment. Measuring noise to gather traces that capture the properties of channel activities and can represent the radio medium requires a considerable amount of energy budget. This is too much for resource-
constrained nodes as they have to spend a lot of energy even before starting the main application tasks. Therefore, to tackle this problem, a separate overlay network can be used. To gather representative noise measurements, the secondary network must be identical to the main network which executes the application tasks in terms of the node types used, configurations (e.g., CCA threshold) and their positions while measuring noise.

The deployment phase helps the nodes to perform three key functions depicted in Figure 5.2: measure noise, characterise noise, and train white-space models.

### 5.2.1 Measure Noise

Before commencing data communication, all the sensor nodes in the wireless network measure noise; this is the fundamental task a sensor node must perform to understand the radio noise. To achieve this, a sensor node listens on its operating IEEE 802.15.4 channel and classifies the receiving signals as noise or not based on the two hardware interrupts of the transceiver: CCA and SFD; this was illustrated in more detail in Section 3.3. When received, the noise signals are timestamped and stored for further processing later. It is assumed that the noise signals that emerge during the measurement period are representative of the noise in the environment, which can be used for training white-space models at a later stage. Therefore, the noise measurement period is dependent on the environment in which the sensor network is deployed. Nonetheless, it is advisable to measure noise for at least 24 hours. The reason for such a long noise measuring period is to capture peak/off-peak variations. A timer is used by the nodes to identify the noise measuring cycle. Sampling the radio environment for 24 hours costs nodes a significant energy budget. Therefore, it is sensible to use a separate network of nodes for collecting noise traces during the deployment phase as was done in this work. The models trained from the traces collected can then be used by the deployed nodes that execute the main application. At the end of this period, all the sensor nodes in the wireless network have their lists of timestamps at which they received noise.
5.2.2 Characterise Noise

In this step, the noise traces collected by each sensor node are processed locally to characterise noise at that node’s location. The characterisation is achieved with the two previously introduced metrics: mean TBNS and the noise signal count. The two metrics are calculated on a slot basis, i.e., the mean TBNS and the noise signal count within a fixed time duration are computed. As presented in Section 3.4, the two-dimensional distribution of the two metrics can then be obtained for each hour of measurements, from which the peak and the off-peak hours in terms of noise can be identified with the NCLR. How the NCLR was used for identifying noise patterns is explained in Section 3.4.3 in more detail.

5.2.3 Training Models

After identifying the peak and the off-peak hours and the corresponding two-dimensional probability distributions of the mean TBNS and the noise signal count, the sensor node is ready to train the models, i.e., GMM and HMM, which were introduced in Section 4.1. For clarity, the terms “noise model” and “white-space model” are used instead of GMM and GMM-HMM, respectively. These models are trained in such a way that they can estimate noise and predict white-spaces at each node’s location. To this end, the procedure illustrated in Section 4.2 was used. Note that two sets of model parameters, one for peak and the other for off-peak noise, are used for both GMM and HMM. These trained models play a vital role in LUCID, presented in Section 5.3, in order to predict white-spaces for the communication between sensor nodes in a receiver-aware manner.

5.3 Model-based Data Communication

This section presents the data communication technique which aims at tackling radio frequency noise to improve the performance of low-power wireless networks. Figure 5.3 depicts an overview of the operation of the proposed solution which consists of four essential functions: network initialisation, model parameter selection, receiver-aware communication, and PDR monitoring and feedback. The timeline of events of LUCID is shown in Figure 5.4. The following subsections present the operation of these functions in detail.
5.3 Model-based Data Communication

5.3.1 Network Initialisation

The network initialisation is the first stage in the model-based data communication, which starts after the deployment phase. This is an important task as it sets the ground for the receiver-aware communication which will be explained in Section 5.3.3. Network-wide time synchronisation, acquiring knowledge about neighbours and the next hop, and the exchange of white-space models are accomplished during this stage.

5.3.1.1 Time Synchronisation

Distributed systems, such as low-power wireless networks, often require a time synchronisation service to enable the coordination between the devices in the network. Especially for time slotted wireless networks, tight synchronisation is essential to do their transmission at the scheduled times. Because the proposed approach also uses time slots, a simple synchronisation mechanism with a flooding-based technique, inspired by Trident [160], is implemented for global time synchronisation. Flooding is relevant in this case as the same global information (even though they come through different paths) is broadcast in the whole network.

---

Fig. 5.3 Overview of the model-based data communication.

Fig. 5.4 Timeline of LUCID events starting from the deployment.
The topic of flooding/synchronous transmissions for low-power wireless networks is extensively studied in the past and the solutions can be categorised based on the communication pattern, traffic pattern, type of packets, reliability mechanism, and network state [173]. The solutions include Glossy [53], Flash [106], Trickle [95], FTSP [114], PulseSync [94], Sparkle [170], Sleeping Beauty [144], and Chaos [88] among many others. Many of these methods exploit the capture effect, which enables a receiver to correctly receive one of the synchronously transmitted packets provided that the incoming signals satisfy certain power and timing conditions. The same information can also be disseminated throughout the network via techniques other than flooding, such as the DODAG version number inside the DIO messages in the RPL protocol [76, 105].

In low-power wireless data communication applications, one sensor node of the wireless network, the network coordinator, collects all the data packets and delivers them to the external server wherein data is stored, processed, and analysed. The sensor nodes in the wireless network communicate with the network coordinator using one of many dynamic network topologies, such as Star, Ring, Mesh, Grid, and Tree [82, 116]. Irrespective of the topology being used, a sensor node sends/forwards the data packets to its next hop which follows the same procedure until the network coordinator receives the packet. As the sensor nodes communicate with their next hop in the scheduled time slots, it is mandatory to synchronise all the clocks of the network with that of the network coordinator so that the events in the network are harmonised. In IoT applications wherein physical objects are envisaged to interact with each other to achieve a particular objective, time synchronisation is essential if LUCID is used. Regardless the network topology and use case, the scheduling of packet transmissions is distributed and implemented locally in the sensor nodes based on the white-space model computed in the deployment phase.

The time synchronisation mechanism is initiated with a broadcast of a control packet, called a synchronisation packet (see Appendix D.1), by the network coordinator node as depicted in Figure 5.5. Such a packet mainly consists of three fields:  

- a) timestamp,
- b) authoritative level, and
- c) time offset with the network coordinator.

When the protocol is being initialised, the network coordinator has the lowest authoritative level, i.e., 0, while the rest of the nodes in the network have 255. After the initialisation phase is over, the network coordinator creates a time synchronisation packet by setting both the authoritative level and the time offset of the control packet to 0 and timestamp the packet before broadcasting it. Note that a sensor node in the wireless network updates its clock upon receiving
a time synchronisation packet if and only if its authoritative level is higher than that of the received control packet. Moreover, to avoid deadlocks, when a node adjust its clock drift, the node’s authoritative level will also be set to one higher than that of the received control packet.

Updating the clock drift is done as follows. Denoting a node’s clock drift with its neighbour as $\Delta t_N$ and the neighbour’s clock drift with the network coordinator as $\Delta t_C^N$, a node’s clock drift with the network coordinator, $\Delta t_C$, has the relationship in Equation 5.1.

$$\Delta t_C = \Delta t_N + \Delta t_C^N$$ \hspace{1cm} (5.1)

When a node receives a synchronisation packet from one of its neighbours, the node computes $\Delta t_N$, which is the difference between the timestamps of the packet transmission at the neighbour and its reception at the node. The former is available in the synchronisation packet itself and the latter is available at the node as the node records the timestamp at which the packet is received. Moreover, the node receives the value of $\Delta t_C^N$ along with the packet. Therefore, assuming the processing time at the node is negligible, the clock drift can easily be computed by Equation 5.1 and once it is done the node updates the three fields of the packet and rebroadcasts it to make sure all the other nodes in the network are synchronised as well.

The network coordinator floods the network with three synchronisation packets with a tunable time gap in between to ensure all nodes in the network received at least one such packet. If a node didn’t receive a time synchronisation packet only means that the node’s clock is not synchronised, but it is still part of the network. As it is important to keep the clocks of the nodes updated to minimise
5.3 Model-based Data Communication

In contrast to this approach, Trident [160] corrects clock drifts per round basis whereas in LUCID it is periodic and independent of the data period.

5.3.1.2 Routing and Model Exchange

Routing is an essential element of communication networks, as it facilitates efficient data delivery to their destination. Moreover, routing may ensure the minimum energy consumption and hence maximisation of the network lifetime, especially in low-power wireless networks, as it affects the performance of the network. Many radio environmental factors, such as reflection, refraction, diffraction, and interference, affect the quality of links between the nodes and subsequently the routing performance [18]. The dynamic nature of the radio environment can forge poor links that cause nodes to have stale topology information, which might create routing loops, leading to packet losses.

Despite its importance, routing in low-power wireless networks is not part of the investigations in this work. Nonetheless, routing plays a vital part of the receiver-aware communication that will be explained in Section 5.3.3, as it delivers the information regarding the next hop. To this end, existing routing protocols, such as the Collection Tree Protocol (CTP) [61], can be used on top of LUCID to acquire knowledge about the next hop. In this work, static routing with CTP was used.

Furthermore, a sensor node must know the white-space models of its neighbours for the functioning of the proposed solution. Therefore, the nodes must share the computed models with their neighbours. The sharing of models is done in the initialisation phase after routing has finished creating the initial topology. Because the models are shared on a slot basis (see Appendix D.2 for the packet format used), the nodes need to be synchronised for receiving them from their neighbours. To avoid collisions among the simultaneously transmitted packets, it is ensured that only a single node broadcasts its models in a given slot. This is done with round-robin transmissions with the node identifier determining the turn of the
node to broadcast its models using random access. The sequence number of the packet is used as a way of confirmation of successful packet transmission.

The time at which a node broadcasts the models, $t_{BC, TX}$, is expressed in Equation 5.2, wherein $t_{BC, start}$, $n$, $N_{broadcast}$, and $T_{window}$ denote the start of the broadcast period, the node identifier, the number of broadcast windows, and the duration of a single window, respectively. Note that $N_{broadcast}$ is dependent on the maximum number of neighbours a node might have in the network and is a configurable parameter whose value is heuristically estimated. $T_{window}$ is set to be 200 milliseconds to allow a node to fully broadcast its model (834 bytes in size), which consumes $\geq 740 \mu A$s assuming 17.4 mA TX current.

$$t_{BC, TX} = t_{BC, start} + (n \mod N_{broadcast}) \times T_{window} \quad (5.2)$$

At the end of the model sharing period ($= N_{broadcast} \times T_{window}$), the sensor nodes in the network possess the models of their neighbours. These models are important to a sensor node for receiver-aware communication. Although only the models of the next hop are used by the node at any given time for accessing the shared medium, the rest of the models could be used when the routing topology changes due to poor link qualities. At such occasions, as the white-space model of the new next hop are readily available, the node does not have to request white-space models from the neighbours again. Therefore, by keeping a set of white-space models of its neighbours, a node saves energy significantly while minimising delays due to radio environmental changes.

### 5.3.2 Model Parameter Selection

Radio frequency noise in wireless networks has a dynamic nature. Especially in low-power wireless networks, depending on the operating channel, deployed location, and the environment, the radio frequency noise perceived by the sensor nodes is unique, as illustrated in Section 3.4.3. Therefore, interference mitigation solutions should be able to adapt to these variations in the radio environment and adjust their parameters/configurations to improve the performance of low-power wireless network.

LUCID exploits the white-space models that are capable of estimating noise and predicting white-spaces at the node’s location. Due to the dynamic nature of the noise, the models should also be re-parametrised to comply with the fluctuating noise patterns. This is achieved as follows. As advocated in Section 3.4.3, a
white-space model with two parameter sets, one for the peak and the other for the off-peak noise, should be used for accurate estimation of noise and subsequently for predicting white-spaces. Therefore, LUCID can choose one out of two parameter sets for the white-space model that might improve the performance of the network.

The selection of the appropriate model parameter set is solely based on the current performance of the data communication application, which is quantified with the $PDR$ metric. Here, $PDR$ is considered as an indirect measure of packet loss due to noise, which is very hard to measure and even harder to collect related results up to the network coordinator. As depicted in Figure 5.3, the $PDR$ feedback loop triggers the model parameter selection. However, care should be taken when switching the parameter set as it can improve or deteriorate the performance of the wireless network. More details on how to make the decision on when to switch the model parameters and the impact of this decision on the performance of the proposed receiver-aware data communication mechanism are discussed in Section 5.3.4.

After making the decision, the network coordinator disseminates the re-parametrisation command by flooding the wireless network with a re-parametrisation control packet (see Appendix D.1) in the dedicated broadcast time slot which is the first slot in the data period. Fields of such a model re-parametrisation control packet consist of the age and the type of the model that is going to be used, i.e., peak or off-peak. Similar to the authoritative field in a time synchronisation packet, see Section 5.3.1.1, a node uses the age field to avoid loops in the flooding. The coordinator initiates the flooding with the age of zero and receiving nodes increase its age by 1 before rebroadcasting.

Also, the age field is used as a means of acknowledgements. A node deems the received re-parametrisation packet as an acknowledgement when the packet’s age is 1 higher than the one that is saved by the node itself. The coordinator broadcasts a maximum of five re-parametrisation packets.

### 5.3.3 Receiver-aware Communication

This section presents the novel receiver-aware communication technique for low-power wireless networks. Unlike traditional receiver-aware communication with periodic beacons, the proposed approach of finding the rendezvous points between a sender and a receiver is based on the white-space model that was introduced in Section 4.1.
Prerequisites. Before the commencement of the proposed receiver-aware communication, the sensor nodes should satisfy themselves of the following:

- all the sensor nodes have measured and characterised the local radio frequency noise and have parametrised the white-space models (peak and off-peak),
- the wireless network is time-synchronised,
- the network topology is formed by the choice of the routing protocol,
- the information on the next hop is acquired from the routing protocol, and
- all the nodes have shared the white-space models with their neighbours.

Operation. Each sensor node in the wireless network computes two sets of model parameters to be used during peak and off-peak noise situations as explained in Section 5.2. The white-space model of each node can estimate noise and predict white-spaces at its own location using one of the parameter sets. Note that the white-space prediction generates a list of FREE slots, i.e., receive slots, in which less noise is expected. The models use random number generators to estimate noise and produce predictions. In order to generate the same prediction list, the models in both the transmitter and the receiver use the same seed which is set during the initialisation phase. The performance of LUCID may be further improved if the predictions could be done on a sub-slot basis. Because the granularity of the noise measurements required is high for training the models and to achieve such a goal, the current version of LUCID does white-space predictions on a slot basis (not on a sub-slot basis).

A sensor node maintains three radio communication states: receive-active, transmit-active and sleep. When the node is in the former state, it turns on its radio to receive transmissions, while in the transmit-active state, the node turns on its radio to transmit data. During the sleep state the node switches off the radio to save energy. The receivers use the predictions from their white-space models for scheduling their own receive-active/sleep states. In other words, the receivers switch their radio on during these-free receive slots for receiving data from neighbours. The white-space model of a receiver might predict a large number of-free receive slots within the current data period, information on which is available at the MAC layer as it is shared by the application layer (cross-layer design). Irrespective of the number of-free receive slots predicted, only a limited number of slots, which is a configurable parameter, is utilised for communication.
This approach can drastically reduce the network-wide energy consumption while keeping the reliability of the data communication high. However, with the growing network size, the number of receive slots used by a node must be increased as well to keep up with the communication reliability. Therefore, the number of utilisable free receive slots, $N_{\text{slot}}$, within a data period, $T_{\text{data}}$, is dependent on the network size and particularly on the maximum sub-tree size for forwarding, thus it is a configurable parameter. The bigger the network size, the higher the number of free receive slots required especially at forwarding nodes and the higher $N_{\text{slot}}$ becomes.

Because each sensor node in the network has access to its receiver’s white-space model, transmitter knows when its receiver is listening for communications. Note that all the sensor nodes in the network, except the coordinator, have their next hop’s white-space models. The individual node is aware of the time slots in which the next hop is ready to receive, thus a rendezvous points can be found for the data communication. As the receiver notifies the sender when it is listening on the channel indirectly via the white-space model, the communication is considered to be receiver-aware.

Figure 5.6 illustrates the proposed model-based receiver-aware communication concept with a 4-node wireless network. According to the underlying topology, node 1 serves as the network coordinator, node 2 is a forwarder node, and node 3 and 4 are end nodes. The sensor nodes go into their receive-active state based on the predictions by their white-space models for receiving packets. For example,
node 1 has receive-active states in slots 1 and 3, relatively from the beginning of the data period. The very first slot of the data period (broadcast slot) is dedicated for disseminating control packets, such as time synchronisation and model re-parametrisation packets. At the beginning of the $n^{th}$ data period, as depicted in Figure 5.6, all the nodes except the network coordinator have a data packet ready to be transmitted toward the coordinator. Therefore, to send the data packet, the individual node checks for the nearest FREE slots at its next hop in which it is in the receive-active state to receive transmissions, which is done by requesting FREE receive slots from the next hop’s white-space model. The nodes transmit/forward their packets within the next hop’s FREE receive slots in a round-robin fashion. This is for the other neighbouring nodes to refrain from simultaneous data transmissions, which otherwise leads to collisions and thereupon packet losses.

The round-robin transmissions within a FREE slot are performed as follows. A FREE slot is further divided into $T_{ss}$ (= 8.512 ms) sub-slots, which is the same value used for the $TH_{IAT}$ when characterising noise in Section 3.4. The length of the sub-slot is devised from the fact that an IEEE 802.15.4 data frame, with its maximum size of 133 bytes, and its acknowledgement spend 8.512 ms on-air at a data rate of 250 kbps for successful transmission. Because LUCID doesn’t use ACKs in its current design, the whole sub-slot is used for transmitting/forwarding multiple data packets. A node uses its identifier to choose a sub-slot within a FREE slot. The transmission sub-slot of a node within a slot, $t_{ss}^{TX}$, is expressed in Equation 5.3, wherein $t_{start}$, $n$, $N_{ss}$, and $T_{ss}$ are the start of time of the slot, the node identifier, the number of sub-slots, and the length of a sub-slot, respectively.

$$t_{ss}^{TX} = t_{start} + (n \mod N_{ss}) \times T_{ss}$$ (5.3)

In wireless communication, collisions can happen when two nodes are forwarding packets to the same node and are outside each of their communication range (hidden terminal problem). The main reason for the hidden terminals is that they cannot coordinate their transmissions. This problem is solved itself in LUCID as it uses round-robin transmissions with allocated sub-slots wherein each node utilises a different sub-slot in the neighbourhood and no coordination is required. Thus, the time slotted receiver-aware communication in LUCID solves the hidden terminal problem effectively if there are sufficient amount of sub-slots to avoid sub-slot collisions with neighbours as explained toward the end of this section.
5.3 Model-based Data Communication

When the noise is high, there could be cases where the white-space model is unable to predict any FREE receive slot. On such occasions, the node utilises the next $N_{slot}$ slots following the first time slot of the data period irrespective of their noise conditions. $N_{slot}$ is a static parameter, and more research is needed to design a technique to adjust it dynamically with the changing network size. LUCID expects packet losses when all the nodes receive high noise and there are no FREE receive slots predicted. There is no countermeasure to tackle this problem in the current version of LUCID.

LUCID tackles the growing requirement of FREE receive slots with the increasing network size by allowing a node to forward as many packets as possible within its transmit sub-slot. This is depicted in Figure 5.6 wherein the node 2 forwards two packets within its transmit sub-slot. Moreover, if it is the last FREE slot of the data period, the node starts utilising its transmit sub-slot and forwards packets until the transmit queue is empty. Transmissions might as well proceed into the following sub-slots.

LUCID does not utilise all the available sub-slots (see Figure 5.6). This is a limitation of the current version of LUCID and more research is needed to alleviate it. A solution to utilise the unused sub-slots would help to further improve the energy efficiency of the nodes.

Curtailing the slot length leads to a low number of sub-slots, which ultimately will increase the probability of concurrent transmissions by collocated sensor nodes (leading to collisions), as explained in Section 5.4.3.2. Collisions due to overlapping sub-slots will eventually increase the number of packet losses in the low-power wireless network if $N_{ss}$ is progressively reduced. Therefore, the slot length should be chosen wisely (i.e., number of nodes $\geq N_{ss}$) especially to avoid systematic collisions. This is a limitation and there is no countermeasure to tackle it in the current version of LUCID.

5.3.4 PDR Monitoring and Feedback Loop

Radio frequency noise is dynamic in a given environment and its properties could change at any time. This nature of noise vastly affects the performance of LUCID. To accommodate the changes in the radio environment, the PDR based feedback loop is used. As depicted in Figure 5.3, the PDR of the data communication application is continuously monitored, and when there is a decline in the value a decision is made whether or not to change the parameter set of the white-space model.
While the application is running in the network, the network coordinator computes the network-wide PDR as expressed in Equation 5.4, wherein $N_{RX}$ and $N_{total}$ are the number of data packets received at the network coordinator and the total number of packets supposed to be received within the data period, respectively. Because LUCID is designed for periodic traffic and the network coordinator has the overall picture of the wireless network, it has the information on $N_{total}$. The PDR is computed periodically at the end of each data period with a duration of $T_{data}$. Note that the value of $T_{data}$ is an application layer parameter that is also available at the MAC layer due to the cross-layer approach of the design.

$$PDR = \frac{N_{RX}}{N_{total}} \times 100\%$$

(5.4)

While computing the $PDR$, the network coordinator also keeps track of the moving average of the $PDR$. The moving average smooths the $PDR$ by eliminating its sudden fluctuations. There are many techniques available to compute the moving average, and to quickly adapt to the changes in the radio environment, the Exponential Moving Average ($EMA$) was used [26]. Equation 5.5 shows how the $EMA$ is computed at the network coordinator, wherein $\alpha$ is the smoothing factor and $EMA_{last}$ is the $EMA$ of the previous data period. By the definition of $EMA$, the smoothing factor is $\alpha = 2/(N_{window} + 1)$, where $N_{window}$ is the length of the moving window. $EMA$ is initialised with the average $PDR$ of the first window. In this work, $N_{window}$ is heuristically determined and its value of 40 gave the most consistent results.

$$EMA = \alpha \times PDR + (1 - \alpha) \times EMA_{last}$$

(5.5)

The $EMA$ of the network-wide $PDR$ that is computed at the network coordinator is a measure of the performance of the proposed solution. When the performance drops, the white-space model should be re-parametrised. The decision on when to trigger the re-parametrisation command is based on a predefined threshold, $TH_{PDR}$, which is an end-user customised parameter. The value of $TH_{PDR}$ can be sought as the minimum communication reliability expected by the end-user. It is an indication of the degradation of the performance when the $EMA$ goes below the threshold. However, the re-parametrisation should not be initiated spontaneously when the $EMA$ crossed the threshold. The reason is that the change in the noise could be momentary, thus the re-parametrisised models would not help to overcome the performance loss after the temporal noise variation. Therefore,
a transient period called \textit{re-parametrisation timeout} is applied, which delays the model re-parametrisation and allows the impulse of noise to fade away and thereby to settle the \textit{EMA} down. If the noise is persistent even after the timeout, the model re-parametrisation is triggered. This technique alleviates the instability that emerges with rapid model re-parametrisations.

Before flooding the network with the control command to re-parametrise the white-space models, the network coordinator switches its parameter set from peak to off-peak and vice versa for the coming data period onward. A re-parametrisation control packet (see Appendix D.1) consists of the age and the type of model that is going to be used next, i.e., peak or off-peak. Similar to the authoritative field in a time synchronisation packet, see Section 5.3.1.1, a node uses the age field to avoid loops in the dissemination of the control packets. The coordinator initiates the flooding with zero-age at the beginning of the next data period when all the nodes are listening for an update in the network; the receiving nodes increase the age of the packet by 1 before rebroadcasting it. The age field also serves as a means of acknowledgement. A node deems the received re-parametrisation packet as an acknowledgement when the packet’s age is 1 higher than that is stored in the node itself. To increase the probability of reception of the re-parametrisation command, the network coordinator broadcasts a maximum of five re-parametrisation packets. Switching from one parameter set to another takes a negligible amount of time. Nonetheless, all the nodes in the network start using predictions from the new parameter set from the coming data period onward. The current data period in which re-parametrisation was triggered is not affected by the new parameter set as otherwise it will create a complex situation when rescheduling when to wake up. Note that all the nodes need to obtain a list of new predictions, which will be gathered during the current data period, so the wake-up schedule can be updated accordingly.

\section*{5.4 Performance of LUCID}

This section presents the performance of LUCID for various RF noise conditions and compares LUCID with the state-of-the-art CRYSTAL \cite{162, 78, 77}, and ContikiMAC \cite{45} in collaboration with CSMA protocols. CRYSTAL is a synchronous transmission protocol that uses channel hopping and noise detection techniques to mitigate interference in low-power wireless networks, delivering high performance. By comparison, CSMA/ContikiMAC is an asynchronous protocol available in the
Contiki protocol stack (v2.7), which uses a power-efficient wake-up mechanism to minimise the energy consumption and re-transmissions to improve the transmission reliability. These two state-of-the-art solutions pave the way to compare the performance of LUCID against two different channel access paradigms. For practical reasons, the evaluation was done in COOJA in collaboration with MATLAB which provides rich libraries for multi-variant GMMs and HMMs, as described in Section 5.4.1.

5.4.1 Proof of Concept Implementation

Previously in this chapter, the operation of LUCID was discussed. This section presents how the solution was implemented, the practical issues encountered during the implementation, and the countermeasures undertaken to overcome them.

To investigate its performance, LUCID was implemented in Contiki/COOJA. Contiki is an open-source operating system tailored to low-power wireless sensor nodes, such as T-Mote Sky [139], while COOJA is a network simulator especially designed for the Contiki operating system, which provides a platform to simulate cyber-physical system applications [131, 136]. In this work, Contiki 2.7 was used to validate the proposed proof of concept.

The implementation of LUCID in Contiki is presented in Figure 5.7, which depicts the flow of the different phases of the proposed solution. As described in Section 5.2 and depicted in Figure 5.7, the outcome of the deployment phase is a white-space model with two parameter sets, which are shared among neighbour nodes during the network initialisation and used for the white-space prediction during the operation of the network. Note that the transitions from deployment and network initialisation phases (see Figure 5.3) are based on timeouts.

Furthermore, the proposed technique to mitigate the effects of RF noise is a cross-layer approach as depicted in Figure 5.8. A big portion of the proposed solution is based in the MAC layer, w.r.t. the IEEE 802.15.4 communication protocol stack, while application layer and network layer provide essential inputs, which includes EMA, the data period and when it starts, and information on the next hop. These inputs are prerequisites for the operation of LUCID.

As described in Section 5.3.3 and illustrated in Figure 5.7, a sensor node accesses the radio channel based on a schedule that is built locally from the predictions made from the receiver’s white-space model. Note that at the beginning of each data period and prior to every single transmission, the node gathers predictions for the current data period from the receiver’s white-space model, which is available
5.4 Performance of LUCID

Legend

TST: time sync timer
WT: wakeup timer
ST: sleep timer
PDRT: compute PDR timer
RT: re-parametrisation timer
TXT: transmit timer
EMA: exponential moving average
THR: threshold
RX: receiver
BC: broadcast

Fig. 5.7 Flowchart of the implementation of LUCID in Contiki/COOJA.
in the node itself as it was shared by the receiver during the initialisation phase. By using the time (i.e., slot ID), at which the data period starts, as the input, the white-space model is able to acquire the most recent channel states in the prediction process. This task requires a substantial amount of computation power to do predictions at run-time by the white-space model. For example, a Dell XPS 15 (9570) computer running 64-bit Windows 10 with 6 Core i7 – 8750H CPUs (2.20 GHz), a 16 GB RAM, and a GTX 1050 Ti Max-Q (4 GB) GPU takes \( \approx 40 \) milliseconds to compute a white-space. As mentioned in Section 5.3.3, a sensor node needs to keep a set of white-space models of its neighbours to efficiently combat noise while facing topology changes. Storing all the white-space models (each model parameter set with 834 bytes) of all the neighbours increases the memory footprint of the sensor node. A resource-constrained node like TMote Sky is unable to facilitate such a degree of computation as well as the memory requirement.

Therefore, to address the high computation power problem, the nodes have to outsource the prediction-based computations to an external entity. As the memory

---

**Fig. 5.8 Protocol stack.**

**Fig. 5.9 System integration with COOJA and MATLAB.**
and computation requirements exceed the capacity of the TMote Sky sensor node that is been considered in this work, the evaluation of LUCID was done in COOJA and the external computations were performed in MATLAB [115], as depicted in Figure 5.9. In the following text, the term “shared processor” is used to indicate the MATLAB implementation. A node requests white-space predictions from the shared processor through commands sent via the in-simulator connection with a baud rate of 115200 bps. The command includes the node identifier, a flag that carries the information regarding the change of the set of model parameters, and the first slot ID of each data period for which the predictions are requested (see Appendix E). Upon receiving the command, the shared processor does prediction-related computing and conveys the result back to the node simulated in COOJA. The conveyed result includes the first slot ID of the data period, whether the predictions are for transmit/receive slots, and the relative slot IDs of the predicted free slots from the start of the data period.

However, the delay incurred during this in-simulator communication mechanism is too high to achieve outsourcing in a real-time scenario wherein COOJA is simulating the wireless network real-time (i.e., the simulation time is equal to the wall clock time). Therefore, to eliminate the high delay, a sensor node keeps a list of pre-computed predictions that will be utilised by the node. The list of predictions is computed in advance in the shared processor and updated through the in-simulator link at run-time (see Figure 4.2). The pre-computed predictions are saved in a data structure in accordance with the node ID, the type of model parameter set used for prediction, and the first slot ID of the data period. This pre-computing enables the sensor nodes to have white-space predictions when they are needed without additional delays.

It is important to note that the countermeasures taken to address the high computation power requirements are only for validating the proof of concept. In practice, such a solution with computations being outsourced might be feasible with techniques available in the edge computing paradigm, which is a distributed information technology architecture wherein capturing, storing, processing, and analysing data is done as close to the originating source of data as possible [102]. Using edge computing technology, it is possible to perform calculations in a more powerful node and share the predictions with the nodes in the network. However, it would require a communication mechanism between the nodes in the network with the powerful node to gather predictions. This additional communication overhead would cost approximately $9.72 \times 10^{-4}$ As energy per node per data period.
5.4 Performance of LUCID

Fig. 5.10 Simulation scenarios with two RF noise generators (marked in red).

(assuming 10 seconds data period) with 50 milliseconds slots and 39 bytes MAC control packets. Alternatively, newer sensor nodes are equipped with modern processors, such as the ARM Cortex-M0 32-bit processor, which are powerful enough (in comparison with a slow 16-bit micro controller and limited RAM of TMoteSky nodes) to do required prediction related computations.

To overcome the high memory requirement issue due to topology changes, a static tree topology was used throughout the evaluation. To this end, the de-facto tree-based routing protocol in WSNs, the Collection Tree Protocol (CTP) [61], was used. The CTP discovers the neighbours within the communication range of the sensor nodes and assigns a parent to a sensor node based on the link qualities, which helps the network to create a tree topology with parents, children, and leaf nodes. LUCID acquires the information about the identity of the parent node from the routing protocol. This piece of information is vital for the receiver-aware communication as explained in Section 5.3.3. The source code of the implementation is presented in Appendix B.

5.4.2 Simulation Configurations and Execution

For the evaluation, two wireless sensor networks that consist of 5 and 16 nodes were considered and studied in varying noise conditions. The choice of the networks is based on the practicality of the simulations especially due to the outsourcing of computing, as illustrated in Section 5.4.1, and memory leaks while running simulations. The simulation results can be scaled to larger networks as well ($N_{\text{slot}}$.
should be adjusted). Nonetheless, more research is needed to study the behaviour of forwarding nodes near the network coordinator and collisions due to overlapping sub-slots with the increasing network size.

Figure 5.10 depicts how the two scenarios were deployed in COOJA, with a distance between the neighbouring sensor nodes of 20 m and 0 dBm transmit power. Note that the nodes in the small network (see Figure 5.10a) were arranged such that it comprises of three forwarding layers for examining the solution in a multi-hop network setting. Moreover, the 16-node network, depicted in Figure 5.10b, was deployed in a grid with a 20 m grid size. It is noteworthy that there will be bottlenecks at nodes that are closer to the network coordinator, such as node 9 in Figure 5.11. These nodes will end up forwarding many more packets in comparison with the nodes further away from the coordinator. This is a fundamental problem for all converge-cast type multi-hop communication scenarios. The situation is exacerbated with the increasing network size, leading to declining performance of the underlying protocol which schedules transmissions in a resource-constrained nature. The role of node 9 in Figure 5.10b is to study the behaviour of the forwarding nodes near the network coordinator.

Furthermore, to replicate the real-life noise conditions, two jammers configured with JamLab [23] were used. JamLab is a tool to record and playback noise patterns as well as to generate customizable and repeatable noise in real-time. In the COOJA simulation, two nodes running JamLab generates repeatable noise patterns according to a TBNS distribution which is extracted from the real-life 1-hour noise measurements, presented in Section 3.3. The two jammer nodes simulate the noise sources by emitting signals according to the generated patterns on the operating IEEE 802.15.4 channels. It has many parameters controlling the behaviour of the jammers: transmit power, duration of the signal, and time between two signals. The former is set to the maximum transmit power of 0 dBm. To simulate Wi-Fi ACK and data packets, the duration of noise signals oscillates between small and large durations, which are JamLab’s default configurations. The time between two noise signals is based on the probability distributions of the measured real-life noise traces in the OFFICE and HOME environments during peak and off-peak noise conditions, which also include emissions from microwave ovens as observed by the author (the LED of the sensor node starts blinking rapidly when the microwave oven is turned on). More information on JamLab settings is available in Appendix C. The nodes 6 and 7 in Figure 5.10a and the nodes 17
and 18 in Figure 5.10b identify the location of simulated noise sources. Note that determined by its position, a node receives noise from a single or both jammers.

In low-power wireless networks, the CCA process plays a vital part in identifying signals/noise on the operating channel. CCA in the transceiver of a sensor node detects the presence of a signal when the energy level of the operating channel is higher than the CCA threshold, $TH_{CCA}$. As discussed in Section 3.3.2, when measuring noise from unknown sources, the CCA threshold was set to $-77$ dBm, and the same value was used throughout the evaluation of LUCID, ContikiMAC and CRYSTAL.

It is common that wireless sensor nodes generate periodic traffic [87] and LUCID is designed for periodic traffic. To evaluate the performance of LUCID, two data generation periods were considered, i.e., $T_{data} = \{10, 60\}$ seconds. These data generation periods helped to investigate the performance of both in high [80] and low [10] data rate applications. Because IEEE 802.15.4 channel 18 is overlapping with different Wi-Fi channels that were busy during the noise measurements in both the HOME and the OFFICE and the collected traces are used to generate noise in the simulator, IEEE 802.15.4 channel 18 is selected for all communication during the simulations.

**LUCID.** Table 5.1 presents a list of parameters and their values, which were used when simulating LUCID in COOJA. $TH_{IAT}$, $TH_{count}$, and $TH_{CCA}$ are used in the deployment phase when measuring, characterising, and deriving model parameters, while the remaining parameters were used during the execution of the data communication. Note that the parameters, $T_{sync}$, $N_{window}$, $slot_{len}$, and $T_{reparam}$, used in the evaluation are empirically determined after a heuristic analysis on the subject, that will be discussed in the following subsections. In real-world scenarios, finding optimal values for each parameter mentioned in every deployment would be challenging. If the end-user selected non-optimal values for parameters, especially for $slot_{len}$ and $T_{reparam}$, it would lead to instability of the MAC protocol hence under-performance of LUCID.

As mentioned in Section 5.3.1.1, the value of $T_{sync}$ is a trade-off between the network-wide tight synchronisation and the lifetime of the low-power wireless network. Therefore, to balance both sides, intuitively the value of the period was chosen to be 5 minutes, as the synchronisation error wouldn’t be significant [167]. After 5 minutes, the synchronisation error is in the sub-milliseconds range at an energy cost of $\approx 1.26 \times 10^{-3}$ As per node with 50 milliseconds slots and 39 bytes MAC control packets (see Appendix D.1).
5.4 Performance of LUCID

Table 5.1 Configurations of LUCID.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( TH_{IAT} )</td>
<td>8.512 ms</td>
<td>mean TBNS threshold</td>
</tr>
<tr>
<td>( TH_{count} )</td>
<td>11</td>
<td>threshold for noise signal count</td>
</tr>
<tr>
<td>( TH_{CCA} )</td>
<td>–77 dBm</td>
<td>CCA threshold</td>
</tr>
<tr>
<td>( T_{sync} )</td>
<td>5 minutes</td>
<td>time synchronisation period</td>
</tr>
<tr>
<td>( N_{\text{slot}} )</td>
<td>{2, 3}</td>
<td>number of slots per data period</td>
</tr>
<tr>
<td>( slot_{\text{len}} )</td>
<td>50 ms</td>
<td>transmit/receive slot length</td>
</tr>
<tr>
<td>( Q_{\text{len}} )</td>
<td>16</td>
<td>forwarding queue size</td>
</tr>
<tr>
<td>( T_{\text{reparam}} )</td>
<td>5 periods</td>
<td>re-parametrisation timeout</td>
</tr>
<tr>
<td>( TH_{PDR} )</td>
<td>93%</td>
<td>PDR threshold for re-parametrisation</td>
</tr>
<tr>
<td>( N_{\text{window}} )</td>
<td>40 samples</td>
<td>moving window size for ( EMA ) calculations</td>
</tr>
<tr>
<td>background noise</td>
<td>–90 dBm</td>
<td>background noise level (thermal noise)</td>
</tr>
<tr>
<td>transmit power</td>
<td>0 dBm</td>
<td>packet transmit power</td>
</tr>
<tr>
<td>PHY model</td>
<td>MRM</td>
<td>PHY propagation model</td>
</tr>
<tr>
<td>JamLab parameters</td>
<td>refer Appendix C</td>
<td>JamLab parameters</td>
</tr>
<tr>
<td>standard</td>
<td>2011</td>
<td>IEEE 802.15.4 standard used</td>
</tr>
</tbody>
</table>

\( TH_{PDR} \) is an application-dependent parameter, thus the value of which is specified by the end-user. However, care should be taken when selecting its value as high values of \( TH_{PDR} \) might induce instability to the performance of the wireless network, as with such values the re-parametrisation process might get triggered more often. Therefore, in the evaluation, after a heuristic evaluation, 93\% was used as the \( TH_{PDR} \).

The length of the moving window when computing \( EMA \) (\( N_{\text{window}} \)) was selected to be 40 samples. This value was heuristically obtained as follows. The \( EMA \) was calculated as the window size was varied from 10 to 100 in steps of 10 and the impact on the stability of the system was investigated in terms of the number of cross-overs around the threshold, \( TH_{PDR} \). Note that long windows cannot capture short-term fluctuations in the \( PDR \) and small windows are too sensitive to short-term variations, leading to an unstable system. Based on this evaluation it was found that a value of 40 for \( N_{\text{window}} \) provided the best performance.

The channel model used in the simulation is Multipath Ray-tracer Medium (MRM), which uses 2D ray-tracing approach to approximate the receive power and capable of computing reflections and refractions along the radio links [136].

CRYSTAL. CRYSTAL uses two unique techniques to mitigate the effects of RF noise: channel hopping and noise detection. The former tackles the effects of noise by escaping it, i.e., switching to a channel that experiences less noise than the current one. The latter fights the noise by changing its termination criterion (i.e.,
CRYSTAL ignores triggers of consecutive silent pairs due to heavy noise) which allows to circumvent noise and renders more opportunities for communications by keeping the network awake until the noise fades away [162]. To compare the ability of LUCID and CRYSTAL to fight noise without escaping it, CRYSTAL was run without channel hopping enabled, thus only IEEE 802.15.4 channel 18 was permitted.

**ContikiMAC.** CSMA is a well-known MAC protocol that has been widely used in wireless communications. Nonetheless, CSMA does not control the duty-cycling which is crucial for low-power wireless networks to save energy. Contiki provides different Radio Duty-Cycling (RDC) protocols, such as Low-Power Probing (LPP) [123], X-MAC [28], and ContikiMAC [45]. The latter is the default RDC protocol in Contiki which delivers the highest performance among them in terms of energy savings. Thus, in the performance comparison, ContikiMAC in collaboration with CSMA was used. Moreover, the CSMA protocol exploits re-transmissions as a countermeasure to tackle interference. Therefore, the comparison of the performance was done with/without re-transmissions. When re-transmissions were enabled, intuitively, the number of maximum transmissions was set to be 3. Increasing the number of re-transmissions will increase the reliability at the cost of energy consumption [81].

**Simulation Execution.** The simulation was conducted based on the scenarios depicted in Figure 5.10. In both scenarios, the noise was injected into the simulation environment with 16 distinct configurations, as shown in Table 5.2. These noise settings reproduce the environments where the noise measurements were taken, see Section 3.3, and their noise patterns, as illustrated in Section 3.4.3. Moreover, the data collection application with 56 bytes packets was run for 2 hours for all simulation configurations.

Because the noise was injected with the probability distributions of the previously collected noise traces, and the white-space models were already parametrised,

### Table 5.2 Simulation execution.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Noise Type</th>
<th>Network Size (#nodes)</th>
<th>Data Periods (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>office</td>
<td>peak</td>
<td>{5, 16}</td>
<td>{10, 60}</td>
</tr>
<tr>
<td>home</td>
<td>off-peak</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>peak</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>off-peak</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
the simulation did not execute the deployment phase. The white-space models of all the nodes can be accessed by the shared processor where the outsourced computation is done.

During the network initialisation phase, the nodes are time-synchronised and the routing topologies are formed. Figure 5.11 depicts the snapshots of the tree topologies that the two sensor networks, shown in Figure 5.10, set up. The same topologies are used in each simulation run. Both routing topologies comprise of 3 forwarding levels, which is beneficial for evaluating the performance of LUCID in a multi-hop setting.

While running, the sensor nodes dump logs that consist of all the information related to the operation of the communication. This includes PDR, EMA, and duty-cycle related data, which are extracted and processed after the simulations have finished. The data uncovered during the post-processing is then used for evaluating the performance.

In this work, duty-cycle is computed based on the total radio on time of a node including that of re-transmissions. A node turns on its radio for idle listening, receiving, and transmitting data. Maximum transmit power of 0 dBm is used and the power levels are assumed to be the same in each radio on case (i.e., transmit power $\approx$ receive power $\approx$ idle listening power).

Furthermore, to have statistical significance in the results, each simulation was conducted 3 times with varying seeds and the average results were used in the evaluation. The number of simulation runs was limited by pragmatic reasons, such

![Network Topologies](image.png)

Fig. 5.11 Network topologies.
as having to run the same simulation multiple times (even more than 20 in some cases) due to memory leaks.

### 5.4.3 Performance Evaluation

Evaluation of LUCID is split into three parts. First, the importance of model re-parametrisation and its timeout is investigated. In this regard, the 5-node network with two interferers, shown in Figure 5.10a, is used with varying re-parametrisation timeouts ($T_{reparam}$). Second, the impact of the slot length on the performance of LUCID is studied. For this, the re-parametrisation enabled 5-node network is used with different slot lengths. Third, LUCID is compared with the two synchronous and asynchronous medium access methods respectively: CRystal [162] and ContikiMAC [45]. For comparing noise combat performances (not by escaping it), the channel hopping mechanism in the former is disabled, as discussed in Section 5.4.2. The latter is compared with/without re-transmissions. The comparison is done with the low-power wireless networks of 5 and 16 nodes mentioned earlier as depicted in Figure 5.10.

The performance metrics used in the evaluation of the communication reliability are the $PDR$ and its exponential moving average, $EMA$. The performance of the network is further analysed with the average $duty-cycle$ and the energy efficiency, as defined in Equation 5.6, of the sensor nodes.

$$\text{energy efficiency} \approx \frac{PDR}{duty-cycle} \quad (5.6)$$

All nodes except the network coordinator generate data at the beginning of each data period. Data generation is periodic and is synchronised among all the nodes in the network. These packets are transmitted based on the predicted receive slots at the receiver. In other words, all nodes that generated a data packet do not try to transmit them at the same time. These transmitted packets can be lost due to one of many reasons as follows:

- thermal and non-CTI noise,
- congestion loss due to buffer overflows,
- IEEE 802.15.4 noise (due to overlapping sub-slots by many nodes, clock drifts, and coexisting other WSNs), and
- CTI.
5.4 Performance of LUCID

Table 5.3 EMA statistics without model re-parametrisation.

<table>
<thead>
<tr>
<th>EMA Statistic</th>
<th>10 s home</th>
<th>10 s office</th>
<th>60 s home</th>
<th>60 s office</th>
</tr>
</thead>
<tbody>
<tr>
<td>min. (%)</td>
<td>82.20</td>
<td>88.88</td>
<td>90.48</td>
<td>93.57</td>
</tr>
<tr>
<td>max. (%)</td>
<td>92.48</td>
<td>96.60</td>
<td>96.62</td>
<td>98.63</td>
</tr>
<tr>
<td>avg. (%)</td>
<td>88.04</td>
<td>93.47</td>
<td>94.27</td>
<td>96.54</td>
</tr>
<tr>
<td>std. (%)</td>
<td>1.82</td>
<td>1.36</td>
<td>1.19</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td>90.50</td>
<td>94.18</td>
<td>94.13</td>
<td>94.89</td>
</tr>
<tr>
<td></td>
<td>95.33</td>
<td>97.56</td>
<td>97.64</td>
<td>98.80</td>
</tr>
<tr>
<td></td>
<td>93.16</td>
<td>96.19</td>
<td>95.94</td>
<td>97.08</td>
</tr>
<tr>
<td></td>
<td>1.24</td>
<td>0.82</td>
<td>0.95</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Because the background noise level is $-90$ dBm while transmit power is $0$ dBm and there are no non-CTI noise sources, packet losses due to thermal and non-CTI noise can be safely ignored. The buffer length is 16, thus considering the network sizes used, the congestion loss can also be disregarded. There is no other coexisting WSNs in the simulations. The clocks are near perfect synchronised due to periodic time synchronisation (drift is always under the sub-milliseconds range). Nonetheless, there could be collisions when two nodes try to transmit packets to a third node utilising the same sub-slot, which will lead to packet losses. These collisions diminish communication performance, especially at forwarding nodes. Therefore, care must be taken when selecting the slot length and the number of receive slots (considering the network size and the maximum sub-tree size) to avoid such overlapping sub-slots.

The radio propagation model in COOJA is capable of computing path loss, reflections, and refractions along the radio links. COOJA uses a normal cumulative distribution function, which is based upon algorithm $5666$ for the error function $[66]$, to find the packet reception probability by taking the mean and the standard deviation of Signal to Noise Ratio (SNR) into account. COOJA stops packet reception and marks the packet as interfered when the reception probability is low, which might have been caused even by a burst of high-power Wi-Fi ACKs generated by JamLab with small signal durations.

5.4.3.1 Model Re-parametrisation

Section 5.3 advocated the use of a feedback loop for tackling sudden RF noise variations in the radio environment where the low-power wireless network was deployed. Therefore, first, the necessity of a feedback loop in the wireless network is investigated.

Table 5.3 presents the performance of LUCID without using the PDR feedback loop. With this setting, when there is a short-term change in the radio environment, the functionality of the white-space model re-parametrisation is disabled. As can be
Table 5.4 EMA statistics with model re-parametrisation.

<table>
<thead>
<tr>
<th>Timeout ($T_{data}$)</th>
<th>EMA statistic</th>
<th>10 s</th>
<th>60 s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>home</td>
<td>office</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>Off-peak</td>
</tr>
<tr>
<td>3</td>
<td>min. (%)</td>
<td>84.12</td>
<td>91.55</td>
</tr>
<tr>
<td></td>
<td>max. (%)</td>
<td>95.58</td>
<td>96.99</td>
</tr>
<tr>
<td></td>
<td>avg. (%)</td>
<td>90.74</td>
<td>94.72</td>
</tr>
<tr>
<td></td>
<td>std. (%)</td>
<td>2.26</td>
<td>1.02</td>
</tr>
<tr>
<td>4</td>
<td>min. (%)</td>
<td>85.73</td>
<td>91.11</td>
</tr>
<tr>
<td></td>
<td>max. (%)</td>
<td>96.76</td>
<td>97.01</td>
</tr>
<tr>
<td></td>
<td>avg. (%)</td>
<td>93.02</td>
<td>94.81</td>
</tr>
<tr>
<td></td>
<td>std. (%)</td>
<td>1.71</td>
<td>1.08</td>
</tr>
<tr>
<td>5</td>
<td>min. (%)</td>
<td>85.73</td>
<td>91.90</td>
</tr>
<tr>
<td></td>
<td>max. (%)</td>
<td>96.87</td>
<td>97.42</td>
</tr>
<tr>
<td></td>
<td>avg. (%)</td>
<td>93.76</td>
<td>94.72</td>
</tr>
<tr>
<td></td>
<td>std. (%)</td>
<td>1.49</td>
<td>1.02</td>
</tr>
<tr>
<td>6</td>
<td>min. (%)</td>
<td>85.73</td>
<td>91.26</td>
</tr>
<tr>
<td></td>
<td>max. (%)</td>
<td>97.82</td>
<td>98.02</td>
</tr>
<tr>
<td></td>
<td>avg. (%)</td>
<td>94.50</td>
<td>94.93</td>
</tr>
<tr>
<td></td>
<td>std. (%)</td>
<td>1.52</td>
<td>1.39</td>
</tr>
</tbody>
</table>

seen in EMA statistics, when the data period $T_{data}$ increases, all the EMA statistics tend to increase their values for all the noise settings in both environments. The long $T_{data}$ allows sufficient time for the quick burst of noise in the radio environment to fade away. One could also argue that with the long $T_{data}$, there are more chances to find free receive slots that minimise the probability of collisions. These two effects link together to deliver higher performance in comparison with small $T_{data}$.

Moreover, LUCID delivers higher performance in the OFFICE in comparison with the HOME in terms of average EMA. Furthermore, the prediction performance of the white-space models is always higher during off-peak noise compared to peak noise periods. The EMA is always above the $TH_{PDR}$ of 93% in all the environments and noise settings combinations except during the peak noise in the HOME, where it delivers 88.04% average EMA with the highest standard deviation of 1.82%.

From the results in Table 5.3, one can conclude that higher data rate applications running in heavy bursty noise settings deliver poor performance in the solution due to more frequent sudden changes in the radio environment than that of low data rate applications and mild noise. The white-space models should be able to adapt to the dynamic nature of the RF noise, which is the motivation behind the model re-parametrisation with the PDR feedback loop.

As elaborated in Section 5.3.4, the model re-parametrisation is not triggered as soon as the EMA falls below the $TH_{PDR}$ by the network coordinator. Instead, the network coordinator allows a transition period, re-parametrisation timeout ($T_{reparam}$), which is measured in terms of data periods, $T_{data}$. Because the small
timeout introduced more instability to the communication system causing rapid model re-parametrisation, $T_{reparam}$ was varied from 3 to 6 data periods in steps of 1 to evaluate the influence of the timeout and re-parametrisation.

The results are shown in Table 5.4. The inclusion of the model re-parametrisation helped the average EMA to exceed the $TH_{PDR}$ for all noise settings and data period combinations after introducing 4 as the re-parametrisation timeout. With this timeout, the EMA has increased by 5% and 1.3% in the HOME with 10 seconds data period, respectively in peak and off-peak noise settings. Moreover, the increment of EMA in the OFFICE with peak noise and 10 seconds data period is 1%. Because of the non-bursty noise or low data rate, all the other combinations showcase negligible changes in average EMA.

Note that the average EMA further increases with the increasing $T_{reparam}$, and its standard deviation also decreases and starts to settle down when the timeout is at 5 for all combinations. Therefore, this value of the $T_{reparam}$ is selected and used in the rest of the analysis, which allows sensor nodes sufficient time until the sudden burst of noise weakens.

5.4.3.2 Impact of Slot-length

The number of data transmissions that could occur within a data period is dependent on the size of the wireless network. Consequently, the load of the forwarding sensor nodes which forward data packets to their parents builds up. To cope with this, the duty-cycle of the nodes should be augmented. Therefore, the slot length has a significant impact on the performance of a wireless network.

The number of data packets (56 bytes packets) that can be transmitted within a single slot is determined by its length. The longer the length, the more the number of packets that can be transmitted within a slot. On the contrary, increasing slot length also escalates the energy consumption of the resource-constrained wireless network. With 10 seconds data period and 100 bytes packets, a node with 50 ms and 150 ms slots consumes $\approx 1.0 \times 10^{-3}$ As and $\approx 2.9 \times 10^{-3}$ As of energy per data period, respectively. In other words, a node with 150 ms slots consumes $\approx 190\%$ more energy than that of 50 ms slots. Thus, in the heuristic analysis, going above 150 ms slot length leads to a much higher duty-cycle and greater energy consumption than a resource-constrained sensor node can sustain.

Furthermore, decreasing the slot length also reduces the number of available sub-slots in which data transmission takes place, as discussed in Section 5.3.3, diminishing the medium access for the nodes in the network. Moreover, small
time slots increase collisions by transmitters with overlapping sub-slots. This has a significant negative impact on forwarding nodes wherein multiple data packets need to be transmitted within a slot, leading to packet losses. Because ACKs are not used, 2 or more packets, depending on the packet size, can be forwarded by a node within the same sub-slot. With this regard, 40 ms was identified as the critical slot length below which significant packet losses due to CTI as well as overlapping sub-slots are inevitable even for a small wireless network with 5 sensor nodes.

Therefore, to investigate its effect on the performance, the slot length was varied as follows: \{150, 100, 50, 40\} ms while running LUCID in the 5-node network. As mentioned in Section 3.4.2, the traces of measured noise from unknown sources were characterised with 100 ms slots under the assumption that the value of such a slot length is a trade-off between the energy consumption and the availability of sub-slots for reliable communication. Note that at this point, LUCID was not designed. Here, 100 ms is used as the basis to vary the slot lengths and evaluate the corresponding performances of the network with LUCID. Three simulations are run for each scenario and the average result is used. Figure 5.12 presents the change in the network-wide PDR and duty-cycle with varying slot lengths while keeping the $N_{slot}$ constant at 2.

As Figure 5.12 demonstrates, the network-wide PDR of the proposed solution, by considering packet losses at any stage of the multi-hop network, with 40 ms slot length delivers less than 75% of the packets in all the data periods, environments, and noise settings combinations. In comparison with the performance of the system with 50 ms slot length, the difference in PDR is more than 17.8% in all the cases. Therefore, 40 ms slot length is not acceptable as it degrades the reliability of the communication network significantly.

With 50, 100, and 150 ms slot lengths, the wireless network delivers similar performance, and the differences w.r.t. the 50 ms slot length are less than 3%. Moreover, when the system applies the 50 ms slot length, the duty-cycle is at its minimum, i.e., 0.84%, and 0.16%, respectively in 10 and 60 seconds data periods. This is valid for all the combinations of settings.

Furthermore, the 50 ms slot tops the energy efficiency (according to Equation 5.6) always among the simulation settings considered.

By considering the above results, the 50 ms slot length is best suited as the solution to improve the performance of the wireless network. Therefore, in the following analysis of LUCID, 50 ms will be used as the slot length.
5.4 Performance of LUCID

Fig. 5.12 Change in PDR (top), duty-cycle (middle), and energy-efficiency in terms of PDR/duty-cycle (bottom) of LUCID with varying slot lengths while keeping $N_{\text{slot}}$ at 2. The standard error is also included.
5.4 Performance of LUCID

5.4.3.3 Comparison of LUCID with CRYSTAL and ContikiMAC

Thus far, the performance of LUCID was studied with distinct data periods in varying noise settings. In the following, LUCID’s performance is studied and compared in comparison to two state-of-the-art protocols. The comparison focuses on CRYSTAL [162] and ContikiMAC [45] with the configurations specified in Section 5.4.2. These two choices enable the evaluation of LUCID against two different access paradigms as mentioned in Section 5.4.

5.4.3.3.1 5-node Network

Figure 5.13 depicts the comparison of network-wide PDR, duty-cycle (i.e., energy use), and energy efficiency of the 5-node wireless network for varying noise conditions. Appendix F presents more data on this matter.

**PDR.** It is evident from Figure 5.13 that ContikiMAC without re-transmissions performs the worst in terms of reliability as it produces less than 60% PDR. This is true for all combinations of data periods, environments, and noise conditions. CSMA is a random channel access mechanism, thus it is expected that ContikiMAC in collaboration with CSMA without re-transmissions has poor performance. The reason is that with the small backoff exponent the probability of choosing the same backoff interval by multiple nodes is high, which leads to collisions [56, 118]. However, when the re-transmissions are enabled in the network, ContikiMAC improves its PDR by more than 30% [3]. This proves the effectiveness of re-transmissions implemented in CSMA for interference mitigation.

LUCID, CRYSTAL, and ContikiMAC with re-transmissions produce more than 90% PDR in all cases. Nonetheless, LUCID outperforms ContikiMAC in all cases except in the HOME peak noise with 1.2% and 0.6% PDR improvements, respectively for 10 and 60 seconds data periods. In the collected dataset the HOME peak exhibits high bursty noise, during which LUCID detects performance degradation through PDR monitoring, leading to model re-parametrisations. As illustrated in Section 5.3.2, to avoid frequent re-parametrisations, a transition period is introduced. However, the transition period contributes to a slight decline in PDR w.r.t. ContikiMAC, especially in very bursty noise settings. However, re-transmissions in ContikiMAC perform well in combating bursty noise but at the cost of enormous energy consumption.

The standard error of PDR is less than 0.72% for all the cases in LUCID and ContikiMAC with re-transmissions (see Appendix F). CRYSTAL shows marginally
5.4 Performance of LUCID

Fig. 5.13 Comparison of PDR (top), duty-cycle (middle), and energy-efficiency in terms of PDR/duty-cycle (bottom) in the 5-node network with the SoA. Here, LUCID utilises 2 FREE slots (receive slots) per data period.
higher standard error (≈ 1.3%) in PDR with 60 seconds data period in home environment. Nevertheless, the differences in standard error are not significant between the techniques to make a claim.

The PDR of LUCID, in comparison with CRystal, shows mixed results. In the following text, the reliability of LUCID and CRystal is further studied in the worst-case and the average-case settings. Here, the worst-case is identified when the wireless nodes receive bursty noise (i.e., peak noise conditions in the home environment). The average-case is considered to be the average reliability over all considered noise settings. In the worst-case (with 60 seconds data period) as well as the average-case, the 95% confidence interval of PDR in LUCID and CRystal are:

- worst-case: 92.73 ± 1.42% and 92.66 ± 2.60%, and
- average-case: 94.43 ± 1.28% and 95.37 ± 1.44%, respectively.

Therefore, it is evident that the reliability of LUCID in the small 5-node network is slightly better than that of CRystal in the worst-case, but it is conversely true in the average-case. Nonetheless, the differences in communication reliability of LUCID and CRystal are negligible.

**Energy use.** In terms of duty-cycle in the 5-node network, irrespective of re-transmissions, ContikiMAC shows the worst results exhibiting at least 3.2% and 2.7% of an average duty-cycle, respectively for 10 and 60 seconds data periods. The re-transmissions are an additional overhead, which increases the energy consumption of the wireless network, and is reflected in the results by showing a higher duty-cycle for ContikiMAC with re-transmissions.

CRystal slightly surpasses LUCID in all noise settings, with a maximum difference of 0.3% and 0.1% in duty-cycle, respectively for 10 and 60 seconds data periods in the 5-node network. However, these differences in the duty-cycle, its standard error, and confidence interval are insignificant between the two techniques.

**Energy efficiency.** The ratio between the PDR and duty-cycle is used as an approximation of the energy efficiency of the network. For all the noise settings, CRystal demonstrates its high energy efficiency, while ContikiMAC showcases its energy inefficiency. In the 5-node network, CRystal is more energy-efficient than LUCID by 33.2% and 38.5% in the worst-case and the average-case, respectively. The high energy efficiency difference is mainly due to unused sub-slots in LUCID.
5.4 Performance of LUCID

### 5.4.3.3.2 16-node Network

Next, the performance of the 16-node low-power wireless network which contains more traffic load and data forwarders than the simple 5-node network is evaluated. This investigation was performed with two settings in addition to the configurations illustrated in Section 5.4.2. Because the 16-node consists of a high number of data forwarders, these might become congested with a large number of data packets (without changing the application data period) that they receive from their children, which makes them bottlenecks and leading to packet losses. Therefore, to allow the nodes to forward as many packets as possible, the number of free slots used by a node within the duration of a data period, $N_{slot}$, is varied.

**PDR.** Figure 5.14 (also refer Appendix F) compares the results with 2 and 3 free slots per data period alongside CRYSTAL and re-transmission enabled ContikiMAC.

---

**Table 5.14**

<table>
<thead>
<tr>
<th></th>
<th>PDR (%)</th>
<th>Duty-cycle (%)</th>
<th>E-efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak</strong> HOME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUCID (2FS)</td>
<td>95.0</td>
<td>2.5</td>
<td>700</td>
</tr>
<tr>
<td>LUCID (3FS)</td>
<td>97.5</td>
<td>2.0</td>
<td>750</td>
</tr>
<tr>
<td>CRYSTAL</td>
<td>92.0</td>
<td>3.0</td>
<td>650</td>
</tr>
<tr>
<td>ContikiMAC (rexmt)</td>
<td>88.0</td>
<td>3.5</td>
<td>600</td>
</tr>
</tbody>
</table>

**Figure 5.14**

Comparison of PDR (top), duty-cycle (bottom), and energy-efficiency in terms of $PDR/duty-cycle$ (bottom) in the 16-node network with the SoA. Here, the performance of LUCID that utilises 2 and 3 free slots (receive slots) per data period.
The increase of the number of free slots per data period boosts the network-wide $PDR$ in all the combinations of noise settings. This gain is prominent in the office off-peak noise with 3.1% and 3.6%, respectively for both 10 and 60 seconds data periods. Furthermore, in comparison with the simple 5-node network, the 16-node sensor network with 2 free slots per data period exhibits more than 3.4% degradation in the $PDR$. This is due to the lack of sub-slots to transmit data and sub-slot collisions especially in node 9 (see Figure 5.11b). Note that the transmit queue is sufficiently large (i.e., 16) to avoid congestion losses. With the adjustment of 3 free slots per data period, the network managed to improve the difference in $PDR$ of the two networks by $\approx 3\%$ on average. After increasing the number of slots, there is a sufficient amount of sub-slots for data transmission. Nonetheless, the sub-slot collision could still happen as the same slot length is used on both occasions.

LUCID has slightly better $PDR$ (i.e., 1.2% increase) than CRYSTAL in the home with a 60 seconds data period and ContikiMAC (rexmt) in all the cases except for the home peak noise. Because the noise is more bursty in the home than the office as discussed in Section 3.3, neither CRYSTAL’s noise detection technique nor the random access method was able to mitigate the RF noise better than LUCID in a multi-hop network. However, CRYSTAL is capable of mitigating light noise slightly better than LUCID, as it delivers a $PDR$ increase of 3.4% in the office off-peak noise conditions w.r.t. LUCID.

In the worst-case (peak noise in home with 60 seconds data period) as well as the average-case, the 95% confidence interval of $PDR$ in LUCID and CRYSTAL are:

- worst-case: $92.00 \pm 1.18\%$ and $90.83.0 \pm 2.98\%$, and
- average-case: $92.99 \pm 1.06\%$ and $92.24 \pm 1.68\%$, respectively.

LUCID demonstrates slightly better reliability than CRYSTAL in both worst-case and the average-case noise settings in the 16-node network with multi-hop communication. However, the differences are negligible.

**Energy use.** Because the transceiver is on for longer, the proposed solution with 3 free slots per data period requires more energy than that with 2 free slots. This is evident in the duty-cycle, wherein 0.5% and 0.08% increase in the duty-cycle is visible. Furthermore, the duty-cycle of LUCID is always less than that of CRYSTAL and ContikiMAC (rexmt) even with the increase in the number of free slots per data period.
5.5 Summary

**Energy efficiency.** Interestingly, despite the relatively low PDR, LUCID with 2 slots per data period showcases the highest energy efficiency for all the noise settings, while ContikiMAC performs the worst. LUCID (with 3 slots per data period) is more energy-efficient than CRYS- TAL by 7.4% and 2.1%, respectively for the worst-case and the average-case.

### 5.5 Summary

In this chapter, the design of LUCID which comprises a deployment phase to access the radio environment and a white-space prediction mechanism with white-space models was presented in detail. The technique uses a time slotted approach, thus network-wide time synchronisation is essential. Radio medium access by low-power wireless nodes in the network is proactive and is based on a novel model-based receiver-aware concept. The accurate white-space models are key to LUCID for which noise measurements are mandatory. Because the noise measurements need a considerable energy budget, it is recommended to accomplish with a secondary network. Moreover, to adapt to the dynamic noise conditions, the performance of the network is continuously monitored to trigger the white-space model re-parametrisation, wherein the model parameters are substituted by new ones when the radio medium changes its properties. This feedback loop is needed to keep the performance of the solution above the desired level for the wireless network.

The performance of LUCID was evaluated in COOJA with different data periods that represent high and low data rate applications under varying noise settings. LUCID was compared with CRYS- TAL and ContikiMAC, which are solutions from two different paradigms to address the effects of RF noise. The results demonstrated that LUCID tackles high bursty noise in wireless networks well while consuming slightly less energy for communication than CRYS- TAL and ContikiMAC, increasing the performance of low-power wireless networks.
Chapter 6

Conclusions and Outlook

This final chapter summarises the contributions of the thesis and lessons learned, acknowledges the limitations of the proposed receiver-aware MAC protocol (LU-CID), and discusses the potential for further research directions in the underlying topic of this doctoral thesis.

6.1 Contributions

This thesis aims at investigating RF noise in IEEE 802.15.4 based low-power wireless networks and designing and developing solutions that increase the performance of them. In particular, the focus was on indoor environments that exhibit varying noise conditions.

The study was begun by measuring radio frequency noise, and first, the concentration was on how the noise sources, especially Wi-Fi, generate noise in indoor environments. Because Wi-Fi has short inter-frame spaces with their high data rates, the IEEE 802.15.4 devices are unable to detect all Wi-Fi noise. It was not entirely evident from the analysis whether the IEEE 802.15.4 devices experience what the noise sources generate. Therefore, the investigation was moved toward sensor nodes’ perspectives where noise reception by the IEEE 802.15.4 devices is considered rather than the viewpoint of sources generating noise, which paved the way to study noise from unknown sources perceived by low-power wireless nodes. The following text summarises the key contributions of this doctoral thesis.

1. **A mechanism to analyse patterns in noise traces.** To identify noise patterns, the radio environment is sampled for collecting noise measurement traces. These traces are characterised by the mean Time Between Noise
Signals (TBNS) and the number of identified noise signals within a selected time period. The two-dimensional distribution of the aforementioned two metrics and a tool called “Normalised Cross-Likelihood Ratio (NCLR)” are the key to identify noise patterns.

2. **A mechanism to use noise patterns to predict noise-free opportunities for transmission.** A white-space model is used to estimate noise patterns and to predict noise-free opportunities for wireless nodes. This model is parametrised based on the noise conditions in the deployed environment. The parametrisation of the model is done with the help of knowledge about the noise patterns. The white-space model is trained to have different parameter sets that are suitable for varying noise conditions and the most suited parameter set is chosen based on the performance of the wireless network.

3. **A protocol, LUCID, that utilises the prediction mechanism to identify rendezvous points for low-interference communication.** The predicted white-spaces from the white-space model is the key to finding the rendezvous points between a transmitter and a receiver. To this end, the receiver’s white-space model is used by the transmitter to decide the receive time slots during which it transmits packets. As the transmitter transmits packets in a round-robin fashion in the dedicated sub-slot within the selected receive time slot, the technique can minimise the collision of packets with other nodes that transmit in the same receive time slot. Therefore, LUCID can effectively tackle the hidden terminal problem as well.

LUCID uses a hybrid time slotted radio medium access mechanism for scheduling transmissions. This technique uses good properties of both centralised and decentralised scheduling paradigms, wherein tight time synchronisation is coordinated by a central entity, the network coordinator, while the actual transmission scheduling is done locally in the nodes with the use of its receiver’s white-space model. The transmission opportunities predicted by the receiver’s white-space model are the key to deciding the transmission timing w.r.t. time slots of individual nodes. It is noteworthy that LUCID is designed and evaluated for periodic data applications.

To have packet loss awareness and adaptation to packet loss dynamics, continuous monitoring of data communication reliability is done to identify
change points. Detection of such change points triggers the re-parametrisation of the white-space model to align with the new noise patterns.

4. A large set of real-world indoor noise traces. The traces were acquired considering both the noise source (Wi-Fi) and the sensor node’s perspectives (noise from unknown sources) and can be used to reproduce realistic indoor noise patterns in a precise and repeatable manner. Although in this work realistic indoor patterns were generated in a simulator, the same can be achieved in testbeds as well. Moreover, existing tools, such as JamLab [23], can use probability distributions extracted from the collected traces for generating noise patterns. The noise traces were characterised in terms of mean Time Between Noise Signals (TBNS) and the noise signal count per slot basis. The two-dimensional characterisation led to identify hidden noise patterns and thus to accurately marking them.

All the aforementioned features/contributions collectively provide LUCID’s robustness and modestly improve the performance of low-power wireless networks.

6.2 Lessons Learned

The study revealed that despite its ability to estimate saturated and unsaturated Wi-Fi traffic, the performance of the MMPP(2) model is heavily dependent on the Hurst parameter which is non-trivial to be estimated. Often different methods yield conflicting estimations about the value of the Hurst parameter. Therefore, the MMPP(2) model is not recommended for estimating noise in low-power wireless networks.

Looking at the sensor nodes’ perspectives is an efficient way to design and develop techniques to mitigate RF interference caused by unknown noise sources in low-power wireless networks. The slot based collaborative use of the mean TBNS and the noise signal count provided an efficient way to characterise noise from unknown sources rather than using time between noise signals alone. The NCLR is a useful metric to identify patterns in the noise perceived by the sensor nodes.

Even if the white-space model can accurately predict white-spaces, their prediction accuracy is affected by the noise dynamics. The effectiveness of the use of the white-space model with multiple parameter sets that are tailored for varying noise conditions along with a feedback loop to notify the change in the radio environment has been shown.
LUCID was evaluated in COOJA with the measured noise traces in the two indoor environments with two noise settings: peak and off-peak. The limited dataset bounds the performance evaluation of the proposed protocol.

The white-space prediction mechanism was designed as a technique to tackle CTI. However, later, it turns out that the same mechanism can be easily used as a MAC protocol for finding rendezvous points between a sender and a receiver. Furthermore, the proposed MAC (LUCID) demonstrated promising results by showcasing slight improvements w.r.t. CRYSTAL. These improvements are more marked in bursty noise conditions with high reliability and high energy efficiency (which can be further improved as discussed in Section 6.3) with the increasing network size, which is a benefit of having LUCID over CRYSTAL.

6.3 Limitations and Further Research Directions

Although LUCID delivers productive and encouraging results, there are a number of limitations that come along with it. Those limitations and their potential improvements are discussed here, as they will open further research directions in the underlying research field of this doctoral thesis.

First, the proposed medium access technique is not a plug and play solution for improving the performance of low-power wireless networks. LUCID requires to assess and understand the radio medium sufficiently enough to accurately parametrise the white-space model. This additional phase in which noise is measured and characterised is an overhead in terms of time and energy consumption. An identical separate secondary network can be used to accomplish this prerequisite task. Nonetheless, more research is needed to investigate the impact of the differences in the physical locations of nodes in the two networks and to design a methodology to forward the trained models from the secondary network in which the models are trained to the primary network wherein application tasks are executed.

LUCID uses predicted **FREE** time slots for communication, wherein a slot is marked **FREE** if it contains at least one noise-free sub-slot. It would also be useful to consider the degree of freeness of the slots (i.e., how many sub-slots are **FREE** within the slot) and use the freest slots. This could create an impact on lowering the packet losses, especially in high noise conditions.

The functionality of LUCID is based on predictions from the white-space model of a node as well as that of its neighbours. Generating predictions is an additional
overhead in terms of computation power required. There are two options to tackle this issue using: nodes that are equipped with a modern fast processor or edge computing techniques. Newer sensor nodes are equipped with modern processors which may be powerful enough to do the required prediction related computations. Alternatively, using edge computing techniques, it is possible to perform calculations in a more powerful node which is close to the originating source of data and share the predictions with the nodes in the network. However, it would require a communication mechanism between the nodes in the network with the powerful node to gather predictions, which requires more research.

In the design of LUCID, a model with two parameter sets, each for peak and off-peak noise conditions were used. However, in environments wherein sudden and rapid noise variations occur, two model configurations might not be able to adequately capture noise characteristics. This can be addressed by adding more than two parameter sets for the model, depending on the noise patterns. However, the use of multiple model configurations increases the memory requirement of sensor nodes. Note that individual sensor nodes must have access to the models of their neighbours as well. Managing a large number of white-space models while generating predictions from them increases the memory requirement and energy consumption of the sensor nodes. Therefore, more research is needed to minimise the memory footprint of the white-space models, cost of computational power and associated energy consumption.

When the radio environment experiences high noise, there could be cases where the white-space model is unable to predict any free receive slot. On such occasions, although the probability of collision is high, the node utilises the next $N_{\text{slot}}$ slots following the first time slot of the data period irrespective of their noise conditions. Channel hopping could overcome such circumstances and is extensively used in many protocols as a countermeasure to tackle high noise. Albeit LUCID does not use channel hopping, it can be adopted in the solution. To achieve this, first of all, a technique to generate a channel hopping sequence is needed. Similar to CRYSTAL [78], IEEE 802.15.4 channels that are 7 channels apart can be used in the hopping sequence to avoid not only persistent Wi-Fi traffic but also noise from microwave ovens. Because any channel (7 channels apart from the current one) out of 16 available in the 2.4 GHz ISM band may be suggested by the channel hopping algorithm, it is mandatory to have a trained white-space model for all 16 channels. In other words, each node should gather noise measurements from all 16 channels, characterise noise, train models, and share them with the neighbours. Furthermore,
LUCID’s control mechanism should be updated to coordinate the channel that is going to be used during the upcoming data period. All these escalate memory and computation requirements, thus further research is needed to optimise LUCID to have channel hopping.

Another limitation of LUCID is that the likelihood of sub-slot collisions increases with the growing network size, especially at forwarding nodes near the network coordinator. A straightforward solution for this issue would be to increase the slot length, but it is not an appropriate solution as you cannot keep inflating the slot length indefinitely due to the accumulating energy consumption. Nonetheless, the energy consumption of nodes with increasing slot length may be reduced by selectively going into receive mode only for those sub-slots that a node’s children utilise. Moreover, CCA can be incorporated with white-space predictions to decrease the probability of collisions. Further studies are required to examine the behaviour of forwarding nodes near the network coordinator, collisions due to overlapping sub-slots, and going into receive mode selectively with the increasing network size.

The GMM noise estimation model can be used for emulating RF noise in testbeds which are used for testing/benchmarking wireless communication networks.

Finally, although LUCID was designed for IEEE 802.15.4 based low-power wireless networks, the principles are fundamental to other wireless networks as well. Therefore, this doctoral thesis opens opportunities to research the feasibility of using the white-space prediction mechanism and the MAC protocol in other wireless communication networks as well.
References


Appendix A

Snippets of Measured Noise Traces

A.1 OFFICE Peak

<table>
<thead>
<tr>
<th>Timestamp (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1505383200.990</td>
</tr>
<tr>
<td>1505383201.000</td>
</tr>
<tr>
<td>1505383201.010</td>
</tr>
<tr>
<td>1505383201.020</td>
</tr>
<tr>
<td>1505383201.090</td>
</tr>
<tr>
<td>1505383201.100</td>
</tr>
<tr>
<td>1505383201.110</td>
</tr>
<tr>
<td>1505383201.120</td>
</tr>
<tr>
<td>1505383201.200</td>
</tr>
<tr>
<td>1505383201.210</td>
</tr>
<tr>
<td>1505383201.210</td>
</tr>
<tr>
<td>1505383201.220</td>
</tr>
<tr>
<td>1505383201.300</td>
</tr>
<tr>
<td>1505383201.310</td>
</tr>
<tr>
<td>1505383201.310</td>
</tr>
<tr>
<td>1505383201.320</td>
</tr>
<tr>
<td>1505383201.400</td>
</tr>
<tr>
<td>1505383201.410</td>
</tr>
<tr>
<td>1505383201.410</td>
</tr>
<tr>
<td>1505383201.420</td>
</tr>
</tbody>
</table>
A.2 OFFICE Off-peak

1505383201.510
1505383201.520
1505383201.530
1505383201.610
1505383201.620
1505383201.620
1505383201.630
1505383201.650
1505383201.710
1505383201.720
1505383201.730
1505383201.820
1505383201.830
1505383201.830
1505383201.830
1505383201.920
1505383201.920
1505383201.930
1505383201.950
1505383202.010
1505383202.010
1505383202.020
1505383202.020
1505383202.030
1505383202.030
1505383202.040
1505383202.040
1505383202.040
1505383202.050
1505383202.050

A.2 OFFICE Off-peak

Timestamp (seconds)
1505354400.920
1505354401.000
1505354401.020
<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1505354401.020</td>
<td></td>
</tr>
<tr>
<td>1505354401.050</td>
<td></td>
</tr>
<tr>
<td>1505354401.090</td>
<td></td>
</tr>
<tr>
<td>1505354401.100</td>
<td></td>
</tr>
<tr>
<td>1505354401.100</td>
<td></td>
</tr>
<tr>
<td>1505354401.120</td>
<td></td>
</tr>
<tr>
<td>1505354401.120</td>
<td></td>
</tr>
<tr>
<td>1505354401.190</td>
<td></td>
</tr>
<tr>
<td>1505354401.200</td>
<td></td>
</tr>
<tr>
<td>1505354401.220</td>
<td></td>
</tr>
<tr>
<td>1505354401.230</td>
<td></td>
</tr>
<tr>
<td>1505354401.290</td>
<td></td>
</tr>
<tr>
<td>1505354401.310</td>
<td></td>
</tr>
<tr>
<td>1505354401.330</td>
<td></td>
</tr>
<tr>
<td>1505354401.330</td>
<td></td>
</tr>
<tr>
<td>1505354401.390</td>
<td></td>
</tr>
<tr>
<td>1505354401.410</td>
<td></td>
</tr>
<tr>
<td>1505354401.430</td>
<td></td>
</tr>
<tr>
<td>1505354401.430</td>
<td></td>
</tr>
<tr>
<td>1505354401.500</td>
<td></td>
</tr>
<tr>
<td>1505354401.510</td>
<td></td>
</tr>
<tr>
<td>1505354401.530</td>
<td></td>
</tr>
<tr>
<td>1505354401.540</td>
<td></td>
</tr>
<tr>
<td>1505354401.550</td>
<td></td>
</tr>
<tr>
<td>1505354401.600</td>
<td></td>
</tr>
<tr>
<td>1505354401.610</td>
<td></td>
</tr>
<tr>
<td>1505354401.630</td>
<td></td>
</tr>
<tr>
<td>1505354401.640</td>
<td></td>
</tr>
<tr>
<td>1505354401.700</td>
<td></td>
</tr>
<tr>
<td>1505354401.720</td>
<td></td>
</tr>
<tr>
<td>1505354401.730</td>
<td></td>
</tr>
<tr>
<td>1505354401.740</td>
<td></td>
</tr>
<tr>
<td>1505354401.810</td>
<td></td>
</tr>
<tr>
<td>1505354401.820</td>
<td></td>
</tr>
<tr>
<td>1505354401.840</td>
<td></td>
</tr>
<tr>
<td>1505354401.840</td>
<td></td>
</tr>
<tr>
<td>1505354401.920</td>
<td></td>
</tr>
<tr>
<td>1505354401.940</td>
<td></td>
</tr>
<tr>
<td>1505354401.940</td>
<td></td>
</tr>
</tbody>
</table>
A.3 HOME Peak

Timestamp (seconds)
1505354402.010
1505354402.020
1505354402.040
1505354402.050
1505354402.120
1505354402.130
1505354402.140
1505354402.150
1505419200.040
1505419200.070
1505419200.090
1505419200.090
1505419200.090
1505419200.110
1505419200.120
1505419200.120
1505419200.120
1505419200.190
1505419200.200
1505419200.220
1505419200.300
1505419200.310
1505419200.310
1505419200.350
1505419200.390
1505419200.500
1505419200.500
1505419200.540
1505419200.600
1505419200.600
1505419200.630
1505419200.630
1505419200.640
A.4 HOME Off-peak

Timestamp (seconds)
1505419200.010
1505419200.040
1505419200.070
1505419200.080
1505419200.090
1505419200.100
1505419200.110
1505419200.120
1505419200.130
1505419200.140
1505419200.150
1505419200.160
1505419200.170
1505419200.180
A.4 HOME Off-peak

1505404800.190
1505404800.190
1505404800.200
1505404800.210
1505404800.220
1505404800.250
1505404800.330
1505404800.330
1505404800.340
1505404800.350
1505404800.350
1505404800.400
1505404800.450
1505404800.560
1505404800.560
1505404800.560
1505404800.600
1505404800.620
1505404800.620
1505404800.640
1505404800.640
1505404800.640
1505404800.660
1505404800.710
1505404800.730
1505404800.760
1505404800.760
1505404800.810
1505404800.830
1505404800.860
1505404800.930
1505404800.960
1505404800.970
1505404801.010
1505404801.070
1505404801.170
1505404801.170
1505404801.190
1505404801.210
## A.4 HOME Off-peak

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1505404801.210</td>
<td>140</td>
</tr>
<tr>
<td>1505404801.270</td>
<td></td>
</tr>
<tr>
<td>1505404801.380</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Source Code

B.1 Contiki

B.1.1 Application

B.1.1.1 Data Collection Application

The following periodic data collection application was used to simulate LUCID.

```c
#include "contiki.h"

#include "dev/serial-line.h"
#include "dev/leds.h"
#include "dev/light-sensor.h"
#include "dev/sht11-sensor.h"
#include "dev/cc2420.h"

#include "sys/etimer.h"
#include "sys/ctimer.h"
#include "sys/rtimer.h"
#include "sys/node-id.h"
#include "sys/process.h"

#include "net/rime/collect.h"
#include "net/mac/netstack_common.h"

#include <stdio.h>
#include <stdlib.h>
```
#include <string.h>

#include "project-conf.h"

/**********************************************************************************/
/* For debugging*/
#define DEBUG_APP_INFO 0
#define DEBUG_APP_WARNINGS 1
#define DEBUG_APP_ERRORS 0

#if DEBUG_APP_INFO
#undef DEBUG_APP_WARNINGS
#define DEBUG_APP_WARNINGS 1
#undef DEBUG_APP_ERRORS
#define DEBUG_APP_ERRORS 1
#endif

#if (DEBUG_APP_WARNINGS && !DEBUG_APP_INFO)
#undef DEBUG_APP_ERRORS
#define DEBUG_APP_ERRORS 1
#else /*DEBUG_APP_INFO*/
#define INFO(...) printf(__VA_ARGS__)
#endif

#if (defined RIMEADDR_SIZE && RIMEADDR_SIZE == 8)
#define printaddr(addr) INFO("%02x%02x:%02x%02x:%02x%02x:%02x%02x\n",
       ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], ((uint8_t *)addr)[2],
       ((uint8_t *)addr)[3], ((uint8_t *)addr)[4], ((uint8_t *)addr)[5],
       ((uint8_t *)addr)[6], ((uint8_t *)addr)[7])
#else//(defined RIMEADDR_SIZE & RIMEADDR_SIZE==2)
#define printaddr(addr) INFO("%02x%02x\n", ((uint8_t *)addr)[0],
       ((uint8_t *)addr)[1])
#endif
#else /*DEBUG_APP_INFO*/
#define INFO(...) printf(__VA_ARGS__)
#endif

#if DEBUG_APP_WARNINGS
...
#define WARNING(...) printf(__VA_ARGS__)
#else
#define WARNING(...) /*
#endif

#if DEBUG_APP_ERRORS
#define ERROR(...) printf(__VA_ARGS__)
#else
#define ERROR(...) /*
#endif

/**********************************************************/

/* Process Definition */
PROCESS(data_collection_app_process, "Data Collection App");
AUTOSTART_PROCESSES(&data_collection_app_process);

/* Global Parameters */
static struct ctimer ct_wsn_period, ct_print_stats;
static struct collect_conn tc;
unsigned int n_transmissions = 0;
unsigned int epoch = 0;
unsigned short n_recv_pkts = 0;
#define REXMITS 0 /* no re-transmissions */
#define APP_TIMEOUT_SEC 10 /* To make sure clocks are sync before
starting the app */

struct sky_collect_msg {
  uint16_t light1;
  uint16_t light2;
  uint16_t temperature;
  uint16_t humidity;
  uint16_t rssi;
  rtimer_clock_t timestamp;
  char txt [44];
};

/* Functions */
static void trigger_packet(void *ptr);
static void start_epoch(void *ptr);

/**************************************************
/* Obtain the synchronized time of the node with sink */
static rtimer_clock_t
get_tsync_time(){
    return RTIMER_NOW() + synchronized_offset;
}

/**************************************************
/* Calculate drift wrt previous transmission time */
static int
cal_drift_second(){
    rtimer_clock_t now = get_tsync_time();
    short drift = now % RTIMER_SECOND;
    if(drift > RT_THR){
        /* Find closest second */
        drift = RTIMER_SECOND - drift;
        drift = - drift; /* We want to add this drift, so negative */
    }
    int drift_ct = (int)1.0*drift*CLOCK_SECOND/RTIMER_SECOND;
    return drift_ct;
}

/**************************************************
/* Get radio on time in ms */
unsigned long
get_radio_on_time(){
    unsigned long avg_radio_on=0;
    #if ENERGEST_CONF_ON
    // Compute average radio-on time.
    avg_radio_on = (energest_type_time(ENERGEST_TYPE_LISTEN) +
        energest_type_time(ENERGEST_TYPE_TRANSMIT)) * 1e6 /
        (energest_type_time(ENERGEST_TYPE_CPU) +
        energest_type_time(ENERGEST_TYPE_LPM));
    #endif /* ENERGEST_CONF_ON */
    return avg_radio_on; //ms
}
static void
sendDataPacket(void *ptr){
    struct sky_collect_msg msg;
    INFO("APP seq %u\n", n_transmissions);
    msg.timestamp = get_tsync_time();
    packetbuf_clear();
    packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER, collect_parent(&tc));
    packetbuf_set_attr(PACKETBUF_ATTR_PACKET_TYPE, PACKETBUF_ATTR_PACKET_TYPE_DATA);
    packetbuf_copyfrom(&msg, sizeof(struct sky_collect_msg));
    if(collect_send(&tc, REXMITS) && node_id != 1){
        n_transmissions++;
        printf("S %lu,%lu\n", epoch, (parent_mac.u8[1]<<8) + parent_mac.u8[0]);
    }
}

static void
trigger_packet(void *ptr){
    clock_time_t waiting_time;
    int drift = 0;
    drift = cal_drift_second();
    waiting_time = APP_DATA_PERIOD - drift;
    ctimer_set(&ct_wsn_period, waiting_time, sendDataPacket, &ct_wsn_period);
    return;
}

static void
print_stats(void *ptr){
    unsigned long avg_radio_on = get_radio_on_time();
    if(node_id == 1){
        unsigned long pdr;
        pdr = n_recv_pkts*10e4/(TOT_NUM_NODES - 1);
        printf("RS %lu.%03lu,%lu.%03lu\n",..."
avg_radio_on / 1000, avg_radio_on % 1000, pdr/1000,
pdr%1000); n_recv_pkts = 0; } else{
    printf("SS %lu,%lu,%03lu,%d\n", epoch, avg_radio_on / 1000,
            avg_radio_on % 1000, (parent_mac.u8[1]<<8) + parent_mac.u8[0]);
    
    } 
}

static void
start_epoch(void *ptr){
clock_time_t waiting_time;
    int drift = 0;
    epoch++;
    drift = cal_drift_second();
    waiting_time = APP_DATA_PERIOD - drift;
    ctimer_set(&ct_wsn_period, waiting_time, start_epoch,
            &ct_wsn_period);
    if(node_id != 1){
        sendDataPacket(NULL);
    }
    /* Print stats just 1s before next round */
    waiting_time = APP_DATA_PERIOD - drift - CLOCK_SECOND;
    ctimer_set(&ct_print_stats, waiting_time, print_stats,
            &ct_print_stats);
}

static void
start_app(void *ptr){
clock_time_t waiting_time;
    int drift = 0;
    drift = cal_drift_second();
    waiting_time = (CLOCK_SECOND*(APP_START_TIMEOUT_SEC -
            APP_TIMEOUT_SEC)) - drift;
    ctimer_set(&ct_wsn_period, waiting_time, start_epoch,
            &ct_wsn_period);
    /* DC calculations */
B.1 Contiki

```c
#ifndef ENERGEST_CONF_ON
    // Initialize Energest values.
    energest_init();
#endif

return;
}

/***********************************************************************/
static void
recv(const rimeaddr_t *originator, uint8_t seqno, uint8_t hops){
    struct sky_collect_msg *msg = malloc(sizeof(struct sky_collect_msg));
    memcpy(msg, packetbuf_dataptr(), sizeof(struct sky_collect_msg));
    INFO("DP %u %u %u
",
        (originator->u8[1] << 8) + originator->u8[0],
        seqno, hops, RTIMER_NOW() - msg->timestamp);
    free(msg);
    if(node_id == 1){
        n_transmissions++;
        no_recv_pkts++;
        n_recv_pkts++;
        INFO("%u\n", n_transmissions);
    }
}

/**********************************************************************/
static const struct collect_callbacks callbacks = { recv };

/**********************************************************************/
PROCESS_THREAD(data_collection_app_process, ev, data){
    PROCESS_EXITHANDLER(goto exit;)
    PROCESS_BEGIN();
    static struct ctimer ct;
    leds_on(LEDS_GREEN);
    INFO("CLOCK_SECOND = %lu\n", CLOCK_SECOND);
    INFO("Data period: %u sec\n", APP_DATA_PERIOD_SEC);
    INFO("THR %d\n", PDR_THR);
    INFO("APP pkt size %d bytes\n", sizeof(struct sky_collect_msg));
}```
The following configurations were used in the periodic data collection application.

```c
#ifndef __PROJECT_CONF_H__
#define __PROJECT_CONF_H__

#if defined (APP_10)
#endif

#define APP_DATA_PERIOD_SEC

#endif
```

B.1.1.2 Project Configuration File

The following configurations were used in the periodic data collection application.
#define APP_DATA_PERIOD_SEC 10
#endif

#if defined (APP_60)
#undef APP_DATA_PERIOD_SEC
#define APP_DATA_PERIOD_SEC 60
#endif

#if defined TIMEOUT_3
#undef EMA_BELOW_THR_CNT_MAX
#define EMA_BELOW_THR_CNT_MAX 3
#endif
#if defined TIMEOUT_4
#undef EMA_BELOW_THR_CNT_MAX
#define EMA_BELOW_THR_CNT_MAX 4
#endif
#if defined TIMEOUT_5
#undef EMA_BELOW_THR_CNT_MAX
#define EMA_BELOW_THR_CNT_MAX 5
#endif
#if defined TIMEOUT_6
#undef EMA_BELOW_THR_CNT_MAX
#define EMA_BELOW_THR_CNT_MAX 6
#endif

#define PDR_THR 93
#define NUM_PDR_SAMPLES_PER_WINDOW 40 /* This is N when calculating EMA */
#define MODEL_CALIBRATION_ON 1 /* if 1: switch models when PDR is below THR */

#define APP_DATA_PERIOD (APP_DATA_PERIOD_SEC*CLOCK_SECOND)
#define RT_APP_DATA_PERIOD (APP_DATA_PERIOD_SEC*RTIMER_SECOND)
#define APP_START_TIMEOUT_SEC 90
#define MODEL_TRANSITION_TIMEOUT_SEC 60
#define MODEL_TRANSITION_MIN_NUM_SAMPLES 5

/* Network parameters */
/* Here we define the network size including sink */
define TOT_NUM_NODES 5 //5 //16 //adis

/* Debugging configurations */
define DEBUG_ADIS 0
define DEBUG_LEDS_ADIS 1
#define DEBUG_PACKETS 1
#define DEBUG_APP_ADIS 1

define RTIMER_SECOND RTIMER_ARCH_SECOND
#ifdef CLOCK_CONF_SECOND
#undef CLOCK_CONF_SECOND
#define CLOCK_CONF_SECOND 256UL
#else
#define CLOCK_CONF_SECOND 256UL
#endif

/* Serial-line buffer size: should be a power of 2 (default=128) */
#ifdef SERIAL_LINE_CONF_BUFSIZE
#undef SERIAL_LINE_CONF_BUFSIZE
#define SERIAL_LINE_CONF_BUFSIZE 128
#endif

/* Network Layer*/
#undef NETSTACK_NETWORK
#define NETSTACK_NETWORK rime_driver

/* MAC parameters */
#undef NETSTACK_MAC
#define NETSTACK_MAC wspmac_driver //LUCID: core/net/mac/[wspmac, wsprdc]

/* RDC Layer*/
#undef NETSTACK_RDC
#define NETSTACK_RDC wsprdc_driver
#undef CONTIKIMAC_CONF_SHORTEST_PACKET_SIZE
#define CONTIKIMAC_CONF_SHORTEST_PACKET_SIZE 43
#undef NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE
#define NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE 32
# undef NETSTACK_FRAMER
#define NETSTACK_FRAMER framer_802154

/* Radio Layer*/
#undef NETSTACK_RADIO
#define NETSTACK_RADIO cc2420_driver
#undef RF_CHANNEL
#define RF_CHANNEL 18
#define CC2420_CONF_SFD_TIMESTAMP 1 // enable PHY timestamping
#define WITH_WSPMAC 1 // this must be enabled
#define COLLECT_CONF_ANNOUNCEMENTS 1

#endif /* _PROJECT_CONF_H_ */

### B.1.1.3 Makefile

The following is the Makefile used to compile the data collection application.

```makefile
CONTIKI_PROJECT=data_collection_app

all: $(CONTIKI_PROJECT)

## Set these flags commandline
## e.g. $ make APP_10=1
ifeq ($(APP_10), 1)
  CFLAGS+=-DAPP_10=1
  $(info APP_DATA_PERIOD_SEC=10)
endif
ifeq ($(APP_60), 1)
  CFLAGS+=-DAPP_60=1
  $(info APP_DATA_PERIOD_SEC=60)
endif
ifeq ($(TIMEOUT_3), 1)
  CFLAGS+=-DTIMEOUT_3=1
  $(info EMA_BELOW_THR_CNT_MAX=3)
endif
ifeq ($(TIMEOUT_4), 1)
  CFLAGS+=-DTIMEOUT_4=1
```
B.1.2 LUCID

B.1.2.1 LUCID Code

The following code implements LUCID.

```c
#include "net/mac/wspmac.h"
#include "net/mac/frame802154.h"
#include "net/mac/rdc.h"
#include "net/mac/netstack_common.h"

#include "net/netstack.h"
#include "net/packetbuf.h"
#include "net/queuebuf.h"
#include "net/rime.h"
#include "net/rime/collect-neighbor.h"

#include "sys/rtimer.h"
#include "sys/ctimer.h"
#include "sys/clock.h"
#include "sys/etimer.h"
#include "sys/process.h"
#include "sys/node-id.h"
```
B.1 Contiki

```c
#include "dev/serial-line.h"
#include "dev/leds.h"
#include "dev/radio.h"
#include "dev/cc2420.h"
#include "dev/cc2420_const.h"
#include "lib/random.h"
#include "lib/list.h"
#include "lib/memb.h"
#include "cfs/cfs.h"
#include "cfs/cfs-coffee.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/************************************************************************
MEMB(neighbor_memb, struct neighbor_queue, CSMA_MAX_NEIGHBOR_QUEUES);
MEMB(packet_memb, struct rdc_buf_list, MAX_QUEUED_PACKETS);
MEMB(metadata_memb, struct qbuf_metadata, MAX_QUEUED_PACKETS);
LIST(neighbor_list_csma);
LIST(neighbor_list_wspmac);

/* Declare memory for traffic models */
MEMB(model_memb, struct traffic_model, WSPMAC_MAX_NEIGHBOR_MODELS);
MEMB(model_pkt_memb, struct model_pkt, WSPMAC_MAX_CTRL_PKTS);
MEMB(timesync_memb, struct timesync, WSPMAC_MAX_NEIGHBOR_MODELS);
MEMB(free_slot_memb, struct free_slot,
  2*MAX_NUM_SLOT_REQUESTS_PER_MODEL); //own + parent models
MEMB(rfsi_entry_memb, struct rfsi_entry, MAX_NUM_PREDICTIONS);
MEMB(model_info_memb, struct model_info, TOT_NUM_NODES);
LIST(tx_model_pkt_list);
LIST(recv_model_pkt_list);
LIST(time_offset_list);
LIST(parent_fs_list);
```
LIST(own_fs_list);

#define DEBUG 0
#if DEBUG
#define PRINTF(...) printf(__VA_ARGS__)
#endif
#define PRINTADDR(addr) PRINTF("%02x%02x:%02x%02x:%02x%02x:%02x%02x ",
   
   ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], \\
   ((uint8_t *)addr)[2], ((uint8_t *)addr)[3], \\
   ((uint8_t *)addr)[4], ((uint8_t *)addr)[5], \\
   ((uint8_t *)addr)[6], ((uint8_t *)addr)[7])
#else /* DEBUG */
#define PRINTF(...) 
#define PRINTADDR(addr) 
#endif /* DEBUG */
#endif /* DEBUG */
#define MAX_NUM_MODEL_REXMT 3
#define MAC_REXMT_ON 0

/* Functions */
static void transmitter(struct rttimer *rt, void *ptr);
void send_model_pkt(void *ptr); // (uint8_t is_rexmt);
uint8_t mac_broadcast(uint8_t cmd_type, void *pl, uint16_t data_len, 
   uint8_t is_rexmt); /* Broadcast control packets */
void compute_model();
void get_future_slots(void *ptr);
void req_slot_parent(void *ptr);
void req_slot_own(void *ptr);
#if MAC_REXMT_ON == 1
static void retransmit_model_pkt(void *n);
#endif
/* Control variables */
uint16_t seqno;
uint8_t mu_count = 0, sigma_count = 0, mixmat_count = 0, prior_count = 0, transmat_count = 0;
uint8_t *model_pl;
uint8_t num_rexmt_model_pkt = 0,
   last_rfsi_own[MAX_NUM_PREDICTIONS_PER_REQ],
last_rfsi_parent[MAX_NUM_PREDICTIONS_PER_REQ];
unsigned long last_slot_parent;
static struct ctimer ct_radio_off, ct_radio_on_app, ct_radio_on_now,
    ct_radio_on_net,
    ct_radio_on_model, ct_radio_on_mac_time, ct_tu_now,
    ct_rdc_on, ct_ack; //ct_parent_radio_off,
    ct_radio_on_mac_pdr;
static struct duty_cycle dc_app, dc_mac_time, dc_model; //dc_net,
    dc_mac_pdr,
static struct ctimer ct_matlab_timeout, ct_send_model,
    ct_broadcast_model, ct_model_pkt_recv,
    ct_transmit, ct_req_slot, ct_model_calibrate, ct_calmod,
    ct_recal_flag_reset, ct_reset_cnt;
static struct rtimer rt_slot;
static struct etimer et_radio; //, et_slot;
unsigned long corresponding_slot_parent, corresponding_slot_own;
unsigned long current_base_slot = 0; //needs when changing models
double pdr_ema_last = 0.0;
double pdr_sum = 0.0;
uint8_t pdr_sample_count;
int calmod_sent_count = 0;
short tu_count = 0;
static int MAX_CALMOD_PKTS = 5;
long platform_seconds_drift = 0;
static int ema_below_thr_cnt = 0;

/* Flags */
uint8_t timer_on = 0, duty_cycling_on = 0;
uint8_t have_own_model = 0, have_parent_model = 1;
uint8_t recalibrating = 0;
uint8_t receiving_neighbor_model = 0, receiving_matlab_model = 0;
uint8_t ack_recv = 0;
uint8_t expecting_packet = 0; /* Use to detect if a packet is expected
to be recv */
uint8_t receiver_on = 1; /* Use to detect if the receiver is on */
uint8_t radio_locked, radio_lock_on, radio_lock_off;
uint8_t change_model_flag = 0;
uint8_t recal_cmd_sent = 0;
struct ack_msg {
    rimeaddr_t src_addr;
    uint8_t seq;
};

/*@ A lock for controlling radio */
#define MAC_GET_LOCK() radio_locked++
static void
MAC_RELEASE_LOCK(void){
    if(radio_locked == 1) {
        if(radio_lock_on) {
            on();
            radio_lock_on = 0;
            receiver_on = 1;
            // printf("r_on lr\n");
        }
        if(radio_lock_off) {
            receiver_on = 0;
            off(0);
            radio_lock_off = 0;
            // printf("r_off lr\n");
        }
    }
    radio_locked--;
}

static unsigned int
status(void){
    uint8_t status = 0;
    CC2420_GET_STATUS(status);
    return status;
}

/*@ Calculate time drift between sink and the node */
/*@ wrt the start of a second */
static long
cal_drift_second(){
    rtimer_clock_t now = get_tsync_time();
    long drift = now % RTIMER_SECOND;
    if(drift > RT_THR){
        /* Find closest second */
        drift = RTIMER_SECOND - drift;
        drift = - drift; /* We want to add this drift, so negative */
    }
    long drift_ct = (long)1.0*abs(drift)*CLOCK_SECOND/RTIMER_SECOND;
    if(drift < 0){
        drift_ct = -drift_ct;
    }
    return drift_ct;
}

static long cal_rtimer_drift_slot(){
    rtimer_clock_t now = get_tsync_time();
    long drift = now % RT_SLOT_LENGTH;
    if(drift > RT_THR_SLOT){
        drift = RT_SLOT_LENGTH - drift;
        drift = - drift; /* We want to add this drift, so negative */
    }
    return drift;
}

static long cal_drift_slot(){
    long drift = cal_rtimer_drift_slot();
    long drift_ct = (long)1.0*abs(drift)*CLOCK_SECOND/RTIMER_SECOND;
    if(drift < 0){
        drift_ct = -drift_ct;
return drift_ct;
}

/**
  * Delete old prediction entries from the memory */
void
remove_old_entries(list_t fs_list, short is_parent){
    INFO("rm P%d old list\n", is_parent);
    process_poll(&wsmac_process);
    struct free_slot *fs_old, *fs = list_head(fs_list);
    struct rfsi_entry *e, *e_old;
    while(fs != NULL){
        if(fs->slot < current_slot_id - (SLOTS_PER_DATA_PERIOD/2)){
            e = list_head(fs->rfsi_list);
            while(e != NULL){
                e_old = e;
                e = list_item_next(e);
                list_remove(fs->rfsi_list, e_old);
                memb_free(&rfsi_entry_memb, e_old);
            }
            /* Get next entry in the list*/
            fs_old = fs;
            fs = list_item_next(fs);
            list_remove(fs->rfsi_list, fs_old);
            memb_free(&rfsi_entry_memb, e_old);
        }
        /* Get next entry in the list*/
        fs_old = fs;
        fs = list_item_next(fs);
        list_remove(fs->rfsi_list, fs_old);
        memb_free(&free_slot_memb, fs_old);
    }else{
        break;
    }
}

/**
  * Delete old prediction entries from the memory */
unsigned long
remove_later_entries(list_t fs_list, unsigned long newest_slot_we_want){
    INFO("rm future fs\n");
    struct free_slot *fs_old, *fs = list_tail(fs_list);
    struct rfsi_entry *e, *e_old;
while(fs != NULL){
    if(fs->slot > newest_slot_we_want){
        e = list_head(fs->rfsi_list);
        while(e != NULL){
            e_old = e;
            e = list_item_next(e);
            list_remove(fs->rfsi_list, e_old);
            memb_free(&rfsi_entry_memb, e_old);
        }
        /* Get next entry in the list*/
        fs_old = fs;
        fs = list_tail(fs_list);
        list_remove(fs_list, fs_old);
        memb_free(&free_slot_memb, fs_old);
    }else{
        break;
    }
}
if(fs != NULL){
    return fs->slot;
}else{
    return 0;
}
}

void switch_radio_off(void *ptr){
    /* Do nothing if the receiver is already off */
    if(receiver_on == 0){
        return;
    }
    /* If we are called when the driver is locked, we indicate that the
     radio should be turned off when the lock is unlocked. */
    if(radio_locked) {
        radio_lock_off = 1;
        return;
    }
279 /* timer for residual RDC calculations stops here */
280 etimer_stop(&et_radio);
281
282 /* Radio is not locked */
283 MAC_GET_LOCK();
284 /* If we are currently receiving a packet (indicated by SFD == 1),
285 we don't actually switch the radio off now, but signal that the
286 driver should switch off the radio once the packet has been
287 received and processed, by setting the 'radio_lock_off'
288 variable. */
289 if((status() & BV(CC2420_TX_ACTIVE)) ||
   packetbuf_attr(PACKETBUF_ATTR_PENDING) ||
   NETSTACK_RADIO.receiving_packet()) {
290   radio_lock_off = 1;
291   /* off after 4 ms */
292   ctimer_set(&ct_radio_off, RADIO_OFF_TIMEOUT_WHEN_RECEIVING,
               switch_radio_off, &ct_radio_off);
293   INFO("r_off timeout\n");
294 }
295 else {
296   receiver_on = 0;
297   off(0);
298 }
299 MAC_RELEASE_LOCK();
300 }
301
302iscriminator*********************************************************************/
303 void
304 switch_radio_on_now(void *ptr){
305   clock_time_t timeout, residual_time = 0, radio_on_duration;
306   /* Get the radio on duration */
307   if(ptr == NULL){
308     timeout = RADIO_ON_DURATION;
309   }else{
310     timeout = *(clock_time_t *)ptr;
311   }
B.1 Contiki

161

/* For calculating residual time of radio on duration */
if(!etimer_expired(&et_radio)){
    residual_time = etimer_expiration_time(&et_radio) - clock_time();
}

/* turn radio off after radio_on_duration. *
* here we compare timeout against residual_time.
* if residual time is longer we keep on radio for that period only */

//radio_on_duration = timeout + RADIO_ON_GUARD_PERIOD + residual_time;
if(timeout > residual_time){
    /* Radio is not on or the remaining duration is not sufficient */
    radio_on_duration = timeout;

/* If radio is not already on, we switch radio on. *
* If not, we keep radio on */
if(!receiver_on){
    if(radio_locked) {
        /* if locker is locked, turn on radio when the locker is unlocked */
        radio_lock_on = 1;
    }else{
        MAC_GET_LOCK();
        on();
        receiver_on = 1;
        MAC_RELEASE_LOCK();
        //printf("r_on\n");
    }
}else{
    if(radio_locked) {
        /* if locker is locked, turn on radio when the locker is unlocked */
        radio_lock_on = 1;
        radio_lock_off = 0;
    }else{
        MAC_GET_LOCK();
        on();
        receiver_on = 1;
    }
}
radio_lock_off = 0;
MAC_RELEASE_LOCK();
//printf("r_on\n");
}
}
ctime_set(&ct_radio_off, radio_on_duration, switch_radio_off, &ct_radio_off);
etimer_stop(&et_radio);
etimer_set(&et_radio, radio_on_duration);
}
}

unsigned long
get_model_prediction(short is_parent, short
arr[MAX_NUM_PREDICTIONS_PER_REQ], short delete_record){
INFO("Get prediction\n");
int i;
unsigned long slot = 0;
list_t fs_list;
if(is_parent){
    fs_list = parent_fs_list;
}else{
    fs_list = own_fs_list;
}

/* During recalibration, we don’t use predictions
 * as the models are invalid */
if(have_own_model && have_parent_model){
    /* Get the slot entry */
    struct free_slot *fs;
    fs = list_head(fs_list);
    if(list_length(fs_list) == 0){
        fs = NULL;
        INFO("Err: %d fs_list empty\n", is_parent);
    }
/* Get the correct entry */
while(fs != NULL){
  if(is_parent){
    /* From parent model:
    * we need the current slot */
    if(fs->slot > current_slot_id - SLOTS_PER_DATA_PERIOD/2 &&
        fs->slot < current_slot_id + SLOTS_PER_DATA_PERIOD/2){
        break;
    }
  }
  else{
    /* From own model:
    * we need the future slot (T slots away) */
    if(fs->slot > current_slot_id + SLOTS_PER_DATA_PERIOD/2 &&
        fs->slot < current_slot_id + 3*SLOTS_PER_DATA_PERIOD/2){
        break;
    }
  }
  fs = list_item_next(fs);
}

/* Find the rfsi correspond to slot entry */
if(fs != NULL){
  struct rfsi_entry *e, *e_old;
  slot = fs->slot;
  e = list_head(fs->rfsi_list);
  for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){  
    if(e != NULL){
      arr[i] = e->rfsi;
      e_old = e;
      e = list_item_next(e);
      /* Remove the rfsi entry from the list */
      if(delete_record){
        list_remove(fs->rfsi_list, e_old);
        memb_free(&rfsi_entry_memb, e_old);
      }
    }else{
      if(is_parent){
        ERROR("Err: P e \%d\n", i+1);
else{
    ERROR("Err: 0 e %d
", i+1);
}
}

/* Remove the whole entry from the list */
if(delete_record){
    list_remove(fs_list, fs);
    memb_free(&free_slot_memb, fs);
}
else{
    if(is_parent){
        ERROR("Err: P fs\n");
    }else{
        ERROR("Err: O fs\n");
    }
}
}
return slot;
}

/**********************************************************/
void switch_radio_on(void *ptr){
    /* Switch on radio for the specified time */
    /* Default on duration is 20ms */
    static clock_time_t timeout;
    clock_time_t waiting_time;
    int drift = 0, i = 0;
    short rfsi[MAX_NUM_PREDICTIONS_PER_REQ];

    /* Get the radio on duration */
    if(ptr == NULL){
        timeout = RADIO_ON_DURATION;
        switch_radio_on_now(&timeout);
    }else{
        struct duty_cycle *dc = (struct duty_cycle *)ptr;
        timeout = dc->duration;
        ctimer_stop(dc->ct);
/* Configure when to turn radio on next
* Here we consider our own model as we need to switch on
* when my child is sending. */
if(dc->type == APPLICATION_DATA_PKT){
    for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){  
        rfsi[i] = -1;
    }

    /* Get prediction from own model */
    unsigned long fs_slot = get_model_prediction(0, rfsi, 1);
    short dummy = 1;
    if(node_id == 1){
        dummy = 2;
    }
    for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){  
        if(rfsi[i] == -1){
            rfsi[i] = dummy + i;
            //INFO("Err: 0 rfsi -1, %lu\n", fs_slot);
        }
    }

    /* We do not know which child is sending
    * so we have to turn on radio for the whole slot.
    * we schedule radio on at the beginning of slot
    * we also have to consider the last rfsi
    * waiting_time = T + slot_len(rfsi - last_rfsi) */
    /* We have to use 2 slots at each packet transmission.
    * The reason is that forwarder nodes might not be
    * able to relay all received packets from its children */

    /* 1st slot */
    drift = cal_drift_slot();
    waiting_time = APP_DATA_PERIOD + ((rfsi[0] -
        last_rfsi_own[0])*SLOT_LENGTH) -
        drift + APP_DRIFT - RADIO_ON_GUARD_PERIOD;
    ctimer_set(&ct_radio_on_app, waiting_time, switch_radio_on, dc);
    switch_radio_on_now(&timeout);
/ * 2nd slot and following slots if any */
for(i=1;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){  
  uint8_t rfsi_diff = last_rfsi_own[i] - last_rfsi_own[i-1];  
  if(rfsi_diff == 1){  
    /* Disable radio off function */  
    ctimer_stop(&ct_radio_off);  
  }  
  //waiting_time = rfsi_diff*SLOT_LENGTH;  
  drift = cal_drift_slot();  
  waiting_time = rfsi_diff*SLOT_LENGTH - drift + APP_DRIFT;  
  ctimer_set(&ct_radio_on_now, waiting_time,  
              switch_radio_on_now, &timeout);  
}

/* current_base_slot is useful when model changing */
if(fs_slot != 0){  
  current_base_slot = fs_slot - SLOTS_PER_DATA_PERIOD;  
}else{  
  current_base_slot += SLOTS_PER_DATA_PERIOD;  
}

/* For next round */
for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){  
  last_rfsi_own[i] = rfsi[i];  
}  
printf("ws %lu P%d,%d,%d\n", fs_slot, 0, rfsi[0], rfsi[1]);

}else if(dc->type == MAC_PDR_UPDATE_PKT){  
  drift = cal_drift_slot();  
  waiting_time = APP_DATA_PERIOD - drift;  
  ctimer_set(&ct_radio_on_net, waiting_time, switch_radio_on, dc);  
  switch_radio_on_now(&timeout);  
}else if(dc->type == MAC_TIME_UPDATE_PKT){  
  waiting_time = MAC_TIME_UPDATE_TIMEOUT;  
  ctimer_set(&ct_radio_on_mac_time, waiting_time,  
             switch_radio_on, dc);  
  switch_radio_on_now(&timeout);  
}else if(dc->type == MAC_MODEL_PKT){  
  /* Wait till all neighbor models are received */
/* For the protocol, you only need parent’s model */
if (node_id != 1) {
    uint8_t parent_id = (10*(uint8_t)parent_mac.u8[1] + (uint8_t)parent_mac.u8[0]);
    //uint8_t rr_id = parent_id % (MAX_NUM_NEIGHBORS + 1); //nb + own
    uint8_t rr_id = parent_id % TOT_NUM_NODES; //nb + own
    drift = cal_drift_second();
    waiting_time = (rr_id + 1)*CLOCK_SECOND - drift - RADIO_ON_GUARD_PERIOD;
    //INFO("model pkt waiting_time %lu\n", waiting_time);
    if (dc->count < 1) {
        ctimer_set(&ct_radio_on_model, waiting_time, switch_radio_on, dc);
        dc->count++;
    }
    switch_radio_on_now(&timeout);
}
}

/*********************/
void
start_radio_dutycycle(void *ptr){
    clock_time_t waiting_time;
    long drift = 0, passed_sync_time = 0;

    /* We set platform seconds here */
    int platform_second = 6;
    platform_seconds_drift = platform_second - clock_seconds(); //tsync
    - local = offset
    clock_set_seconds(platform_second);

    /* We align everything wrt sync clock second */
    drift = cal_drift_second();
    passed_sync_time = drift;

    /* Switch off radio after a timeout */
/* This will help to fully initialize the network */
waiting_time = ((INIT_RADIO_ON_DURATION_SEC -
    TIME_SYNC_INIT_TIMEOUT_SEC)*CLOCK_SECOND)
    - passed_sync_time;
ctimer_set(&ct_radio_off, waiting_time, switch_radio_off,
    &ct_radio_off);

/* Periodic listening */
/* For Application data */
/* Note that the application starts after a timeout */
dc_app.ct = &ct_radio_on_app;
dc_app.duration = SLOT_LENGTH + 2*RADIO_ON_GUARD_PERIOD;
dc_app.type = APPLICATION_DATA_PKT;
strcpy(dc_app.msg, "APP");

/* Check own free slot to receive from children */
short rfsi = 1;

/* We do not know which child is sending
 * so we have to turn on radio for the whole slot.
 * we schedule radio on at the beginning of slot */
waiting_time = ((APP_START_TIMEOUT_SEC - TIME_SYNC_INIT_TIMEOUT_SEC -
    RCD_INIT_TIMEOUT_SEC)*CLOCK_SECOND) - passed_sync_time +
    APP_DRIFT +
    (rfsi*SLOT_LENGTH) - RADIO_ON_GUARD_PERIOD;

int i;
for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){
    last_rfsi_own[i] = rfsi+i;
}
ctimer_set(&ct_radio_on_app, waiting_time, switch_radio_on,
    &dc_app); //APP

/* MAC layer: for time update packets */
dc_mac_time.ct = &ct_radio_on_mac_time;
dc_mac_time.duration = SLOT_LENGTH; //RADIO_ON_DURATON;
dc_mac_time.type = MAC_TIME_UPDATE_PKT;
strcpy(dc_mac_time.msg, "MACT");
waiting_time = MAC.TIME.UPDATE.TIMEOUT + (5 -
    TIME_SYNC.INIT.TIMEOUT_SEC - RCD.INIT.TIMEOUT_SEC)*CLOCK.SECOND
    - passed_sync_time - SUB_SLOT_LEN;
ctimer_set(&ct_radio_on_mac_time, waiting_time, switch_radio_on,
    &dc_mac_time); //mac time

#if MODEL_CALIBRATION_ON
    /* MAC layer: for PDR update packets */
dc_mac_time.ct = &ct_radio_on_mac_time;
dc_mac_time.duration = SLOT_LENGTH; //RADIO.ON.DURATION;
dc_mac_time.type = MAC.PDR.UPDATE_PKT;
strcpy(dc_mac_time.msg, "MACP");
waiting_time = ((APP.START.TIMEOUT_SEC - TIME_SYNC.INIT.TIMEOUT_SEC -
    RCD.INIT.TIMEOUT_SEC)*CLOCK.SECOND) - passed_sync_time +
    APP.DRIFT -
    RADIO.ON.GUARD.PERIOD - (APP.DATA.PERIOD - PDR.CAL.OFFSET);
ctimer_set(&ct_radio_on_mac_time, waiting_time, switch_radio_on,
    &dc_mac_time); //mac time
#endif

/* MAC model packet: radio on at 30 sec for 200ms */
/* radio will be on every 1 sec for number of neighbours */
dc_model.ct = &ct_radio_on_model;
dc_model.duration = MODEL.RADIO.ON.DURATION;
dc_model.type = MAC.MODEL_PKT;
dc_model.count = 0;
dc_model.num_neighbors = MAX_NUM_NEIGHBORS;
strcpy(dc_model.msg, "MOD");
waiting_time = ((INIT.BROADCAST.MODEL.TIMEOUT_SEC -
    TIME_SYNC.INIT.TIMEOUT_SEC - RCD.INIT.TIMEOUT_SEC)*CLOCK.SECOND)
    - passed_sync_time; // - drift;
ctimer_set(&ct_radio_on_model, waiting_time, switch_radio_on,
    &dc_model);

printf("rdc started at %u, seconds diff %ld, passed synct %ld, drift
    %ld\n",
    get_tsync_time(), platform_seconds_drift, passed_sync_time,
    drift);
}
/*******************************/
static struct neighbor_queue *
neighbor_queue_from_addr(const rimeaddr_t *addr, list_t neighbor_list){
  struct neighbor_queue *n = list_head(neighbor_list);
  while(n != NULL) {
    if(rimeaddr_cmp(&n->addr, addr)) {
      return n;
    }
    n = list_item_next(n);
  }
  return NULL;
}

/*****************************/
static clock_time_t
default_timebase(void){
  clock_time_t time;
  /* The retransmission time must be proportional to the channel
   * check interval of the underlying radio duty cycling layer. */
  time = NETSTACK_RDC.channel_check_interval();
  /* If the radio duty-cycle has no channel check interval (i.e., it
   * does not turn the radio off), we make the retransmission time
   * proportional to the configured MAC channel check rate. */
  if(time == 0) {
    time = CLOCK_SECOND / NETSTACK_RDC_CHANNEL_CHECK_RATE;
  }
  return time;
}

/*****************************/
static void
transmit_packet_list_csma(void *ptr){
  struct neighbor_queue *n = ptr;
  if(n){
    struct rdc_buf_list *q = list_head(n->queued_packet_list);
    if(q != NULL) {
B.1 Contiki

676  PRINTF("csma: preparing number %d %p, queue len %d\n",
677      n->transmissions, q,
678      list_length(n->queued_packet_list));
679
680  /* Check if we are sending or receiving
681   * CSMA might send other packets */
682  if(packetbuf_attr(PACKETBUF_ATTR_PENDING) ||
683      NETSTACK_RADIO.receiving_packet()){
684      /* Report collision: MAC may try sending later
685       * if rexmt is enabled. */
686      /* Prepare the packetbuf for callback */
687      queuebuf_to_packetbuf(q->buf);
688      /* Return MAC_TX_COLLISION so the MAC may try again later */
689      mac_call_sent_callback(packet_sent_csma, ptr,
690                              MAC_TX_COLLISION, 1);
691      return;
692  }else{
693      /* Send packets in the neighbor's list */
694      NETSTACK_RDC.send_list(packet_sent_csma, n, q);
695  }
696  }
697
698  /*********************************************************************************/
699  static void
700  free_packet_csma(struct neighbor_queue *n, struct rdc_buf_list *p){
701      if(p != NULL) {
702          /* Remove packet from list and deallocate */
703          list_remove(n->queued_packet_list, p);
704          queuebuf_free(p->buf);
705          memb_free(&metadata_memb, p->ptr);
706          memb_free(&packet_memb, p);
707          PRINTF("csma: free_queued_packet, queue length %d\n",
708                  list_length(n->queued_packet_list));
709          if(list_head(n->queued_packet_list) != NULL) {
710              /* There is a next packet. We reset current tx information */
711              n->transmissions = 0;
n->collisions = 0;
n->deferrals = 0;
/* Set a timer for next transmissions */
ctimer_set(&n->transmit_timer, default_timebase(),
    transmit_packet_list_csma, n);
} else {
    /* This was the last packet in the queue, we free the neighbor */
    ctimer_stop(&n->transmit_timer);
    list_remove(neighbor_list_csma, n);
    memb_free(&neighbor_memb, n);
}
}

/************************************************************************

static void
packet_sent_csma(void *ptr, int status, int num_transmissions){
    struct neighbor_queue *n;
    struct rdc_buf_list *q;
    struct qbuf_metadata *metadata;
    clock_time_t time = 0;
    mac_callback_t sent;
    void *cptr;
    int num_tx;
    int backoff_transmissions;
    n = ptr;
    if(n == NULL) {
        return;
    }
    switch(status) {
        case MAC_TX_OK:
        case MAC_TX_NOACK:
            n->transmissions++;
            break;
        case MAC_TX_COLLISION:
            n->collisions++;
            break;
    }
}
B.1 Contiki

750 case MAC_TX_DEFERRED:
751     n->deferrals++;
752     break;
753 }
754
755 for(q = list_head(n->queued_packet_list);
756     q != NULL; q = list_item_next(q)) {
757     if(queuebuf_attr(q->buf, PACKETBUF_ATTR_MAC_SEQNO) ==
758         packetbuf_attr(PACKETBUF_ATTR_MAC_SEQNO)) {
759         break;
760     }
761 }
762
763 if(q != NULL) {
764     metadata = (struct qbuf_metadata *)q->ptr;
765     if(metadata != NULL) {
766         sent = metadata->sent;
767         cptr = metadata->cptr;
768         num_tx = n->transmissions;
769         if(status == MAC_TX_COLLISION ||
770             status == MAC_TX_NOACK) {
771             /* If the transmission was not performed because of a
772              collision or noack, we must retransmit the packet. */
773             switch(status) {
774                 case MAC_TX_COLLISION:
775                     PRINTF("csma: rexmit collision %d\n", n->transmissions);
776                     break;
777                 case MAC_TX_NOACK:
778                     PRINTF("csma: rexmit noack %d\n", n->transmissions);
779                     break;
780                 default:
781                     PRINTF("csma: rexmit err %d, %d\n", status,
782                             n->transmissions);
783                     break;
784             }
785         }
786     }
/* The retransmission time must be proportional to the
 channel
 check interval of the underlying radio duty cycling layer. */
 time = default_timebase();

/* The retransmission time uses a linear backoff so that the
 interval between the transmissions increase with each
 retransmit. */
 backoff_transmissions = n->transmissions + 1;

/* Clamp the number of backoffs so that we don’t get a too
 long
 timeout here, since that will delay all packets in the
 queue. */
 if(backoff_transmissions > 3) {
   backoff_transmissions = 3;
 }
 time = time + (random_rand() % (backoff_transmissions *
   time));

if(n->transmissions < metadata->max_transmissions) {
  PRINTF("csma: retransmitting with time %lu %p
", time,
    q);
  ctimer_set(&n->transmit_timer, time,
    transmit_packet_list_csma, n);
  /* This is needed to correctly attribute energy that we
 spent
 transmitting this packet. */
  queuebuf_update_attr_from_packetbuf(q->buf);
} else {
  PRINTF("csma: drop with status %d after %d transmissions,
    %d collisions\n",
    status, n->transmissions, n->collisions);
  free_packet_csma(n, q);
  mac_call_sent_callback(sent, cptr, status, num_tx);
}
} else {
  if(status == MAC_TX_OK) {

PRINTF("csma: rexmit ok \%d\n", n->transmissions);
} else {
    PRINTF("csma: rexmit failed \%d: \%d\n", n->transmissions, status);
}
free_packet_csma(n, q);
mac_call_sent_callback(sent, cptr, status, num_tx);
}
}
}

/*****************************/
static void
send_packet_csma(mac_callback_t sent, void *ptr){
    struct rdc_buf_list *q;
    struct neighbor_queue *n;
    const rimeaddr_t *addr = packetbuf_addr(PACKETBUF_ADDR_RECEIVER);
    if(seqno == 0) {
        /* PACKETBUF_ATTR_MAC_SEQNO cannot be zero, due to a peculiarity in framer-802154.c. */
        seqno++;
    }
    packetbuf_set_attr(PACKETBUF_ATTR_MAC_SEQNO, seqno++);
    /* Look for the neighbor entry */
    n = neighbor_queue_from_addr(addr, neighbor_list_csma);
    if(n != NULL) {
        /* Allocate a new neighbor entry */
        n = memb_alloc(&neighbor_memb);
        if(n != NULL) {
            /* Init neighbor entry */
            rimeaddr_copy((rimeaddr_t *)&n->addr, addr);
            n->transmissions = 0;
            n->collisions = 0;
            n->deferrals = 0;
            /* Init packet list for this neighbor */
            LIST_STRUCT_INIT(n, queued_packet_list);
/* Add neighbor to the list */
list_add(neighbor_list_csma, n);
}

if(n != NULL) {
    /* Add packet to the neighbor’s queue */
    q = memb_alloc(&packet_memb);
    if(q != NULL) {
        q->ptr = memb_alloc(&metadata_memb);
        if(q->ptr != NULL) {
            q->buf = queuebuf_new_from_packetbuf();
            if(q->buf != NULL) {
                struct qbuf_metadata *metadata = (struct qbuf_metadata *)q->ptr;
                /* Neighbor and packet successfully allocated */
                if(packetbuf_attr(PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS) == 0) {
                    /* Use default configuration for max transmissions */
                    metadata->max_transmissions =
                        CSMA_MAX_MAC_TRANSMISSIONS;
                } else {
                    metadata->max_transmissions =
                        packetbuf_attr(
                            PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS);
                }
                metadata->sent = sent;
                metadata->cptr = ptr;
                metadata->relative_free_slot_id = -1;

                if(packetbuf_attr(PACKETBUF_ATTR_PACKET_TYPE) ==
                    PACKETBUF_ATTR_PACKET_TYPE_ACK) {
                    list_push(n->queued_packet_list, q);
                } else {
                    list_add(n->queued_packet_list, q);
                }

            } else {
                list_add(n->queued_packet_list, q);
            }

        } if(q->buf != NULL) {
    } else {
    q->buf = queuebuf_new_from_packetbuf();
    }

    if(q->ptr != NULL) {
        list_push(n->queued_packet_list, q);
    } else {
        list_add(n->queued_packet_list, q);
    }

} /* If q is the first packet in the neighbor’s queue, send asap */
if(list_head(n->queued_packet_list) == q) {
    ctimer_set(&n->transmit_timer, 0,
               transmit_packet_list_csma, n);
}
return;
}
memb_free(&metadata_memb, q);
PRINTF("csma: could not allocate queuebuf, dropping packet\n");
}
memb_free(&packet_memb, q);
PRINTF("csma: could not allocate queuebuf, dropping packet\n");
} /* The packet allocation failed. Remove and free neighbor entry */
if(list_length(n->queued_packet_list) == 0) {
    list_remove(neighbor_list_csma, n);
    memb_free(&neighbor_memb, n);
}
PRINTF("csma: could not allocate packet, dropping packet\n");
} else {
    PRINTF("csma: could not allocate neighbor, dropping packet\n");
} mac_call_sent_callback(sent, ptr, MAC_TX_ERR, 1);
}

="/******************************************************************/
uint8_t
is_neighbor(rimeaddr_t addr){
    list_t neighbor_list = neighbor_list_mac.list;
    struct collect_neighbor *neighbor = list_head(neighbor_list);
    while(neighbor != NULL){
        if(rimeaddr_cmp(&neighbor->addr, &addr)){
            //INFO("NB exists\n");
            return 1;
        }
        neighbor = list_item_next(neighbor_list);
    }
    return 0;
static void free_packet_wspmac(struct neighbor_queue *n, struct rdc_buf_list *p) {
    INFO("free_packet_wspmac\n");
    if(p != NULL) {
        /* Remove packet from list and deallocate */
        list_remove(n->queued_packet_list, p);

        queuebuf_free(p->buf);
        memb_free(&metadata_memb, p->ptr);
        memb_free(&packet_memb, p);
        PRINTF("wspmac: free_queued_packet, queue length %d\n",
                list_length(n->queued_packet_list));
        if(list_head(n->queued_packet_list) != NULL) {
            /* There is a next packet. We reset current tx information */
            n->transmissions = 0;
            n->collisions = 0;
            n->deferrals = 0;

            /* Set a timer for next transmissions */
            ctimer_set(&n->transmit_timer, default_timebase(),
                       transmit_packet_list_wspmac, n);
        } else {
            /* This was the last packet in the queue, we free the neighbor */
            ctimer_stop(&n->transmit_timer);
            list_remove(neighbor_list_wspmac, n);
            memb_free(&neighbor_memb, n);
        }
    }
}

/*****************************/
/* Transmit the data packet in the predicted slot. */
/* Request another free slot from MATLAB for a */
/* future data packet */
static void transmit(void *ptr){
    INFO("transmit\n");
    struct neighbor_queue *n = ptr;
    struct rdc_buf_list *q = list_head(n->queued_packet_list);

    /* Check if we are sending or receiving */
    * CSMA migh send other packets */
    if(packetbuf_attr(PACKETBUF_ATTR_PENDING) ||
      NETSTACK_RADIO.receiving_packet()){
        /* Report collision: MAC may try sending later */
        * if rexmt is enabled. */
        /* Prepare the packetbuf for callback */
        queuebuf_to_packetbuf(q->buf);
        /* Return DEFERRED so the MAC may try again later */
        mac_call_sent_callback(packet_sent_wspmac, ptr, MAC_TX_DEFERRED, 1);
        return;
    }else{
        /* Send packets in the neighbor’s list */
        NETSTACK_RDC.send_list(packet_sent_wspmac, n, q);
        INFO("txmt\n");
    }
}

/*****************************/
struct wait_time {
    long waiting_time;
    long res_time;
};
static wait_time wt;

static void find_waiting_time(long drift){
    /* Find corresponding sub-slot for the node based on */
    * current time */
    int num_ss, cur_ss_index, node_ss_index;
    long residual_time_slot, drift_slot, drift_ss, ret = 0,
    drift_after_third_ss;
node_ss_index = node_id % 2;
drift_slot = cal_drift_slot();

if(drift_slot > 0){
    residual_time_slot = SLOT_LENGTH - drift;
} else{
    residual_time_slot = abs(drift);
    drift_slot = SLOT_LENGTH - abs(drift);
}

drift_after_third_ss = drift_slot - (3*SUB_SLOT_LEN);
if(drift_after_third_ss < 0){
    drift_after_third_ss = 0;
    num_ss = 0; //num passed subslots after 3rd ss
    cur_ss_index = 0;
    if(cur_ss_index == node_ss_index){
        ret = 0;
    } else{
        ret = SUB_SLOT_LEN;
    }
} else{
    num_ss = abs(1.0*drift_after_third_ss/SUB_SLOT_LEN); //num passed subslots after 3rd ss
    drift_ss = drift_after_third_ss - (num_ss*SUB_SLOT_LEN);
    cur_ss_index = (num_ss+1) % 2; //have to add 1 as we currently in the (num_ss + 1)th ss
    if(cur_ss_index == node_ss_index){
        ret = 2*SUB_SLOT_LEN - drift_ss;
    } else{
        ret = SUB_SLOT_LEN - drift_ss;
    }
}
wt.waiting_time = ret;
wt.res_time = residual_time_slot;
//return ret;

/***************************************************************************/
static short first_fs0_packet_sent = 0;
static short fs_0_count = 0;

void
reset_fs0_counter(void *ptr){
    /* Here we reset this counter at the beginning of the period
     * that is used to detect empty predictions in
     * transmit_packet_list_wspmac() */
    fs_0_count = 0;

    /* Reset the flag for the next packet */
    first_fs0_packet_sent = 0;
}

static void
transmit_packet_list_wspmac(void *ptr){
    struct neighbor_queue *n = ptr;
    int i = 0;
    long drift = 0;
    long residual_time_slot = 0;
    clock_time_t waiting_time = -1;
    short rfsi[MAX_NUM_PREDICTIONS_PER_REQ], rfsi_value, rfsi_diff,
            divisor = 4;
    long passed_sync_time = 0;
    //passed_sync_time =
    (long)1.0*get_tsync_time() * CLOCK_SECOND/RTIMER_SECOND;

    if(n){
        //printf("tx l \%d\n", list_length(n->queued_packet_list));

        struct rdc_buf_list *q = list_head(n->queued_packet_list);
        if(q != NULL) {
            PRINTF("wspmac: preparing number %d %p, queue len %d\n",
                    n->transmissions, q,
                    list_length(n->queued_packet_list));

            /* Check if free slot is available
             * if so, utilize it
             * otherwise, use csma */
for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){
    rfsi[i] = -1;
}

/* Get prediction for parent model */
unsigned long parent_slot = get_model_prediction(1, rfsi, 0);
short dummy = 1;
if(node_id == 2){
    dummy = 2;
}
for(i=0;i<MAX_NUM_PREDICTIONS_PER_REQ;i++){
    if(rfsi[i] == -1){
        rfsi[i] = dummy + i;
    }
}

/* Counter for empty predictions within the same data period */
if(parent_slot == 0){
    fs_0_count++;
}

INFO("txmt cs %lu, ps %lu, ls %lu\n", current_slot_id,
    parent_slot, last_slot_parent);
if(parent_slot == last_slot_parent || (first_fs0_packet_sent &&
    fs_0_count > 1)){
    /* Not the first packet */
    if(parent_slot == 0){
        /* if no entry found */
        rfsi_value = 0;
    }else{
        /* The relative #slots between start of the data period
         and current slot */
        rfsi_diff = current_slot_id - parent_slot;
        rfsi_value = rfsi[1] - rfsi_diff - 1;  //how many slots
        away is the second prediction
    }

    /* drift from the start of the slot */
    drift = cal_drift_slot();
if (drift > 0) {
    residual_time_slot = SLOT_LENGTH - drift;
} else {
    residual_time_slot = abs(drift);
    drift = SLOT_LENGTH - abs(drift);
}

if (rfsi_value >= 0) {
    /* 2nd free slot is further ahead */
    /* Check if we can still use the 1st slot */
    if (residual_time_slot >= MIN_RESIDUAL_TIME &&
        residual_time_slot <= (N_SUBSLOTS_PER_SLOT - 1) * SUB_SLOT_LEN) {
        /* Use the 1st slot */
        /* there are two remaining sub slots per slot
           * we Round-Robin them */
        find_waiting_time(drift);
        waiting_time = wt.waiting_time - passed_sync_time;
        residual_time_slot = wt.res_time;
        if (waiting_time <= residual_time_slot) {
            ctimer_set(&ct_transmit, waiting_time, transmit, n);
            INFO("txmt s1 2 rfsi %d res %ld, d %ld, w %lu\n", 
                rfsi_value, residual_time_slot, drift,
                waiting_time);
            //printf("w1 %lu\n", waiting_time);
            return;
        }
    }
    /* Use the 2nd slot */
    drift = cal_drift_slot();
    waiting_time = (rfsi_value*SLOT_LENGTH) +
        residual_time_slot +
        ((node_id % divisor)*SUB_SLOT_LEN) +
        2*RADIO_ON_GUARD_PERIOD + APP_DRIFT - passed_sync_time;
    ctimer_set(&ct_transmit, waiting_time, transmit, n);
INFO("txmt s2 1 rfsi %u, res %lu, d %ld, w %lu\n",
   rfsi_value, residual_time_slot, drift, waiting_time);
   return;
} else if(rfsi_value == -1){
   /* 2nd free slot is now */
   find_waiting_time(drift);
   waiting_time = wt.waiting_time - passed_sync_time;
   residual_time_slot = wt.res_time;
   if(residual_time_slot >= MIN_RESIDUAL_TIME &&
      waiting_time <= residual_time_slot){
      ctimer_set(&ct_transmit, waiting_time, transmit, n);
      INFO("txmt s2 2 rfsi %d, res %lu d %ld, w %lu\n",
         rfsi_value, residual_time_slot, drift,
         waiting_time);
      return;
   }
}

INFO("w drop %lu\n", waiting_time);

/* In any other case
 * we drop the packet */
struct qbuf_metadata *metadata = (struct qbuf_metadata *
   q->ptr);
if(metadata != NULL) {
   mac_callback_t sent = metadata->sent;
   void *cptr = metadata->cptr;
   int num_tx = n->transmissions;

   free_packet_wspmac(n, q);
   mac_call_sent_callback(sent, cptr, MAC_TX_ERR, num_tx);
   printf("tx drop nws\n");
}
return;
}else{
   /* 1st packet */
   if(parent_slot == 0 && fs_0_count == 1){
      /* No slot entry */
      last_slot_parent += SLOTS_PER_DATA_PERIOD;
first_fs0_packet_sent = 1; //this is useful for the 2nd packet
ctimer_set(&ct_reset_cnt, APP_DATA_PERIOD - CLOCK_SECOND, reset_fs0_counter, NULL);
} else{
    /* This is the 1st slot */
    last_slot_parent = parent_slot;
}
rfsi_value = rfsi[0];

drift = cal_drift_slot();
waiting_time = (rfsi_value*SLOT_LENGTH) + ((node_id % divisor)*SUB_SLOT_LEN) -
    drift + RADIO_ON_GUARD_PERIOD + APP_DRIFT - passed_sync_time;
ctimer_set(&ct_transmit, waiting_time, transmit, n);
INFO("txmt s1 1 drift %ld, w %lu\n", drift, waiting_time);
printf("ws %lu P%d, %d\n", parent_slot, 1, rfsi[0], rfsi[1]);
return;
}
}
}

/***********************/
static void
packet_sent_wspmac(void *ptr, int status, int num_transmissions){
    struct neighbor_queue *n;
    struct rdc_buf_list *q;
    struct qbuf_metadata *metadata;
    mac_callback_t sent;
    void *cptr;
    int num_tx;

    PRINTF("pkt_sent_wspmac: status %d, #tx %d\n", status, num_transmissions);
n = ptr;
if(n == NULL) {
    return;
}
switch(status) {
case MAC_TX_OK:
case MAC_TX_NOACK:
    n->transmissions++;
    break;
case MAC_TX_COLLISION:
    n->collisions++;
    break;
case MAC_TX_DEFERRED:
    n->deferrals++;
    break;
}
for(q = list_head(n->queued_packet_list);
    q != NULL; q = list_item_next(q)) {
    if(queuebuf_attr(q->buf, PACKETBUF_ATTR_MAC_SEQNO) ==
       packetbuf_attr(PACKETBUF_ATTR_MAC_SEQNO)) {
        break;
    }
}
if(q != NULL) {
    metadata = (struct qbuf_metadata *)q->ptr;
    if(metadata != NULL) {
        sent = metadata->sent;
        cptr = metadata->cptr;
        num_tx = n->transmissions;
        if(status == MAC_TX_COLLISION ||
           status == MAC_TX_NOACK) {
            /* If the transmission was not performed because of a
             collision or noack, we must retransmit the packet. */
        }
    }
switch(status) {
    case MAC_TX_COLLISION:
        PRINTF("wspmac: rexmit collision %d\n", n->transmissions);
        break;
    case MAC_TX_NOACK:
        PRINTF("wspmac: rexmit noack %d\n", n->transmissions);
        break;
    default:
        PRINTF("wspmac: rexmit err %d, %d\n", status, n->transmissions);
        break;
}

if(n->transmissions < metadata->max_transmissions){
    transmit_packet_list_wspmac(n);
    /* This is needed to correctly attribute energy that we
    spent transmitting this packet. */
    queuebuf_update_attr_from_packetbuf(q->buf);
    PRINTF("wspmac: %p needs rexmt", q);
} else{
    PRINTF("wspmac: drop with status %d after %d
    transmissions, %d collisions\n", status, n->transmissions, n->collisions);
    free_packet_wspmac(n, q);
    mac_call_sent_callback(sent, cptr, status, num_tx);
    printf("tx drop col\n");
}
} else if(status == MAC_TX_DEFERRED){
    /* We try sending the packet after a timeout */
    if(n->deferrals <= WSPMAC_MAX_MAC_DEFERRED){
        /* we don’t know when the parent is free at this moment
        * so, we have to go through transmit_packet_list */
        transmit_packet_list_wspmac(n);
    } else{
        free_packet_wspmac(n, q);
        mac_call_sent_callback(sent, cptr, status, num_tx);
B.1 Contiki

```c
    printf("tx drop col\n");

    }
}
else {
    if(status == MAC_TX_OK) {
        PRINTF("wspmac: rexmit ok %d\n", n->transmissions);
    } else {
        PRINTF("wspmac: rexmit failed %d: %d\n", 
            n->transmissions, status);
    }
    free_packet_wspmac(n, q);
    mac_call_sent_callback(sent, cptr, status, num_tx);
}
return;
}
}

/**************************************************************************/
static void
send_packet_wspmac(mac_callback_t sent, void *ptr){
    INFO("send_packet_wspmac\n");
    struct neighbor_queue *n;
    struct rdc_buf_list *q;
    const rimeaddr_t *addr = packetbuf_addr(PACKETBUF_ADDR_RECEIVER);
    if(seqno == 0) {
        /* PACKETBUF_ATTR_MAC_SEQNO cannot be zero, due to a peculiarity 
         * in framer-802154.c. */
        seqno++;
    }
    packetbuf_set_attr(PACKETBUF_ATTR_MAC_SEQNO, seqno++);

    /* Look for the neighbor entry */
    n = neighbor_queue_from_addr(addr, neighbor_list_wspmac);
    if(n == NULL){
        /* Allocate a new neighbor entry */
        n = memb_alloc(&neighbor_memb);
        if(n != NULL){
            /*
             * */
```
/* Init neighbor entry */
    rimeaddr_copy(&n->addr, addr);
    n->transmissions = 0;
    n->collisions = 0;
    n->deferrals = 0;
/* Init packet list for this neighbor */
    LIST_STRUCT_INIT(n, queued_packet_list);
/* Add neighbor to the list */
    list_add(neighbor_list_wsmac, n);
}

if(n != NULL){
    //INFO("send addr n->addr "); printaddr(&n->addr);
    /* Add packet to the neighbor’s queue */
    q = memb_alloc(&packet_memb);
    if(q != NULL) {
        q->ptr = memb_alloc(&metadata_memb);
        if(q->ptr != NULL) {
            q->buf = queuebuf_new_from_packetbuf();
            if(q->buf != NULL) {
                struct qbuf_metadata *metadata = (struct qbuf_metadata *
                    )q->ptr;
                /* Neighbor and packet successfully allocated */
                if(packetbuf_attr(PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS) == 0) {  
                    /* Use default configuration for max transmissions */
                    metadata->max_transmissions =  
                        WSMAC_MAX_MAC_TRANSMISSIONS;
                }else {
                    metadata->max_transmissions =
                        packetbuf_attr(  
                            PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS);  
                }
            metadata->sent = sent;
            metadata->cptr = ptr;
            metadata->relative_free_slot_id = -1;
            if(packetbuf_attr(PACKETBUF_ATTR_PACKET_TYPE) ==
PACKETBUF_ATTR_PACKET_TYPE_ACK) {
    list_push(n->queued_packet_list, q);
} else {
    list_add(n->queued_packet_list, q);
}

/* If q is the first packet in the neighbor’s queue, send asap */
if(list_head(n->queued_packet_list) == q) {
    ctimer_set(&n->transmit_timer, 0,
               transmit_packet_list_wspmac, n);
    //printf("tx n pkt now\n");
    return;
} //printf("tx n pkt later\n");
    return;
}
memb_free(&metadata_memb, q->ptr);
PRINTF("wspmac: could not allocate queuebuf, dropping packet\n");
} memb_free(&packet_memb, q);
PRINTF("wspmac: could not allocate queuebuf, dropping packet\n");
}

/* The packet allocation failed. Remove and free neighbor entry */
if empty. */
if(list_length(n->queued_packet_list) == 0) {
    list_remove(neighbor_list_wspmac, n);
    memb_free(&neighbor_memb, n);
}
PRINTF("wspmac: could not allocate packet, dropping packet\n");
} else {
    PRINTF("wspmac: could not allocate neighbor, dropping packet\n");
}

mac_call_sent_callback(sent, ptr, MAC_TX_ERR, 1);
# Relay Model Calibration Command

```c
void relay_model_calibrate_cmd(void *p) {
    struct mac_ctrl_pkt *data = (struct mac_ctrl_pkt *)p;
    unsigned long base_slot = data->base_slot;
    clock_time_t waiting_time;
    data->age = age_model_calibrate;
    mac_broadcast(MAC_COMMAND_CALIBRATE_MODELS, data, 0, 0);
    calmod_sent_count++;

    /* Set timeout till the ack received */
    waiting_time = SLOT_LENGTH;
    if (calmod_sent_count < MAX_CALMOD_PKTS && calmod_sent_count > 0) {
        /* send during current data period */
        ctimer_set(&ct_calmod, waiting_time, relay_model_calibrate_cmd, data);
    } else {
        /* Stop */
        calmod_sent_count = 0;
    }
    printf("calmod a%d d\n", age_model_calibrate, calmod_sent_count);

    /* Remove all saved predictions of both own and parent models */
    /* after next data period. This makes sure we have */
    /* correct prediction from the switched model */
    remove_later_entries(parent_fs_list, base_slot+SLOTS_PER_DATA_PERIOD); //keep next parent entry
    remove_later_entries(own_fs_list, base_slot); //delete all

    /* Now start prediction requests from new models */
    /* Request slots in RR order */
    waiting_time = (node_id % TOT_NUM_NODES)*SLOT_LENGTH;
    ctimer_set(&ct_req_slot, waiting_time, req_slot_own, NULL);
}
```

/* Relayed by receiving nodes other than sink */
/* Sent by sink */
void
send_model_calibrate_cmd(void *ptr){
    struct mac_ctrl_pkt data;
    clock_time_t waiting_time;

    calmod_sent_count++;

    /* Set timeout till the ack received */
    if(calmod_sent_count <= MAX_CALMOD_PKTS){
        data.cmd_type = MAC_COMMAND_CALIBRATE_MODELS;
        data.calibrate = 1;
        data.model_type = change_model_flag;
        data.age = 0;
        data.base_slot = current_base_slot;
        age_model_calibrate = data.age;
        mac_broadcast(MAC_COMMAND_CALIBRATE_MODELS, &data, 0, 0);

        waiting_time = SLOT_LENGTH;
        ctimer_set(&ct_calmod, waiting_time, send_model_calibrate_cmd, NULL);
        printf("calmod a%d d\n", age_model_calibrate, calmod_sent_count);
    }else{
        /* Stop */
        calmod_sent_count = 0;
    }
}

/*********************
/* Calculate Exponential Moving Average (EMA) of PDR */
static void
cal_ema(double pdr_current){
    double ema = (pdr_current*SMOOTHING_FACTOR) + (1- SMOOTHING_FACTOR)*pdr_ema_last;
    pdr_ema_last = ema;
}

/*********************
/* Return exponential moving average of PDR */
double
get_pdr_ema(){
    return pdr_ema_last;
}

/**************************************************************/
/* We have to reset this flag after MODEL_TRANSITION_TIMEOUT_SEC */
/* this will effectively avoid sending recalibration packet */
void
reset_recal_flag(void *ptr){
    recal_cmd_sent = 0;
}

/**************************************************************/
void
switch_models(void *ptr){
    clock_time_t waiting_time;
    
    /* We reset the flag after a timeout */
    /* This timeout will give sufficient time to */
    /* validate if the chosen model is suitable */
    /* for prediction */
    recal_cmd_sent = 1;
    ctimer_set(&ct_recal_flag_reset, MODEL_TRANSITION_TIMEOUT, reset_recal_flag, NULL);

    #if MODEL_CALIBRATION_ON
    /* We switch models: peak <-> offpeak*/
    change_model_flag = !change_model_flag;
    INFO("calmod EMA%d\n", (int)ema);
    INFO("modch fs_len P %d, 0 %d\n", list_length(parent_fs_list), list_length(own_fs_list));
    printf("modch %u\n", change_model_flag);
    
    /* Temporary cease all prediction requests now */
    ctimer_stop(&ct_req_slot);
    
    
}
/* Set correct slot id for own and parent models before requesting predictions */
unsigned long dummy = current_base_slot + 
    (MODEL_TRANSITION_NUM_SAMPLES + 1)*SLOTS_PER_DATA_PERIOD;
if(dummy < current_slot_id){
    ERROR("corr no changes\n");
}else{
    corresponding_slot_parent = dummy;
}
corresponding_slot_own = corresponding_slot_parent;
printf("corr slot %lu\n", corresponding_slot_parent);

/* Sink starts broadcasting update model command. */
* The command is broadcast every slot for MAX_PDR_PKT_COUNT.
* we use predictions from the new models after 2 periods
* from now */
calmod_sent_count = 0;
waiting_time = RADIO_ON_GUARD_PERIOD;
ctimer_set(&ct_model_calibrate, waiting_time,
    send_model_calibrate_cmd, NULL);

/* Remove all saved predictions of both own and parent models */
* after next data period. This makes sure we have
* correct prediction from the switched model */
remove_later_entries(parent_fs_list,
    current_base_slot+SLOTS_PER_DATA_PERIOD); //keep next parent entry
remove_later_entries(own_fs_list, current_base_slot); //delete all

/* Now start prediction requests from new models */
waiting_time = (node_id % TOT_NUM_NODES)*SLOT_LENGTH;
ctimer_set(&ct_req_slot, waiting_time, req_slot_own, NULL);
#endif /*MODEL_CALIBRATION_ON*/
}

/***********************/
/* Only executed by sink */
void
update_pdr(void *ptr){
INFO("Update PDR\n");
clock_time_t waiting_time;
long drift = 0;
double pdr;

if(no_recv_pkts > MAX_NUM_DATA_PKTS_PER_SESSION){
    pdr = 100.0;
} else{
    pdr = 100.0*no_recv_pkts/MAX_NUM_DATA_PKTS_PER_SESSION;
}

/* We multiply pdr by mul_factor
  * we cannot print double variables in contiki
  * 1000 -> 100% */
pdr = pdr*PDR_MUL_FACTOR;

/* Next update */
no_recv_pkts = 0;
drift = cal_drift_second();
waiting_time = APP_DATA_PERIOD - drift;
ctimer_set(&ct_update_pdr, waiting_time, update_pdr, NULL);

/* EMA init and update*/
if(pdr_sample_count >= NUM_PDR_SAMPLES_PER_WINDOW){
    cal_ema(pdr);
    double ema = get_pdr_ema();
    int dummy = (int)ema;
    printf("PDR%d, EMA%d\n", (int)pdr, dummy);
    if(dummy < PDR_THR_MUL && !recal_cmd_sent){
        /* This is the counter that track how many samples
           * were below threshold */
        ema_below_thr_cnt++;
        /* First we should check if change of model
           * is absolutely necessary.
           * So, we allow a timeout to see if
           * EMA is improving, if not we change models. */
        if(ema_below_thr_cnt > EMA_BELOW_THR_CNT_MAX){
            ema_below_thr_cnt = 0;
        }
switch_models(NULL);
}
} else if(dummy >= PDR_THR_MUL){
    /* We reset the counter when ema when above threshold */
    ema_below_thr_cnt = 0;
}
}

return;

/**************************************************************************/
uint16_t
serialize_model_data(uint8_t *pl, struct traffic_model *model){
    uint16_t idx = 0;
    uint8_t i;

    //INFO("serialize model, addr_size %u\n", RIMEADDR_SIZE);
    printaddr(&model->addr);

    for(i=0;i<RIMEADDR_SIZE;i++){
        //INFO("s addr [%u]: %02x\n",RIMEADDR_SIZE-1-i,
            model->addr.u8[RIMEADDR_SIZE-1-i]);
        pl[idx++] = model->addr.u8[(RIMEADDR_SIZE-1)-i];
    }

    for(i=0;i<28;i++){
        //INFO("mu[%d]\n",i);
        pl[idx++] = model->mu[i][0];
pl[idx++] = model->mu[i][1];
pl[idx++] = model->mu[i][2];
pl[idx++] = model->mu[i][3];
pl[idx++] = model->mu[i][4];
pl[idx++] = model->mu[i][5];
pl[idx++] = model->mu[i][6];
pl[idx++] = model->mu[i][7];
}

for(i=0;i<56;i++){
    //INFO("sigma[%d]\n",i);
    pl[idx++] = model->sigma[i][0];
    pl[idx++] = model->sigma[i][1];
    pl[idx++] = model->sigma[i][2];
    pl[idx++] = model->sigma[i][3];
    pl[idx++] = model->sigma[i][4];
    pl[idx++] = model->sigma[i][5];
    pl[idx++] = model->sigma[i][6];
    pl[idx++] = model->sigma[i][7];
}

for(i=0;i<14;i++){
    //INFO("mixmat[%d]\n",i);
    pl[idx++] = model->mixmat[i][0];
    pl[idx++] = model->mixmat[i][1];
    pl[idx++] = model->mixmat[i][2];
    pl[idx++] = model->mixmat[i][3];
    pl[idx++] = model->mixmat[i][4];
    pl[idx++] = model->mixmat[i][5];
    pl[idx++] = model->mixmat[i][6];
    pl[idx++] = model->mixmat[i][7];
}

for(i=0;i<2;i++){
    //INFO("prior[%d]\n",i);
    pl[idx++] = model->prior[i][0];
    pl[idx++] = model->prior[i][1];
    pl[idx++] = model->prior[i][2];
    pl[idx++] = model->prior[i][3];
pl[idx++] = model->prior[i][4];
pl[idx++] = model->prior[i][5];
pl[idx++] = model->prior[i][6];
pl[idx++] = model->prior[i][7];
}

for(i=0;i<4;i++){
    //INFO("transmat[\%d\\n",i);
    pl[idx++] = model->transmat[i][0];
    pl[idx++] = model->transmat[i][1];
    pl[idx++] = model->transmat[i][2];
    pl[idx++] = model->transmat[i][3];
    pl[idx++] = model->transmat[i][4];
    pl[idx++] = model->transmat[i][5];
    pl[idx++] = model->transmat[i][6];
    pl[idx++] = model->transmat[i][7];
}
INFO("Serialize model done!\n");
return idx;

/**************************************************************************/
void
deserialize_payload(uint8_t *data, struct model_pkt *payload){
    memcpy(&payload->cmd_type, &data[0], 1);
    memcpy(&payload->data[0], &data[1], MAX_CTRL_PAYLOAD_SIZE);
    memcpy(&payload->seq, &data[MAX_CTRL_PAYLOAD_SIZE+1], 1);
    memcpy(&payload->timestamp, &data[MAX_CTRL_PAYLOAD_SIZE+2], 2);
    //last two bytes should be the timestamp
    return;
}

/**************************************************************************/
uint8_t
serialize_payload(struct model_pkt *data, uint8_t data_len, uint8_t *payload){
    if(data_len != 0){
        memset(payload, 0, PKT_PAYLOAD_LEN);
        memcpy(&payload[0], &data->cmd_type, 1);
memcpy(&payload[1], data->data, MAX_CTRL_PAYLOAD_SIZE);
memcpy(&payload[MAX_CTRL_PAYLOAD_SIZE+1], &data->seq, 1);
memcpy(&payload[MAX_CTRL_PAYLOAD_SIZE+2], &data->timestamp, 2);
    //last two bytes should be the timestamp
    return 1;
} else{
    return 0;
}
}

/**************************************************************************************************/
/* Here data should be serialized before saving
*/
/* We save what is pointing in model_pl */
int
save_model(uint8_t *pl, char name[12], int len, rimeaddr_t addr){
    int r, fd;
    /* Remove existing file */
    cfs_remove(name);
    /* Write new file */
    fd = cfs_open(name, CFS_WRITE | CFS_APPEND | CFS_READ);
    if(fd < 0) {
        ERROR("Err: failed to open %s\n", name);
        return -1;
    }
    INFO("Write file %s\n", name);
    r = cfs_write(fd, pl, len);
    cfs_close(fd);
    if(r != len) {
        ERROR("Err: failed to write %d bytes to %s\n",len, name);
        return -1;
    }
    /* Add model info if it already not there */
    if(!is_model_available(addr)){
        struct model_info *m = memb_alloc(&model_info_memb);
        strcpy(m->name, name);
        list_add(model_info_list, m);
    printf("%s saved\n", name);  
    }  
    return 1;  
  }
  
  /*******************************************************************************/
  int
  is_broadcast_addr(uint8_t mode, uint8_t *addr){
    int i = mode == FRAME802154_SHORTADDRMODE ? 2 : 8;
    while(i-- > 0) {
      if(addr[i] != 0xff) {
        return 0;
      }
    }
    return 1;
  }
  
  /*******************************************************************************/
  /* Obtain corresponding time entry */
  static struct timesync *
  timesync_block_from_addr(const rimeaddr_t from){
    struct timesync *t = list_head(time_offset_list);  
    while(t != NULL){
      if(rimeaddr_cmp(&t->neighbour, &from)){
        return t;
      }
      t = list_item_next(t);  
    }
    return NULL;
  }
  
  /*******************************************************************************/
  /* Obtain corresponding time offset with addr */
  static rtimer_clock_t
  get_neighbor_offset(rimeaddr_t addr){
    struct timesync *t = timesync_block_from_addr(addr);  
    if(t == NULL){
        ERROR("Err: no time-entry\n");
    }
return -1;
} else{
    return t->offset;
}

/**********************************************************/
/* Obtain clock of the parent */
static rtimer_clock_t get_parent_offset(){
    return get_neighbor_offset(parent_mac);
}

/**********************************************************/
/* Obtain the synchronized time offset of the node with sink */
/* tsync_offset = offset with parent + parent’s offset with sink */
static rtimer_clock_t get_tsync_offset(){
    rtimer_clock_t offset;
    if(node_id == 1){
        offset = 0;
    }else{
        struct timesync *t = timesync_block_from_addr(parent_mac);
        if(t == NULL){
            offset = 0;
        }else{
            offset = t->offset + t->sync_offset; //offset with neighbour + neighbour’s offset with sink
        }
    }
    synchronized_offset = offset;
    return offset;
}

/**********************************************************/
/* Obtain the synchronized time of the node with sink */
/* tsync = Local time + offset with sink */
static rtimer_clock_t get_tsync_time(){
    return RTIMER_NOW() + get_tsync_offset();
}
B.1 Contiki

1811 }
1812
1813 /**********************************************************************************/
1814 /* Save timediff between neighbors */
1815 void
1816 update_timediff(rimeaddr_t from, struct mac_ctrl_pkt *p, rtimer_clock_t timestamp){
1817  //printaddr(&from);
1818  struct timesync *t = timesync_block_from_addr(from);
1819  if(t == NULL){
1820     /* Create new entry */
1821     t = memb_alloc(&timesync_memb);
1822     t->neighbour = from;
1823     list_add(time_offset_list, t);
1824  }
1825  /* Update entry */
1826  t->offset = p->timestamp - timestamp;
1827  t->sync_offset = p->sync_offset; //offset of node with sink
1828  authority_level = p->authority_level + 1;
1829  printf("offset %ld, tsync time %u\n", t->offset, get_tsync_time());
1830  return;
1831 }
1832
1833 /**********************************************************************************/
1834 void
1835 send_time_update_now(void *ptr){
1836     mac_broadcast(MAC_COMMAND_UPDATE_TIMEDIFF, ptr, 0, 0);
1837     //INFO("time pkt sent AL %u, at %u, offset %u, drift %d\n",
1838             authority_level, now, offset, drift);
1839     printf("TU%d\n", authority_level);
1840 }
1841
1842 /**********************************************************************************/
1843 void
1844 send_time_update(void *ptr){
1845     static struct mac_ctrl_pkt data;
1846     clock_time_t waiting_time;
1847     int drift = 0;
ctimer_stop(&ct_update_timediff);

if(node_id == 1){
    tu_count++;
    if(ptr == NULL){
        /* Initial execution */
        clock_time_t passed_sync_time = (clock_time_t)
            1.0*get_tsync_time()* \ 
            CLOCK_SECOND/RTIMER_SECOND;
        /* we delay tx by subslotlen to make sure pdr update is txmt
         before this */
        waiting_time = MAC_TIME_UPDATE_TIMEOUT - passed_sync_time + SUB_SLOT_LEN;
        waiting_time = TIME_SYNC_INIT_TIMEOUT_SEC*CLOCK_SECOND -
            passed_sync_time + SUB_SLOT_LEN;
    }else{
        /* Follow up executions */
        if(tu_count < 4){
            waiting_time = TIME_SYNC_INIT_TIMEOUT_SEC*CLOCK_SECOND;
        }else{
            waiting_time = MAC_TIME_UPDATE_TIMEOUT;
            tu_count = 10;
        }
    }
    /* Consider drift wrt start of a clock second as well */
    drift = cal_drift_second();
    waiting_time = waiting_time - drift;
    INFO("TU count %d, waiting_time %lu, drift %d\n", tu_count,
            waiting_time, drift);
    ctimer_set(&ct_update_timediff, waiting_time, send_time_update,
            &ct_update_timediff);
}

/* Send time update */
data.cmd_type = MAC_COMMAND_UPDATE_TIMEDIFF;
data.authority_level = authority_level;
rtimer_clock_t offset = get_tsync_offset(); //clock offset with sink
data.sync_offset = offset;
/send time updates in round-robin
waiting_time = (node_id%TOT_NUM_NODES);
ctimer_set(&ct_tu_now, waiting_time, send_time_update_now, &data);

/* Start Duty-cycling */
if(!duty_cycling_on){
  ctimer_set(&ct_rdc_on, CLOCK_SECOND*RCD_INIT_TIMEOUT_SEC,
    start_radio_dutycycle, NULL);
  duty_cycling_on = 1;
}

if(!timer_on){
  /* Start timer for time-slotting */
  /* Slot starts when the tsync timer overflows */
  /* This gives other nodes to start the slot at the same time */
  /* t_local = t_sync - offset */
  /* start time is when t_sync = 0 */
  /* t_local = -offset */
  if(rtimer_set(&rt_slot, -get_tsync_offset(), 1, transmitter,
    NULL)) {
    PRINTF("TIMER Error #3\n");
  }else {
    timer_on = 1;
    //etimer_set(&et_slot, SLOT_LENGTH);
    INFO("TIMER Starting timer in %u rticks\n",
      -get_tsync_offset());
  }

  /* Start creating a list of future white spaces if models exist */
  /* we are at 1 second now */
  clock_time_t waiting_time = (INIT_SLOT_LIST_TIMEOUT_SEC
    -1)*CLOCK_SECOND;
  ctimer_set(&ct_req_slot, waiting_time, get_future_slots, NULL);
}
return;
/***********************************************************************************************/
void remove_recv_model_pkts(void *ptr){
    struct model_pkt *pkt_next, *pkt = list_head(recv_model_pkt_list);
    pkt_next = pkt;
    while(pkt_next){
        pkt_next = list_item_next(pkt);
        list_remove(recv_model_pkt_list, pkt);
        memb_free(&model_pkt_memb, pkt);
    }

    /* Update variable for RX next traffic model*/
    recv_byte_count = 0;
    receiving_neighbor_model = 0; /* Model received */
    have_parent_model = 1;
}

/**************************************************************/
/* Process Broadcast control data MAC */
uint8_t process_ctrl_pkt(frame802154_t frame, uint8_t len, rtimer_clock_t timestamp){
    INFO("process_ctrl_pkt\n");
    if(is_broadcast_addr(frame.fcf.dest_addr_mode, frame.dest_addr)){
        /*This is a control packet*/
        rimeaddr_t from;

        /* Copy addr of the origin */
        rimeaddr_copy(&from, (rimeaddr_t *)&frame.src_addr);

        /* Extract pkt data*/
        if(frame.payload[0] == MAC_COMMAND_UPDATE_TIMEDIFF){
            struct mac_ctrl_pkt *p = (struct mac_ctrl_pkt *)frame.payload;
            if(p->authority_level < authority_level){
                INFO("Update time\n");
                update_timediff(from, p, timestamp);
                send_time_update(NULL);
            }
        }else{
            INFO("Unknown control packet\n");
        }
    }
    return;
}
return 1;
} else if (frame.payload[0] == MAC_COMMAND_CALIBRATE_MODELS) {
  // struct mac_ctrl_pkt *p = (struct mac_ctrl_pkt *)frame.payload;
  static struct mac_ctrl_pkt pkt;
  memcpy(&pkt, frame.payload, sizeof(struct mac_ctrl_pkt));
  static struct mac_ctrl_pkt *p = &pkt;
  if (p->calibrate && p->age < age_model_calibrate) {
    /* Relay the received packet */
    calmod_sent_count = 0;
    age_model_calibrate = p->age + 1;
    clock_time_t waiting_time = 1*(node_id % 2);
    ctimer_set(&ct_calmod, waiting_time,
               relay_model_calibrate_cmd, (void *)p);
    unsigned long base_slot = p->base_slot;
    INFO("calibrate req recv new age %d\n", age_model_calibrate);
  }
  /* Temporary cease all prediction requests now */
  ctimer_stop(&ct_req_slot);
  /* Set correct slot id for own and parent models before
   * requesting predictions */
  corresponding_slot_parent = base_slot +
    (MODEL_TRANSITION_MIN_NUM_SAMPLES +
    SLOTS_PER_DATA_PERIOD);
  corresponding_slot_parent = base_slot +
    (MODEL_TRANSITION_NUM_SAMPLES +
    SLOTS_PER_DATA_PERIOD);
  if (p->model_type != change_model_flag) {
    /* We switch models: peak <-> offpeak*/
    change_model_flag = !change_model_flag;
    INFO("modch fs_len P %d, 0 %d\n",
         list_length(parent_fs_list),
         list_length(own_fs_list));
    INFO("modch %u\n", change_model_flag);
  }
#if MODEL_CALIBRATION_ON
  if (p->model_type != change_model_flag) {
    /* We switch models: peak <-> offpeak*/
    change_model_flag = !change_model_flag;
    INFO("modch fs_len P %d, 0 %d\n",
         list_length(parent_fs_list),
         list_length(own_fs_list));
    INFO("modch %u\n", change_model_flag);
  }
#endif
corresponding_slot_own = corresponding_slot_parent;
printf("corr slot %lu\n", corresponding_slot_parent);

/* Now start prediction requests from new models */
req_slot_own(NULL);
}
#endif /*MODEL_CALIBRATION_ON*/

} else if(p->calibrate && p->age == age_model_calibrate+1){
    /* We consider this as an ack for the packet we sent earlier */
    calmod_sent_count = 0;
    ctimer_stop(&ct_calmod);
    printf("calmod ack\n");
}
//free(p);
return 1;
} else if(frame.payload[0] == MAC_COMMAND_BROADCAST_MODEL){
    /* Broadcast the model to neighbours */
    process_poll(&wspmac_process);
    while(process_post(&wspmac_process, BROADCAST_MODELS_REQUEST, NULL) == PROCESS_ERR_FULL){continue;}
    return 1;
} else if(frame.payload[0] == MAC_COMMAND_ADD_MODEL){
    //INFO("Process model pkt\n");
    uint8_t pkt_data_len = MAX_CTRL_PAYLOAD_SIZE - 1;
    uint16_t begin = 0;
    uint16_t end = TOT_MODEL_LEN - 1;

    /* Add receiving neighbour's model */
    if(frame.seq != recv_seq_num){
        /* A model is being received by a neighbour */
        if(!receiving_neighbor_model){
            ctimer_set(&ct_model_pkt_recv, MODEL_RADIO_ON_DURATION, remove_recv_model_pkts, NULL);
        }
        receiving_neighbor_model = 1;
        struct model_pkt *p = memb_alloc(&model_pkt_memb);
        deserialize_payload(frame.payload, p);
//INFO("ctrl pkt seq %u, payload seq %u\n", frame.seq,
    p->seq);
recv_seq_num = frame.seq;
list_add(recv_model_pkt_list, p);
recv_byte_count += frame.payload_len - 4;

//INFO("recv_byte_count %u, tot_model_len %u\n",
    recv_byte_count, TOT_MODEL_LEN);
if(recv_byte_count != 0 && recv_byte_count >= TOT_MODEL_LEN){
    /*All data received*/
    //INFO("Extract model...\n");
    if(!model_pl){
        free(model_pl);
    }
    model_pl = malloc(TOT_MODEL_LEN);
    memset(model_pl, 0, TOT_MODEL_LEN);
    while(begin < end){
        //INFO("begin %u\n", begin);
        struct model_pkt *pkt = list_head(recv_model_pkt_list);
        memcpy(&(model_pl[begin]), pkt->data, pkt_data_len);
        list_remove(recv_model_pkt_list, pkt);
        memb_free(&model_pkt_memb, pkt);
        /* Update variables for next data block*/
        begin += pkt_data_len;
        if(begin < end && begin + pkt_data_len > end){
            pkt_data_len = end - begin + 1;
        }
    }
}

/* Save model in the Flash */
//INFO("saving recv model\n");
char name[12];
sprintf(name, "model_%02x%02x", from.u8[0], from.u8[1]);
save_model(model_pl, name, TOT_MODEL_LEN, from);
free(model_pl);

/* Update variable for RX next traffic model*/
recv_byte_count = 0;
receiving_neighbor_model = 0; /* Model received */

/* Check if we have parent model */
if(is_model_available(parent_mac)){
    have_parent_model = 1;
}
else{
    /* Retransmission detected */
    //INFO("rexmt ctrl pkt recv!\n");
    return 1;
}
else{
    /* Frame is not for us*/
    return 0;
}

*****************************************************************************
void
setup_mac_pkt(frame802154_t *params){
    INFO("setup_mac_pkt\n");
    /* init to zeros */
    memset(params, 0, sizeof(frame802154_t));
    /* Build the FCF. */
    params->fcf.security_enabled = 0;
    params->fcf.frame_pending = 0;
    params->fcf.panid_compression = 0;
    /* Insert IEEE 802.15.4 (2003) version bit. */
    params->fcf.frame_version = FRAME802154_IEEE802154_2003;
    /* Complete the addressing fields. */
/**
 * For phase 1 the addresses are all long. We’ll need a mechanism
 * in the rime attributes to tell the mac to use long or short for
 * phase 2.
 */
params->fcf.src_addr_mode = FRAME802154_LONGADDRMODE;
params->dest_pid = mac_dst_pan_id;
/* Set the source PAN ID to the global variable. */
params->src_pid = mac_src_pan_id;
/*
 * Set up the source address using only the long address mode for
 * phase 1.
 */
#if NETSTACK_CONF_BRIDGE_MODE
rimeaddr_copy((rimeaddr_t *)&params->src_addr, packetbuf_addr(PACKETBUF_ADDR_SENDER));
#else
rimeaddr_copy((rimeaddr_t *)&params->src_addr, &rimeaddr_node_addr);
#endif
return;

/******************************************************************************/
/* Broadcast ctrl data from MAC */
uint8_t
mac_broadcast(uint8_t cmd_type, void *pl, uint16_t data_len, uint8_t is_rexmt){
INFO("mac_broadcast\n");
frame802154_t params;
setup_mac_pkt(&params);
/* Build the FCF. */
params.fcf.frame_type = FRAME802154_CMDFRAME;
/* Broadcast requires short address mode. */
params.fcf.dest_addr_mode = FRAME802154_SHORTADDRMODE;
params.dest_addr[0] = 0xFF;
params.dest_addr[1] = 0xFF;

/* Distinguish model pkt and timediff pkt*/
if (cmd_type == MAC_COMMAND_UPDATE_TIMEDIFF || cmd_type ==
MAC_COMMAND_CALIBRATE_MODELS
    || cmd_type == MAC_COMMAND_BROADCAST_MODEL){
    struct mac_ctrl_pkt *data = (struct mac_ctrl_pkt *)pl;
    data->seq = 0;
    params.seq = 0;
    params.fcf.ack_required = 0;
    //packetbuf_attr(PACKETBUF_ATTR_RELIABLE);

    /* Set timestamp */
    /* Timestamp is set by the radio */
    /* Last two bytes of the payload will be used for that */
    data->timestamp = 0;
    params.payload = (uint8_t *)data;
    params.payload_len = sizeof(struct mac_ctrl_pkt);
}

} else if (cmd_type == MAC_COMMAND_ADD_MODEL){
    struct model_pkt *data = (struct model_pkt *)pl;
    if(!is_rexmt){
        /* Increment and set the data sequence number. */
        params.seq = mac_dsn++;
    }else{
        /* Retransmit*/
        params.seq = seq_num;
    }
#endif
#if MAC_REXMT_ON == 1
    params.fcf.ack_required = 1;
    //packetbuf_attr(PACKETBUF_ATTR_RELIABLE);
#else
    params.fcf.ack_required = 0;
    //packetbuf_attr(PACKETBUF_ATTR_RELIABLE);
# Contiki

```c
2170  #endif
2171  data->seq = params.seq;
2172  /* Upload payload data */
2173  params.payload = malloc(PKT_PAYLOAD_LEN);
2174  serialize_payload(data, data_len, params.payload);
2175  params.payload_len = data_len;
2176  data->timestamp = RTIMER_NOW();
2177 }
2178
2179  packetbuf_clear();
2180  packetbuf_copyfrom(params.payload, params.payload_len); //assign mac
2181  free(params.payload);
2182  uint8_t len = frame802154_hdrlen(&params);
2183  if(packetbuf_hdralloc(len)){
2184    frame802154_create(&params, packetbuf_hdrptr(), len); //assign
2185    header of params to packetbuf_hdr
2186    PRINTF("MAC-UT: %2X", params.fcf.frame_type);
2187    PRINTADDR(params.dest_addr);
2188    PRINTF("%u %u (%u)\n", len, packetbuf_datalen(),
2189                       packetbuf_totlen());
2190
2191  packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER, &rimeaddr_null);
2192  if(cmd_type == MAC_COMMAND_UPDATE_TIMEDIFF){
2193    packetbuf_set_attr(PACKETBUF_ATTR_PACKET_TYPE,
2194                        PACKETBUF_ATTR_PACKET_TYPE_TIMESTAMP);
2195  }else{
2196    packetbuf_set_attr(PACKETBUF_ATTR_PACKET_TYPE,
2197                        PACKETBUF_ATTR_PACKET_TYPE_DATA);
2198  }
2199  packetbuf_set_attr(PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS, 0);
2200  /* Look for the neighbor entry */
2201  const rimeaddr_t *addr = &rimeaddr_null;
2202  struct neighbor_queue *n = neighbor_queue_from_addr(addr,
2203                                          neighbor_list_csma);
2204  if(n == NULL) {
2205    /* Allocate a new neighbor entry */
```
n = memb_alloc(&neighbor_memb);
if(n != NULL) {
    /* Init neighbor entry */
    rimeaddr_copy((rimeaddr_t *)&n->addr, addr);
    n->transmissions = 0;
    n->collisions = 0;
    n->deferrals = 0;
    /* Init packet list for this neighbor */
    LIST_STRUCT_INIT(n, queued_packet_list);
    /* Add neighbor to the list */
    list_add(neighbor_list_csma, n);
}
send_packet_csma(packet_sent_csma, n);
//INFO("bc sent seq %u \n", params.seq);
return params.seq;
} else {
    INFO("MAC Large header %u\n", len);
    PRINTF("6MAC-UT: too large header: %u\n", len);
    return 0;
}
}

/***************************************************************************/
void
send_ack_ctrl(void *ptr){ //uint8_t dest_addr[8], uint8_t seq){
    INFO("send_ack_ctrl\n");
    frame802154_t params;
    uint8_t len;
    struct ack_msg msg = *(struct ack_msg *)ptr;
    uint8_t dest_addr = msg.src_addr.u8[0];
    uint8_t seq = msg.seq;
    setup_mac_pkt(&params);
    /* Build the FCF. */
    params.fcf.frame_type = FRAME802154_ACKFRAME;
    params.fcf.ack_required = 0;
2242  /* Increment and set the data sequence number. */
2243  params.seq = seq;
2244
2245  /* Broadcast requires short address mode. */
2246  params.fcf.dest_addr_mode = FRAME802154_LONGADDRMODE;
2247  rimeaddr_copy((rimeaddr_t *)&params.dest_addr, (rimeaddr_t *)&dest_addr);
2248
2249  packetbuf_clear();
2250  //INFO("ACK payload_len \%u\n", params.payload_len);
2251  //packetbuf_copyfrom(params.payload, params.payload_len);
2252  len = frame802154_hdrlen(&params);
2253  if(packetbuf_hdralloc(len)) {
2254    frame802154_create(&params, packetbuf_hdrptr(), len);
2255    PRINTF("6MAC-UT: \%2X", params.fcf.frame_type);
2256    PRINTADDR(params.dest_addr);
2257    PRINTF("\%u \%u (\%u)\n", len, packetbuf_datalen(),
2258      packetbuf_totlen());
2259    //INFO("b4 ack send hdr \%u, data \%u, tot \%u\n",
2260      packetbuf_hdrlen(), packetbuf_datalen(), packetbuf_totlen());
2261
2262    //packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER, (rimeaddr_t *)&params.dest_addr);
2263    packetbuf_set_attr(PACKETBUF_ATTR_PACKET_TYPE,
2264      PACKETBUF_ATTR_PACKET_TYPE_ACK);
2265    packetbuf_set_attr(PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS, 0);
2266
2267    /* Look for the neighbor entry */
2268    const rimeaddr_t *addr = &rimeaddr_null;
2269    struct neighbor_queue *n = neighbor_queue_from_addr(addr,
2270      neighbor_list_csma);
2271    if(n == NULL) {
2272      /* Allocate a new neighbor entry */
2273      n = memb_alloc(&neighbor_memb);
2274      if(n != NULL) {
2275        /* Init neighbor entry */
2276        rimeaddr_copy((rimeaddr_t *)n->addr, addr);
n->transmissions = 0;
n->collisions = 0;
n->deferrals = 0;

/* Init packet list for this neighbor */
LIST_STRUCT_INIT(n, queued_packet_list);

/* Add neighbor to the list */
list_add(neighbor_list_csma, n);

send_packet_csma(packet_sent_csma, n);

send_model_pkt(void *ptr){  //uint8_t is_rexmt){
  uint8_t is_rexmt = *(uint8_t *)ptr;
  //INFO("mpkt rexmt %d\n", is_rexmt);
  if(!is_rexmt){
    /* Reset the counter for retransmissions */
    num_rexmt_model_pkt = 0;
  }

  if(list_length(tx_model_pkt_list) != 0){
    //INFO("sending ctrl pkt at %u\n", get_tsync_time());
    struct model_pkt *data = list_head(tx_model_pkt_list);
    data->cmd_type = MAC_COMMAND_ADD_MODEL;
    seq_num = mac_broadcast(MAC_COMMAND_ADD_MODEL, data, 
  PKT_PAYLOAD_LEN, is_rexmt);

  #if MAC_REXMT_ON == 1
  /* Start timer*/
  ctimer_set(&ct_ack_timeout, MAC_TIMEOUT, retransmit_model_pkt, 
  NULL);
  #else

  #endif
}
2312     /* no model rexmt */
2313     is_rexmt = 0;
2314     ctimer_set(&ct_ack_timeout, MAC_TIMEOUT, send_model_pkt,
                 &is_rexmt);
2315
2316     /* remove sent pkt */
2317     if(data != NULL){
2318         if(data->seq == seq_num){
2319             list_remove(tx_model_pkt_list, data);
2320             memb_free(&model_pkt_memb, data);
2321         }
2322     }
2323     else{
2324         INFO("BC model done \%d\n", list_length(tx_model_pkt_list));
2325         recalibrating = 0; /*Models are recalibrated*/
2326     }
2327 }
2328 #endif
2329 }else{
2330     INFO("BC model done \%d\n", list_length(tx_model_pkt_list));
2331     recalibrating = 0; /*Models are recalibrated*/
2332 }
2333 
2334 #if MAC_REXMT_ON == 1
2335 static void
2336 retransmit_model_pkt(void *n){
2337     ctimer_stop(&ct_ack_timeout);
2338     if(num_rexmt_model_pkt > MAX_NUM_MODEL_REXMT){
2339         /* I might not have a child */
2340         /* So, stop sending model */
2341         //i_am_a_leaf_node = 1;
2342         INFO("No child is listening!\n");
2343         return;
2344     }
2345     uint8_t is_rexmt = 1;
2346     send_model_pkt(&is_rexmt);
2347     num_rexmt_model_pkt++;
2348 }
2349 #endif
2350 
2351 /*********************************************************************************/
2352 
2353 void
broadcast_own_model(void *ptr){
    if(receiving_neighbor_model){
        INFO("A model is being received\n");
        /* Do not broadcast if neighbor’s model is being received */
        if(ptr){
            ctimer_stop((struct ctimer *)ptr);
            ctimer_set((struct ctimer *)ptr, CLOCK_SECOND,
                       broadcast_own_model, ptr);
        } else{
            /* First time in the function */
            ctimer_set(&ct_send_model, CLOCK_SECOND, broadcast_own_model,
                       &ct_send_model);
        }
    } else{
        /* If radio is not already on, we switch radio on */
        /* to receive acks */
        if(!receiver_on){
            if(radio_locked) {
                /* if locker is locked, turn on radio when the locker is
                 unlocked */
                radio_lock_on = 1;
            } else{
                MAC_GET_LOCK();
                on();
                receiver_on = 1;
                MAC_RELEASE_LOCK();
                // printf("r_on bc\n");
            }
        }
    }
    /* radio off after timeout */
    ctimer_set(&ct_radio_off, MODEL_RADIO_ON_DURATION,
               switch_radio_off, &ct_radio_off);
    INFO("r_off timer set bc\n");
    INFO("BC models...\n");
    uint8_t is_rexmt = 0;
    send_model_pkt(&is_rexmt);
}
B.1 Contiki

2385    return;
2386 }
2387
2388 /*******************************************************************************/
2389 void
2390 queue_model_pkt(uint16_t begin, uint16_t data_len, uint8_t *pl){
2391     //INFO("q model pkt\n");
2392     struct model_pkt *data = memb_alloc(&model_pkt_memb);
2393     memcpy(&data->data[0], &pl[begin], data_len);
2394     list_add(tx_model_pkt_list, (void *)data);
2395     //INFO("q model pkt done\n");
2396     return;
2397 }
2398
2399 /*******************************************************************************/
2400 void
2401 split_and_queue_model(uint8_t *pl, int len){
2402     uint16_t data_len = MAX_CTRL_PAYLOAD_SIZE - 1;
2403     uint16_t pkt_end = len - 1;
2404     uint16_t begin = 0;
2405     //INFO("splitting model\n");
2406     while(begin < pkt_end){
2407         //INFO("sender begin %u, data_len %u, end %u\n", begin, data_len, 
               pkt_end);
2408         queue_model_pkt(begin, data_len, pl);
2409         /* Update variables to send next data block*/
2410         begin += data_len;
2411         if((begin < pkt_end && begin+data_len > pkt_end){
2412             data_len = pkt_end - begin + 1;
2413         }
2414     }
2415     free(pl);
2416     INFO("splitting model done\n");
2417     broadcast_own_model(NULL); /* Send first ctrl packet*/
2418     return;
2419 }
2420
2421 /*******************************************************************************/
2422 int
load_model(char name[12], void *pl, int len){
    int r, fd;
    INFO("loading model \%s\n", name);
    fd = cfs_open(name, CFS_READ);
    if(fd < 0) {
        ERROR("Err: failed to open \%s\n", name);
        return -1;
    } else {
        r = cfs_read(fd, pl, len);
        cfs_close(fd);
        if(r != len) {
            ERROR("Err: failed to load \%d bytes from \%s\n", len, name);
            return -1;
        } else {
            INFO("model \%s loaded\n", name);
            return 1;
        }
    }}

/**************************************************************/
void request_own_model(void *ptr){
    ctimer_stop(ptr);
    if(!have_own_model && !receiving_matlab_model){
        compute_model();
    } else{
        INFO("Has own model\n");
    }
    return;
}

/**************************************************************/
void compute_model(){
    printf("CTRL\n"); /* Control msg*/
    process_poll(&wspmac_process);
    ctimer_set(&ct_model_recv, 30*CLOCK_SECOND, request_own_model, &ct_model_recv);
    return;
static int is_model_available(rimeaddr_t addr){
    char name[12];
    sprintf(name, "model_%02x%02x", addr.u8[0], addr.u8[1]);
    struct model_info *model = list_head(model_info_list);
    while(model){
        if(strcmp(name, model->name) == 0){
            INFO("NB model available\n");
            return 1;
        }
        model = list_item_next(model);
    }
    return 0;
}

void broadcast_model(void *ptr){
    if(ptr == NULL){
        unsigned long passed_sync_time = 0;
        long drift = 0;
        drift = cal_drift_second();
        /* Initial execution after 30 s */
        /* adjust time and call again at (30+i) s where */
        /* is determined with the node_id and num of neighbours */
        ctimer_stop(&ct_broadcast_model);
        uint8_t rr_id = node_id % TOT_NUM_NODES;
        clock_time_t waiting_time = (rr_id + 1)*CLOCK_SECOND -
                                  passed_sync_time - drift;
        ctimer_set(&ct_broadcast_model, waiting_time, broadcast_model,
                    &ct_broadcast_model);
        printf("rr_id %lu, waiting_time %ld\n", rr_id, waiting_time, drift);
} else {
    /* Broadcast the model to neighbors */
    process_poll(&wspmac_process);
    while(process_post(&wspmac_process, BROADCAST_MODELS_REQUEST, NULL) == PROCESS_ERR_FULL)
    {continue;}
    ctimer_stop(&ct_broadcast_model);
}

/**************************************************************************/
/* Request a free slot (own model) from MATLAB */
void
req_slot_own(void *ptr)
{
    clock_time_t delay;

    /* Remove old slot entries */
    remove_old_entries(own_fs_list, 0);
    process_poll(&wspmac_process);

    delay = 0.5*CLOCK_SECOND;

    if(list_length(own_fs_list) >= MAX_NUM_SLOT_REQUESTS_PER_MODEL){
        /* Here we request a free slot from parent model */
        ctimer_set(&ct_req_slot, delay, req_slot_parent, NULL);
        //INFO("own cs %lu\n", current_slot_id);
    } else {
        /* Request prediction from own model */
        while(process_post(&wspmac_process,
                           MATLAB_REQUEST_WHITESPACE_OWN, NULL)
                   == PROCESS_ERR_FULL){continue;}

        /* Here we schedule a request to get free slots from own model
         * until the list is full
         * this will be overridden when the requested slot
         * received from Matlab */
        ctimer_set(&ct_req_slot, delay, req_slot_own, NULL);
    }
}
B.1 Contiki

```c
2535 /**************************************************************************/
2536 /* Request a free slot (parent model) from MATLAB */
2537 void
2538 req_slot_parent(void *ptr){
2539     clock_time_t delay;
2540     /* Remove old slot entries */
2541     remove_old_entries(parent_fs_list, 1);
2542     process_poll(&wspmac_process);
2543     delay = 0.5*CLOCK_SECOND;
2544
2545     if(node_id == 1){
2546         /* Node 1 does not have a parent
2547          * so, we don't request parent slots
2548          * from node 1 (sink) */
2549         //delay = CLOCK_SECOND/4;
2550         ctimer_set(&ct_req_slot, delay, req_slot_own, NULL);
2551         return;
2552     }
2553
2554     if(list_length(parent_fs_list) >= MAX_NUM_SLOT_REQUESTS_PER_MODEL){
2555         /* Now we request a slot from own model */
2556         //delay = 0;
2557         ctimer_set(&ct_req_slot, delay, req_slot_own, NULL);
2558         //INFO("parent cs %lu\n", current_slot_id);
2559     }else{
2560         /* Request prediction from parent model */
2561         while(process_post(&wspmac_process,
2562                             MATLAB_REQUEST_WHITESPACE_PARENT, ptr)
2563             == PROCESS_ERR_FULL){continue;}
2564
2565         /* Now we schedule a request for a slot from parent model
2566          * until the list is full */
2567         //delay = CLOCK_SECOND/2;
2568         ctimer_set(&ct_req_slot, delay, req_slot_parent, NULL);
2569     }
```
/** start with node 1 */
uint16_t timeout_slots = 
   (uint16_t)(node_id-1)*INTER_NODE_SLOT_LIST_DELAY_SEC* \
   1000.0/SLOT_LENGTH_MS;
/* Request slots starting from the same slot */
* Nodes do this at different times, so we need to account for that as well */
corresponding_slot_parent = current_slot_id + 
   SLOT_OFFSET_AT_LIST_INIT - timeout_slots - SLOT_OFFSET;
/* Here we need to start from the same slot so as to */
* RNG behaves uniquely */
corresponding_slot_own = corresponding_slot_parent; // +
   SLOTS_PER_DATA_PERIOD;
/* Request slots in RR order */
clock_time_t waiting_time = (node_id % TOT_NUM_NODES)*SLOT_LENGTH;
ctimer_set(&ct_req_slot, waiting_time, req_slot_parent, NULL);

/* Get a list of free slots available at the parent, */
* from MATLAB, before hand that can be utilized by */
* the application */
void
get_future_slots(void *ptr){
   clock_time_t waiting_time;
   if(have_own_model){
      /* start with node 1 */
double timeout = 1.0*(node_id-1)*INTER_NODE_SLOT_LIST_DELAY_SEC;

/* Start receiving slots from Matlab after a timeout  
 * This will make sure different nodes receive at different times  
 */
waiting_time = (clock_time_t)1.0*timeout*CLOCK_SECOND;
ctimer_set(&ct_req_slot, waiting_time, get_slot_list,
            &ct_req_slot);
    //INFO("slot offset %u, slot id %lu\n", SLOT_OFFSET_AT_LIST_INIT, 
current_slot_id);
}
}

/*******************************************************************************/
/* Save lists of pre-predictions from                                          
 * 1: parent model                                                          
 * 2: own model */                                                          
void
record_rfsi(short rfsi[MAX_NUM_PREDICTIONS_PER_REQ], uint8_t
             is_parent_list, unsigned long slot){
    struct free_slot *fs;
    struct rfsi_entry *e1, *e2;
    
    /* Check if the list is full */
    if(is_parent_list){
        remove_old_entries(parent_fs_list, 0);
            if(list_length(parent_fs_list) >= MAX_NUM_SLOT_REQUESTS_PER_MODEL){
                /* Request next prediction */
                ctimer_stop(&ct_req_slot);
                /* Request next prediction */
                ctimer_set(&ct_req_slot, 0, req_slot_own, NULL);
                return;
            }
    }
    else{
        remove_old_entries(own_fs_list, 0);
        if(list_length(own_fs_list) >= MAX_NUM_SLOT_REQUESTS_PER_MODEL){
            /* Request next prediction */
            return;
        }
    }
B.1 Contiki

/* Allocate memb for new free slot entry */
fs = memb_alloc(&free_slot_memb);
LIST_STRUCT_INIT(fs, rfsi_list);

/* record rfsi entries */
e1 = memb_alloc(&rfsi_entry_memb);
e1->rfsi = rfsi[0];
e2 = memb_alloc(&rfsi_entry_memb);
e2->rfsi = rfsi[1];

/* store a list of predicted slots */
list_add(fs->rfsi_list, e1);
list_add(fs->rfsi_list, e2);
fs->slot = slot;
if(is_parent_list){
    list_add(parent_fs_list, fs);
} else{
    list_add(own_fs_list, fs);
}

/* We only save MAX_NUM_PREDICTIONS_PER_REQ predictions per request */
if(is_parent_list){
    //INFO("predicted for %lu rfsi %d,%d: parent
        slot, rfsi[0], rfsi[1]);
    /* Update setting for free slot requests to 
       * next data packet period */
    corresponding_slot_parent += SLOTS_PER_DATA_PERIOD;
} else{
    //INFO("predicted for %lu rfsi %d,%d: own
        slot, rfsi[0], rfsi[1]);
    /* Update setting for free slot requests to 
       * next data packet period */
    corresponding_slot_own += SLOTS_PER_DATA_PERIOD;
}
}
static void transmitter(struct rtimer *rt, void *ptr){
  /* Beginning of a time slot */
  ++current_slot_id;
  /* Poll wspmac_process
   * This will make sure serial_line
   * communication works fine */
  process_poll(&wspmac_process);
  //INFO("slot \%lu at \%u\n", current_slot_id, get_tsync_time());
  if(expecting_packet){
    /* Switch on transceiver only when the flag is enabled */
    switch_radio_on(NULL);
  }
  /* Node checks if there is a drift between sync time and the slot
   * start */
  /* if so, deduct that drift from the next time period */
  long time_drift = 0;
  time_drift = cal_rtimer_drift_slot();
  rtimer_clock_t next = RTIMER_NOW() + RT_SLOT_LENGTH - time_drift;
  if(rtimer_set(&rt_slot, next, 1, transmitter, NULL)){
    //INFO("transmitter failed next \%u\n", next);
  }
  return;
}

/* This is the main process of WSP MAC */
PROCESS_THREAD(wspmac_process, ev, data){
  PROCESS_BEGIN();
  char *read_data;
  char prefix[6];
  char rfsi_recv_str_1[4], rfsi_recv_str_2[8], slot_recv_str[8];
short rfsi_recv_1, rfsi_recv_2, rfsi_recv[2];
unsigned long slot_recv;

while(1){
    read_data = (char *)data;
    if(ev == serial_line_event_message && read_data != NULL){
        INFO("ev ser\n");
        /* Receiving from MATLAB */

        /* Extract the prefix of the msg */
        sprintf(prefix, "%.3s", read_data);
        //INFO("data: %s\n", read_data);

        if(!strcmp(prefix, "PN:")){
            /* Parent node free slot
             * PN:xx:yy:slot */
            sprintf(rfsi_recv_str_1, "%.2s", &read_data[3]);
            sprintf(rfsi_recv_str_2, "%.2s", &read_data[6]);
            sprintf(slot_recv_str, "%.2s", &read_data[9]);
            rfsi_recv_1 = atoi(rfsi_recv_str_1);
            rfsi_recv_2 = atoi(rfsi_recv_str_2);
            rfsi_recv[0] = rfsi_recv_1;
            rfsi_recv[1] = rfsi_recv_2;
            slot_recv = strtol(slot_recv_str, NULL, 10);
            if(slot_recv == corresponding_slot_parent){
                ctimer_stop(&ct_matlab_timeout);
                record_rfsi(rfsi_recv, 1, slot_recv);
            }else{
            }
        }else if(!strcmp(prefix, "ON:")){
            /* Own node free slot
             * ON:xx:yy:slot */
            sprintf(rfsi_recv_str_1, "%.2s", &read_data[3]);
            sprintf(rfsi_recv_str_2, "%.2s", &read_data[6]);
            sprintf(slot_recv_str, "%.2s", &read_data[9]);
        }
    } else if(!strcmp(prefix, "ON:")){
        /* Own node free slot
         * ON:xx:yy:slot */
        sprintf(rfsi_recv_str_1, "%.2s", &read_data[3]);
        sprintf(rfsi_recv_str_2, "%.2s", &read_data[6]);
        sprintf(slot_recv_str, "%.2s", &read_data[9]);
    }
    read_data = NULL;
    data = NULL;
} else if(!strcmp(prefix, "ON:")){
    /* Own node free slot
     * ON:xx:yy:slot */
    sprintf(rfsi_recv_str_1, "%.2s", &read_data[3]);
    sprintf(rfsi_recv_str_2, "%.2s", &read_data[6]);
    sprintf(slot_recv_str, "%.2s", &read_data[9]);
}
```c
rfsi_recv_1 = atoi(rfsi_recv_str_1);
rfsi_recv_2 = atoi(rfsi_recv_str_2);
rfsi_recv[0] = rfsi_recv_1;
rfsi_recv[1] = rfsi_recv_2;
slot_recv = strtol(slot_recv_str, NULL, 10);

if(slot_recv == corresponding_slot_own){
    ctimer_stop(&ct_matlab_timeout);
    record_rfsi(rfsi_recv, 0, slot_recv);
} else{
    read_data = NULL;
data = NULL;
}

/* Extract the control prefix */
sprintf(prefix, "%.2s", read_data);
//INFO("%s\n", prefix);

/* Copy model data */
if(!strcmp(prefix, "A:")) {
    receiving_matlab_model = 1;
    memcpy(&own_model->mu[mu_count][0], &read_data[2],
        sizeof(char)*8);
    //INFO("A: %s\n", own_model->mu[mu_count]);
    mu_count++;
} else if(!strcmp(prefix, "B:")) {
    memcpy(&own_model->sigma[sigma_count][0], &read_data[2],
        sizeof(char)*8);
    //INFO("B: %s\n", own_model->sigma[sigma_count]);
sigma_count++;
} else if(!strcmp(prefix, "C:")) {
    memcpy(&own_model->mixmat[mixmat_count][0],
        &read_data[2], sizeof(char)*8);
    //INFO("C: %s\n", own_model->mixmat[mixmat_count]);
mixmat_count++;
} else if(!strcmp(prefix, "D:")) {
    memcpy(&own_model->prior[prior_count][0], &read_data[2],
        sizeof(char)*8);
```
B.1 Contiki

2788     //INFO("D: %s\n", own_model->prior[prior_count]);
2789     prior_count++;
2790 } else if(!strcmp(prefix, "E:")) {
2791     memcpy(&own_model->transmat[transmat_count][0],
2792            &read_data[2], sizeof(char)*8);
2793     //INFO("E: %s\n", own_model->transmat[transmat_count]);
2794     transmat_count++;
2795 } else if(!strcmp(prefix, "F:")) {
2796     /* All parameters are received! */
2797     have_own_model = 1;
2798     receiving_matlab_model = 0;
2799     mu_count = sigma_count = mixmat_count = prior_count =
2800            transmat_count = 0;
2801     /* Serialized received model */
2802     if(!model_pl){
2803         free(model_pl);
2804         model_pl = malloc(TOT_MODEL_LEN);
2805         memset(model_pl, 0, TOT_MODEL_LEN);
2806         serialize_model_data(model_pl, own_model);
2807         memb_free(&model_memb, own_model);
2808     } else {
2809         /* Save model in the file system */
2810         //INFO("saving own model\n");
2811         char name[12];
2812         sprintf(name, "model_%02x%02x", rimeaddr_node_addr.u8[0],
2813                 rimeaddr_node_addr.u8[1]);
2814         save_model(model_pl, name, TOT_MODEL_LEN, rimeaddr_node_addr);
2815         free(model_pl);
2816         read_data = NULL;
2817         data = NULL;
2818     }
2819     PROCESS_YIELD();
2820 } /* serial_line_event */
if(ev == MATLAB_REQUEST_COMPUTE_MODELS){
    INFO("ev compute models\n");
    /* Request MATLAB to compute models and send them */
    own_model = memb_alloc(&model_memb); /* Allocated memory for
    the own traffic model */
    rimeaddr_copy(&own_model->addr, &rimeaddr_node_addr);
    compute_model();
    PROCESS_YIELD();
}

if(ev == MATLAB_REQUEST_WHITESPACE_PARENT){
    INFO("ev req ws p\n");
    /* Send request to MATLAB */
    //INFO("MATLAB req slot at %lu\n", get_tsync_time());
    /* We are expecting a free slot for a data transmission
    * in advance! */
    /* this line is important
    * CSP:xx:y:z...z
    * x: dst_id
    * y: change model flag
    * z: slot id */
    printf("CSP:%02x:%lu:%lu\n",
            parent_mac.u8[0], change_model_flag,
            corresponding_slot_parent);
    PROCESS_YIELD();
}

if(ev == MATLAB_REQUEST_WHITESPACE_OWN){
    INFO("ev req ws o\n");
    /* Send request to MATLAB */
    //INFO("MATLAB req slot at %lu\n", get_tsync_time());
    /* We are expecting a free slot for a data transmission
    * in advance! */
    /* this line is important
    * CSO:xx:y:z...z

2855 * x: dst_id
2856 * y: change model flag
2857 * z: slot id */
2858
2859 printf("CSO:%02x:%u:%lu\n",
         rimeaddr_node_addr.u8[0], change_model_flag,
         corresponding_slot_own);
2860 PROCESS_YIELD();
2861 }
2862
2863 if(ev==SNIFF_CHANNEL_REQUEST){
2864   INFO("ev sniff\n");
2865   /* Sniff channel and compute model parameters*/
2866   //TODO
2867   PROCESS_YIELD();
2868 }
2869
2870 if(ev==BROADCAST_MODELS_REQUEST){
2871   INFO("ev bc model\n");
2872   /* Start Broadcasting models */
2873   if(have_own_model){
2874     INFO("start broadcast model\n");
2875     char name[12];
2876     sprintf(name, "model_%02x%02x", rimeaddr_node_addr.u8[0],
                rimeaddr_node_addr.u8[1]);
2877     uint8_t *bc_pl = malloc(TOT_MODEL_LEN);
2878     if(load_model(name, bc_pl, TOT_MODEL_LEN) > 0){
2879       /* now we are broadcasting the model, so the timer stops */
2880       ctimer_stop(&ct_broadcast_model);
2881       split_and_queue_model(bc_pl, TOT_MODEL_LEN);
2882       free(bc_pl);
2883     }
2884   }else{
2885     ERROR("Err: own model is not available!\n");
2886   }
2887 }else{
2888     ERROR("Err: own model is not available!\n");
2889   }
2890 PROCESS_YIELD();
B.1 Contiki

2896 } 
2897 
2898 PROCESS_YIELD(); /* For any other event */
2899 } /* while */
2900 INFO("wspmac_process ends\n");
2901 PROCESS_END();
2902 } /* wspmac_process */
2903 
2904 /**************************************************************************
2905 /* Start WSPMAC after the offset */
2906 static void start_wspmac(void *ptr){
2907 INFO("starting WSPMAC\n");
2908 ctimer_stop(ptr);
2909 // process_poll(&background_process);
2910 #if COOJA_SIM==0
2911 /* Sniffing is only done in real nodes */
2912 while(process_post(&wspmac_process, SNIFF_CHANNEL_REQUEST, NULL) ==
2913 PROCESS_ERR_FULL){continue;}
2914 #else
2915 /* Traces already exists */
2916 /* Calculate model parameters */
2917 while(process_post(&wspmac_process, MATLAB_REQUEST_COMPUTE_MODELS,
2918 NULL) == PROCESS_ERR_FULL){continue;}
2919 #endif
2920 }
2921 
2922 /**************************************************************************
2923 /* Forward execution to corresponding MAC functionality */
2924 /* based on PDR (Packet Delivery Ratio) */
2925 static void
2926 send_packet(mac_callback_t sent, void *ptr){
2927 /* Get destination node id from routing */
2928 const rimeaddr_t *dest_addr =
2929 packetbuf_addr(PACKETBUF_ADDR_RECEIVER);
2930 if(rimeaddr_cmp(dest_addr, &rimeaddr_null) ||
2931 packetbuf_attr(PACKETBUF_ATTR_PACKET_TYPE)
== PACKETBUF_ATTR_PACKET_TYPE_ACK || packetbuf_datalen() < 52){
    //adis: check this value
    /* Detect broadcast and ACK packets */
    send_packet_csma(sent, ptr);
    return;
} else{
    if(have_own_model && have_parent_model){
        /* Use white space prediction */
        send_packet_wspmac(sent, ptr);
    }else{
        send_packet_csma(sent, ptr);
        WARNING("No models available\n");
    }
    return;
}
}

/*************************************************************/
static void
input_packet(void){
    uint8_t len;
    frame802154_t frame;
    len = packetbuf_datalen(); //header + payload
    rtimer_clock_t timestamp = packetbuf_attr(PACKETBUF_ATTR_TIMESTAMP);
    //packet received at the PHY at this time
    if(frame802154_parse(packetbuf_dataptr(), len, &frame)) {
        /*Check if the received pkt is ctrl */
        if(frame.fcf.frame_type == FRAME802154_CMDFRAME){
            if(frame.fcf.ack_required){
                if(frame.payload[0] == MAC_COMMAND_ADD_MODEL &&
                    !rimeaddr_cmp((rimeaddr_t *)&frame.src_addr,
                    &parent_mac)){
                    /* Neglect other model packets */
                    /* Only send ack for model packets from the parent */
                    return;
                }
            }
        }
    //send ack
    struct ack_msg msg_data;
rimeaddr_copy(&msg_data.src_addr, (rimeaddr_t *)frame.src_addr);
msg_data.seq = frame.seq;
clock_time_t waiting_time = node_id % 2;
ctimer_set(&ct_ack, waiting_time, send_ack ctrl, (void *) &msg_data);
}

/* Process received packet*/
if(!process_ctrl_pkt(frame, len, timestamp)){
    /* If pkt cannot be processed, forward it to upper layer*/
    NETSTACK_NETWORK.input();
}
else if(frame.fcf.frame_type == FRAME802154_ACKFRAME){
    /* Ack received */
    if(frame.seq == seq_num){
        ack_recv = 1;
        /* Stop ACK timer*/
        ctimer_stop(&ct_ack_timeout);
        /* Free memory of sent ctrl pkt*/
        struct model_pkt *pkt = list_head(tx_model_pkt_list);
        if(pkt != NULL){
            if(pkt->seq == seq_num){
                list_remove(tx_model_pkt_list, pkt);
                memb_free(&model_pkt_memb, pkt);
            }
        }
    }
    if(list_length(tx_model_pkt_list) > 0){
        /* Send next ctrl pkt if any*/
        uint8_t is_rexmt = 0;
        send_model_pkt(&is_rexmt);
    }else{
        /* Ack is not for WSP*/
        NETSTACK_NETWORK.input();
    }
}else{
B.1 Contiki

```c
3003       NETSTACK_NETWORK.input();
3004   
3005 } else {
3006     ERROR("Err: 6MAC failed to parse hdr\n");
3007     PRINTF("6MAC: failed to parse hdr\n");
3008   }
3009 
3010 } /****************************************************/
3011 static int
3012 on(void){
3013   return NETSTACK_RDC.on();
3014 }
3015
3016 } /****************************************************/
3017 static int
3018 off(int keep_radio_on){
3019   return NETSTACK_RDC.off(keep_radio_on);
3020 }
3021
3022 } /****************************************************/
3023 static unsigned short
3024 channel_check_interval(void){
3025   if(NETSTACK_RDC.channel_check_interval) {
3026     return NETSTACK_RDC.channel_check_interval();
3027   }
3028   return 0;
3029 }
3030
3031 } /****************************************************/
3032 static void
3033 init(void){
3034   //cfs_coffee_format();
3035   clock_init();
3036
3037   /* Set time offset with sink to 0 */
3038   synchronized_offset = 0;
3039
3040   memb_init(&packet_memb);
```
memb_init(&metadata_memb);
memb_init(&neighbor_memb);
memb_init(&model_memb);
memb_init(&model_pkt_memb);
memb_init(&timesync_memb);
memb_init(&free_slot_memb);
memb_init(&rfsi_entry_memb);
memb_init(&model_info_memb);

mac_dsn = random_rand() % 256;

if(node_id == 1){
    /* Sink initiate time updates */
    authority_level = 0;
    /* Start periodic time synch after 1 second so that all nodes
    have boot up */
    ctimer_set(&ct_update_timediff,
                TIME_SYNC_INIT_TIMEOUT_SEC*CLOCK_SECOND, send_time_update,
                NULL);
    /* Start periodic PDR update */
    ctimer_set(&ct_update_pdr, (CLOCK_SECOND*APP_START_TIMEOUT_SEC) +
               PDR_CAL_OFFSET,
               update_pdr, NULL);
    /* for pdr calculations */
    no_recv_pkts = 0;
    pdr_sample_count = 0;
    pdr_sum = 0;
    pdr_ema_last = 0;

    have_parent_model = 1;
}

//process_start(&background_process, NULL);
process_start(&wspmac_process, NULL);

/* start wspmac after a timeout */
ctimer_set(&ct_start, WSPMAC_START_OFFSET_SEC*CLOCK_SECOND,
           start_wspmac, &ct_start);
/* start model broadcast after a timeout */
B.1 Contiki

```c
ctimer_set(&ct_broadcast_model,
    INIT_BROADCAST_MODEL_TIMEOUT_SEC*CLOCK_SECOND,
    broadcast_model, NULL);

INFO("SLOTS_PER_HOUR %lu\n", SLOTS_PER_HOUR);
```

```c
*/
const struct mac_driver wspmac_driver = {
    "WSPMAC",
    init,
    send_packet,
    input_packet,
    on,
    off,
    channel_check_interval,
};
*/
```

**B.1.2.2 LUCID Header File**

```c
#ifndef __WSPMAC_H__
#define __WSPMAC_H__

#include "net/netstack.h"
#include "net/packetbuf.h"
#include "net/queuebuf.h"
#include "net/rime/broadcast.h"
#include "net/mac/frame802154.h"
#include "sys/rtimer.h"
#include "sys/ctimer.h"
#include "sys/clock.h"
#include "sys/etimer.h"
#include "sys/process.h"
#include "sys/node-id.h"
```
#include "dev/serial-line.h"
#include "dev/leds.h"

#include "lib/random.h"
#include "lib/list.h"
#include "lib/memb.h"

#include "cfs/cfs.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/**********************************************************/
/* For debugging*/
#define DEBUG_MAC_INFO 0
#define DEBUG_MAC_WARNINGS 1
#define DEBUG_MAC_ERRORS 1

#if DEBUG_MAC_INFO
#undef DEBUG_MAC_WARNINGS
#define DEBUG_MAC_WARNINGS 1
#undef DEBUG_MAC_ERRORS
#define DEBUG_MAC_ERRORS 1
#endif

#if (DEBUG_MAC_WARNINGS && !DEBUG_MAC_INFO)
#undef DEBUG_MAC_ERRORS
#define DEBUG_MAC_ERRORS 1
#endif

#if DEBUG_MAC_INFO
#define INFO(...) printf(__VA_ARGS__)
#endif

#if (defined RIMEADDR_SIZE && RIMEADDR_SIZE == 8)
#define printaddr(addr) INFO("%02x%02x:%02x%02x:%02x%02x:%02x%02x\n",
    ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], ((uint8_t *)addr)[2],
...)
((uint8_t *)addr)[3], ((uint8_t *)addr)[4], ((uint8_t *)addr)[5],
((uint8_t *)addr)[6], ((uint8_t *)addr)[7])
#else // (defined RIMEADDR_SIZE && RIMEADDR_SIZE==2)
#define printaddr(addr) INFO("%02x%02x\n", ((uint8_t *)addr)[0],
((uint8_t *)addr)[1])
#endif
#else /*DEBUG_MAC_INFO*/
#define INFO(...) 
#define printaddr(addr)
#endif /*DEBUG_MAC_INFO*/

#if DEBUG_MAC_WARNINGS
#define WARNING(...) printf(__VA_ARGS__)
#else
#define WARNING(...) 
#endif

#if DEBUG_MAC_ERRORS
#define ERROR(...) printf(__VA_ARGS__)
#else
#define ERROR(...) 
#endif

/**************************************************************************/
#define COOJA_SIM 1 /* Simulations*/
#ifndef MODEL_CALIBRATION_ON
#define MODEL_CALIBRATION_ON 1 //define in project-conf.h
#endif

/**************************************************************************/
/* Application specific parameters */
/* These parameters should normally be defined in project-conf.h in the 
 application */
#ifndef APP_START_TIMEOUT_SEC
#define APP_START_TIMEOUT_SEC 90
#endif

#ifndef TOT_NUM_NODES
/* Here we define the network size including sink */
#define TOT_NUM_NODES 5 //5 //16 //project-conf.h
#endif

#define PDR_MUL_FACTOR 100
#define PDR_THR_MUL (PDR_THR*PDR_MUL_FACTOR)

#ifdef CC2420_CONF_SFD_TIMESTAMPS
#define CC2420_CONF_SFD_TIMESTAMPS 1 //enable PHY timestamping
#endif

/*******************************************************************************/
/* WSP Parameters */
#define SLOT_LENGTH_MS 50 //20 //40 //50 //100 //150 //adis
#define WSPMAC_START_OFFSET_SEC 5 /*WSP starts after this time period*/
#define MAC_TIME_UPDATE_TIMEOUT_SEC 300
#ifdef MODEL_TRANSITION_TIMEOUT_SEC
#define MODEL_TRANSITION_TIMEOUT_SEC 60 //we give 1min to settle with
   //the chosen model
#endif
#define MAX_NUM_SNIFF_SAMPLES 1000
/* Calculate PDR after receiving all packets from nodes */
#define PDR_CAL_OFFSET_SEC (4.0*APP_DATA_PERIOD_SEC/5)
   //(APP_DATA_PERIOD_SEC - 1)
#define PDR_CAL_OFFSET (PDR_CAL_OFFSET_SEC*CLOCK_SECOND)
   //(2.5*CLOCK_SECOND)
#define MAX_CTRL_PAYLOAD_SIZE 96
#define SUB_SLOT_LEN_MS 8.512
#define INIT_RADIO_ON_DURATION_SEC 10
#define INIT.Broadcast.MODEL_TIMEOUT_SEC 30
#define MAC_TIMEOUT_MS 20 //timeout for retransmissions
#define MAX_NUM_NEIGHBORS 3
/* Configure how free slots are stored beforehand */
#define MAX_NUM_SLOT_REQUESTS_PER_MODEL 3 /* Store slots for data
   //packets */
#define MAX_NUM_PREDICTIONS_PER_REQ 2
/* MAC transmissions */
#define WSPMAC_MAX_MAC_TRANSMISSIONS 1
#define WSPMAC_MAX_MAC_DEFERRED 3
B.1 Contiki

124 /***********************************************************************************/
125 /*CSMA Parameters*/
126 #ifndef CSMA_MAX_MAC_TRANSMISSIONS
127 #ifdef CSMA_CONF_MAX_MAC_TRANSMISSIONS
128 #define CSMA_MAX_MAC_TRANSMISSIONS CSMA_CONF_MAX_MAC_TRANSMISSIONS
129#else
130 #define CSMA_MAX_MAC_TRANSMISSIONS 1
131 #endif /* CSMA_CONF_MAX_MAC_TRANSMISSIONS */
132 #endif /* CSMA_MAX_MAC_TRANSMISSIONS */
133
134 #if CSMA_MAX_MAC_TRANSMISSIONS < 1
135 #error CSMA_CONF_MAX_MAC_TRANSMISSIONS must be at least 1.
136 #error Change CSMA_CONF_MAX_MAC_TRANSMISSIONS in contiki-conf.h or in your Makefile.
137 #endif /* CSMA_CONF_MAX_MAC_TRANSMISSIONS < 1 */
138
139 /* The maximum number of co-existing neighbor queues */
140 #ifndef CSMA_CONF_MAX_NEIGHBOR_QUEUES
141 #define CSMA_MAX_NEIGHBOR_QUEUES CSMA_CONF_MAX_NEIGHBOR_QUEUES
142#else
143 #define CSMA_MAX_NEIGHBOR_QUEUES 2
144 #endif /* CSMA_CONF_MAX_NEIGHBOR_QUEUES */
145
146 #define MAX_QUEUED_PACKETS QUEUEBUF_NUM
147
148 #ifndef APP_DATA_PERIOD_SEC
149 #define APP_DATA_PERIOD_SEC 10
150 #define APP_DATA_PERIOD (APP_DATA_PERIOD_SEC*CLOCK_SECOND)
151 #endif
152 #ifndef EMA_BELOW_THR_CNT_MAX
153 #define EMA_BELOW_THR_CNT_MAX 5
154 #endif
155 #ifndef PDR_THR
156 #define PDR_THR 93
157 #endif
158
159 /***********************************************************************************/
160 /* Processes */
161 //PROCESS(background_process, "Background Process");
PROCESS(wspmac_process, "WSPMAC Process");

/* Data structures */
struct neighbor_queue;
struct qbuf_metadata;
struct traffic_model;
struct model_info;
struct model_pkt;
struct mac_ctrl_pkt;
struct timesync;

/* Events */
enum{
    SNIFF_CHANNEL_REQUEST = 0x70, //112, /*start channel sniffing*/
    COMPUTE_MODELS_REQUEST, /*Compute models*/
    BROADCAST_MODELS_REQUEST, /*Broadcast models*/
    MATLAB_REQUEST_COMPUTE_MODELS, /*Calculate models*/
    MATLAB_REQUEST_WHITESPACE_PARENT, /*Req WS of parent model*/
    MATLAB_REQUEST_WHITESPACE_own, /*Req WS of own model*/
    WSPMAC_RESET_POINTERS /*Reset pointers that are associated with prediction. this is triggered when timeout occurred*/
};

/* MAC commands */
enum{
    MAC_ASSOCIATION_REQ = 0x01,
    MAC_ASSOCIATION_RES,
    MAC_DISASSOCIATION_NOTIFY,
    MAC_DATA_REQ,
    MAC_PAN_ID_CONFLICT_NOTIFICATION,
    MAC_ORPHAN_NOTIFICATION,
    MAC_BEACON_REQ,
    MAC_COORDINATOR_REALIGNMENT,
    MAC_GTS_REQ,
    MAC_COMMAND_ADD_MODEL, /* Add model */
    MAC_COMMAND_BROADCAST_MODEL, /* Re-broadcast model */
    MAC_COMMAND_UPDATE_TIMEDIFF, /* Update timediff */
    MAC_COMMAND_UPDATE_PDR /* Update PDR from sink */
    MAC_COMMAND_CALIBRATE_MODELS /* Calibrate traffic models */
B.1 Contiki

243

*/ Periodic packet types */
enum{
    APPLICATION_DATA_PKT = 0x01,
    //NETWORK_ND_CTRL_PKT,
    MAC_PDR_UPDATE_PKT,
    MAC_TIME_UPDATE_PKT,
    MAC_MODEL_PKT
};

*/ Constants */
#define INIT_SLOT_LIST_TIMEOUT_SEC 40
#define INTER_NODE_SLOT_LIST_DELAY_SEC 5
#define SINGLE_SLOT_REQ_DELAY_SEC 1.5
#define WSPMAC_MAX_CTRL_PKTS 9
#define WSPMAC_MAX_NEIGHBOR_MODELS 1
#define SUB_SLOT_LEN 2
#define RADIO_ON_DURATION SUB_SLOT_LEN
#define SLOT_LENGTH (1.0*SLOT_LENGTH_MS*CLOCK_SECOND/1000.0)
#define RT_SLOT_LENGTH
    (rtimer_clock_t)(1.0*SLOT_LENGTH_MS*RTIMER_SECOND/1000.0)
#define MAC_TIME_UPDATE_TIMEOUT
    (CLOCK_SECOND*MAC_TIME_UPDATE_TIMEOUT_SEC)
#define RT_MAC_TIME_UPDATE_TIMEOUT
    (RTIMER_SECOND*MAC_TIME_UPDATE_TIMEOUT_SEC)
#define MAC_PDR_UPDATE_TIMEOUT (MAC_PDR_UPDATE_TIMEOUT_SEC*CLOCK_SECOND)
#define RT_MAC_PDR_UPDATE_TIMEOUT
    (MAC_PDR_UPDATE_TIMEOUT_SEC*RTIMER_SECOND)
#define TOT_MODEL_LEN sizeof(struct traffic_model)
#define PKT_PAYLOAD_LEN (MAX_CTRL_PAYLOAD_SIZE + 2 +
    sizeof(rtimer_clock_t))
#define MAC_TIMEOUT CLOCK_SECOND*(MAC_TIMEOUT_MS/1000.0)
#define MATLAB_RESPONSE_TIMEOUT (SLOT_LENGTH*10) //waiting time until
    matlab responds with channel state
#define RADIO_OFF_TIMEOUT_WHEN_RECEIVING (4*CLOCK_SECOND/1000.0)
#define MODEL_RADIO_ON_DURATION (0.2*CLOCK_SECOND)
#define SLOTS_PER_DATA_PERIOD (unsigned
    long)(APP_DATA_PERIOD_SEC/(1.0*SLOT_LENGTH_MS*1e-3))
#define MAX_NUM_PREDICTIONS
(2*MAX_NUM_SLOT_REQUESTS_PER_MODEL*MAX_NUM_PREDICTIONS_PER_REQ)
#define RADIO_ON_GUARD_PERIOD 1
#define MIN_RESIDUAL_TIME (SUB_SLOT_LEN + RADIO_ON_GUARD_PERIOD)
#define RT_THR_SLOT (rtimer_clock_t)(3.0*RT_SLOT_LENGTH/5.0)
#define APP_DRIFT 0
#define TIME_SYNC_INIT_TIMEOUT_SEC 2
#define RCD_INIT_TIMEOUT_SEC 4
#define N_SUBSLOTS_PER_SLOT (SLOT_LENGTH/SUB_SLOT_LEN)
#define SLOTS_PER_HOUR (unsigned long)(1.0*3600/(SLOT_LENGTH_MS*1e-3))

/* how many slots lag before the first data transmission from slot_list
init at 40 seconds */
#define SLOT_OFFSET_AT_LIST_INIT (uint16_t)((APP_START_TIMEOUT_SEC -
INIT_SLOT_LIST_TIMEOUT_SEC)/(1.0*SLOT_LENGTH_MS*1e-3))

/* offset at each slot req after pkt transmission */
#define SLOT_OFFSET_PERIODIC (uint16_t)(60.0/(1.0*SLOT_LENGTH_MS*1e-3))

/* Sliding window PDR calculations*/
#define MIN_NUM_PDR_SAMPLES_PER_WINDOW 10 /* min N */
#define MAX_NUM_DATA_PKTS_PER_SESSION (TOT_NUM_NODES - 1) /*except sink */

#ifndef NUM_PDR_SAMPLES_PER_WINDOW
#define NUM_PDR_SAMPLES_PER_WINDOW (MIN_NUM_PDR_SAMPLES_PER_WINDOW)
#endif

#define NUM_PDR_SAMPLES_PER_WINDOW (MIN_NUM_PDR_SAMPLES_PER_WINDOW) /*
N */
#endif

#ifndef MODEL_TRANSITION_MIN_NUM_SAMPLES
#define MODEL_TRANSITION_MIN_NUM_SAMPLES 3
#endif

#if (MODEL_TRANSITION_TIMEOUT_SEC/APP_DATA_PERIOD_SEC <
    MODEL_TRANSITION_MIN_NUM_SAMPLES)
#define MODEL_TRANSITION_TIMEOUT
    (MODEL_TRANSITION_MIN_NUM_SAMPLES*APP_DATA_PERIOD_SEC)*CLOCK_SECOND
#define MODEL_TRANSITION_NUM_SAMPLES (MODEL_TRANSITION_MIN_NUM_SAMPLES)
#else
#define MODEL_TRANSITION_TIMEOUT ((MODEL_TRANSITION_TIMEOUT_SEC -
    APP_DATA_PERIOD_SEC)*CLOCK_SECOND)
#define MODEL_TRANSITION_NUM_SAMPLES ((MODEL_TRANSITION_TIMEOUT_SEC -
    APP_DATA_PERIOD_SEC)/APP_DATA_PERIOD_SEC)
#endif

/**************************************************
 */
/* Definition of data structures */
/* Packet metadata */
struct qbuf_metadata {
    mac_callback_t sent;
    void *cptr;
    uint8_t max_transmissions;
    short relative_free_slot_id;
};

/* Every neighbor has its own packet queue */
struct neighbor_queue {
    struct neighbor_queue *next;
    rimeaddr_t addr;
    struct ctimer transmit_timer;
    uint8_t transmissions;
    uint8_t collisions, deferrals;
    LIST_STRUCT(queued_packet_list);
};

/* Store timediff between neighbours */
struct timesync{
    struct timesync *next;
```c
  rimeaddr_t neighbour;
  long offset; //clock offset with neighbour
  long sync_offset; //neighbour’s clock offset with sink
};

/* Payload of model pkt*/
struct model_pkt{
  struct model_pkt *next;
  uint8_t cmd_type;
  uint8_t data[MAX_CTRL_PAYLOAD_SIZE];
  uint8_t seq;
  //uint16_t no_recv_pkts; //for pdr update
  //uint8_t age; //for pdr pkt relay: 0 -> originator
  uint16_t timestamp;
};
//__attribute__((packed)); //stop padding to align 4 bytes

/* MAC control packet payload */
struct mac_ctrl_pkt{
  uint8_t cmd_type;
  uint8_t seq;
  unsigned long base_slot; //for pdr update
  uint8_t calibrate; //true if pdr < thr
  uint8_t model_type; //model type: 0 or 1
  uint8_t age; //for model calibrate pkt relay: 0 -> originator
  uint8_t authority_level; //for time updates. 0 -> originator
  long sync_offset; //clock offset with sink
  uint16_t timestamp; //must be the last two bytes
};

/* For controlling radio */
struct duty_cycle{
  struct ctimer *ct;
  //struct rtimer *rt;
  clock_time_t duration; //radio on duration
  uint8_t type;
  char msg[4];
  uint8_t count;
  uint8_t num_neighbors;
```
For saving predicted slots */
struct rfsi_entry{
    struct rfsi_entry *next;
    short rfsi; /* Relative free slot id */
};
struct free_slot{
    struct free_slot *next;
    unsigned long slot;
    LIST_STRUCT(rfsi_list); /* Multiple predictions: Relative free slot id */
};

/******************************************************************************/
/* Functions */
static int on(void);
static int off(int keep_radio_on);
static void packet_sent_csma(void *ptr, int status, int num_transmissions);
static void transmit_packet_list_csma(void *ptr);
static void start_wspmac(void *ptr);
static void packet_sent_wspmac(void *ptr, int status, int num_transmissions);
static void transmit_packet_list_wspmac(void *ptr);
static double get_pdr_ema();
static int is_model_available(rimeaddr_t addr);
static rtimer_clock_t get_tsync_offset();
static rtimer_clock_t get_tsync_time();

/******************************************************************************/
/* Other variables */
/* White space prediction */
#define SLOT_OFFSET 10 //to align with our prediction db
unsigned long current_slot_id = SLOT_OFFSET;
static struct traffic_model *own_model;

/* Broadcast ctrl data */
uint16_t recv_byte_count = 0;
uint8_t seq_num;
uint8_t recv_seq_num;
uint8_t age_model_calibrate = 100; // for model calibrate pkt relay. 0 -> originator(sink)
uint8_t authority_level = 100; //for time updates. 0->originator (sink)
static struct ctimer ct_start, ct_model_recv, ct_ack_timeout,
    ct_update_timediff, ct_update_pdr;

/** 
 \brief The sequence number (0x00 - 0xff) added to the transmitted 
 * data or MAC command frame. The default is a random value within 
 * the range.
 */
static uint8_t mac_dsn;

/** 
 \brief The 16-bit identifier of the PAN on which the device is 
 * sending to. If this value is 0xffff, the device is not 
 * associated.
 */
static uint16_t mac_dst_pan_id = IEEE802154_PANID;

/** 
 \brief The 16-bit identifier of the PAN on which the device is 
 * operating. If this value is 0xffff, the device is not 
 * associated.
 */
static uint16_t mac_src_pan_id = IEEE802154_PANID;

extern const struct mac_driver wspmac_driver;
const struct mac_driver *wspmac_init(const struct mac_driver *r);

B.1.3 Noise Measuring Tools

The following tools were used to measure noise from unknown sources and record their timestamps.
B.1 Contiki

B.1.3.1 Scanner

```c
#include "contiki.h"
#include "net/rime.h"
#include "net/netstack.h"

#include "cc2420.h"
#include "cc2420_const.h"

#include "dev/spi.h"
#include "dev/leds.h"
#include "dev/serial-line.h"
#include "dev/button-sensor.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <radio-sensor.h>

/*-----------------------------------------------*/
/* Handle serial-line commands */
static uint8_t
handle_command(char* data)
{
    uint8_t len = strlen(data);
    if(len <= 0){
        leds_on(LEDS_ALL);
        printf("#Error. Wrong input (length < 0). Enter:\"stop\" or \"start\"\n");
        leds_off(LEDS_ALL);
        leds_on(LEDS_BLUE);
    }
    else if(!strcmp(data, "start")){
        leds_off(LEDS_ALL);
        leds_on(LEDS_GREEN);
        printf("#Scan\n");
        return 1;
    }
}
```
else if(!strcmp(data, "stop")){
  leds_off(LEDS_ALL);
  printf("#Stop\n");
  return 2;
}
return 0;
}

PROCESS(wifi_timestamper_process, "WiFi Timestamper");
PROCESS(scanner_process, "Scanner");
AUTOSTART_PROCESSES(&wifi_timestamper_process);

/*-----------------------------*/
PROCESS_THREAD(scanner_process, ev, data){
  PROCESS_BEGIN();
  /*start wifi timestamping*/
  while(1){
    PROCESS_PAUSE();
    /*detecting interference*/
    if(!(CC2420_SFD_IS_1 && CC2420_CCA_IS_1){
      /*interference*/
      printf("1\n");
    }
    /*stopping criteria*/
    if(ev == serial_line_event_message) {
      uint8_t ret_value = handle_command((char*)data);
      if(ret_value==2){
        break;
      }
    }
  }
  PROCESS_END();
}

/*-----------------------------*/
PROCESS_THREAD(wifi_timestamper_process, ev, data){
  PROCESS_BEGIN();
  cc2420_on();
  /*waiting until start command from the pc*/
  while(1){
    PROCESS_WAIT_EVENT();
  }
if(ev == serial_line_event_message) {
    uint8_t ret_value = handle_command((char*)data);
    if (ret_value==1){
        process_start(&scanner_process, NULL);
    }
    else if(ret_value==2){
        process_exit(&scanner_process);
    }
}

B.1.3.2 Timestamper

import serial
import datetime
import threading
import sys, os
import subprocess
import time

#definition of threading events
event_close = threading.Event()
event_save_wsn = threading.Event()
event_close.clear()
event_save_wsn.clear()

# Stopping criteria
def timer_expired():
    print("")
    print("Sniffing was done successfully!")
timer.cancel()
event_close.set()
cmd = "pkill tcpdump"
print(cmd)
os.system(cmd)

# Timer to stop sniffing
timer = threading.Timer(2*3600, timer_expired)
timer.start()

class ManageExperiment(threading.Thread):
    """
    Collect sensor data over usb port and add timestamp
    usage:
        me = ManageExperiment(port, path, ch, expType)
        me.setDeamon(True)
        me.start()
    Make sure that python "pyserial" library is installed.
    """
    def __init__(self, **kwargs):
        self.port = kwargs.get('port', '60001')
        self.ch = kwargs.get('ch', '11')
        self.path = kwargs.get('path', '.*')
        self.baudrate = kwargs.get('baudrate', 115200)
        self.expType = kwargs.get('expType', 'wsn')
        self.fname_wsn = "Ch{}.log".format(self.ch)
        self.serial = None
        self.proc = None
        threading.Thread.__init__(self)

    def open_serial(self):
        try:
            self.serial = serial.Serial(port=self.port,
                                         baudrate=self.baudrate)
            print("Serial connection for WSN channel {} open".
                  format(self.ch))
        except (serial.SerialException, serial.SerialTimeoutException,
                ValueError, IOError, OSError) as e:
            print('Error opening port: %s' % (self.port,))
            print('The error was: %s' % (e.args,))
sys.exit(1)
def close_serial(self):
    self.serial.close()

def run(self):
    # Handle WSN nodes
    if self.expType == 'wsn':
        counter = 0
        log = open("{}/{}").format(self.path, self.fname_wsn), 'w')
        log_str = "Counter\tTimestamp\n"
        log.write(log_str)
        self.open_serial()
        self.serial.flushInput()
        self.serial.flushOutput()

        while True:
            if self.serial.isOpen():
                try:
                    line = self.serial.readline()
                    time_now = datetime.datetime.now()
                    if line != "":
                        line_split = line.split()
                        if '#' not in line and len(line_split)==1:
                            counter += 1
                            log_str = "{}\t{}").format(counter,
                                    time_now.strftime("%Y-%m-%d_%H%M%S.%f"))
                            log.write(log_str)
                        line = ""
                except serial.SerialException as e:
                    print('SerialException: The error was %s' %
                          (e.args,))
                    self.close_serial()
                else:
                    self.open_serial()

                if event_close.isSet():
                    cmd = "echo "stop" > {}").format(self.port)
                    print(cmd)
                    os.system(cmd)
print("Closing port {} for channel 
    {}.format(self.port, self.ch))

    self.close_serial()
    log.close()
    break

elif event_save_wsn.isSet():
    str_time_now = datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")
    print("Coping files at {}.format(str_time_now))
    os.system("mkdir {}/{}").format(self.path, str_time_now))
    os.system("cp -r {}/{} {}/{}").format(self.path, self.fname_wsn, self.path, str_time_now))
    event_save_wsn.clear()

    def main(self, port, ch):
        ser = ManageExperiment(port=port, ch=ch, path=self.workingDirWsn)
        ser.setDaemon(True)
        ser.start()

        if __name__ == "__main__":
            """Specify the serial port that the node is connected to and the 
            channel it is scanning on""
            main(port='60001', ch=11)

#=================================================================================
#EOF

B.2 MATLAB

B.2.1 Starting Script

The following code was used to create TCP/IP connections between MATLAB and the emulated nodes in Cooja.

```matlab
% Read inputs from dummy file
fid = fopen('matlab_parameters.csv');
```
params = textscan(fid, '%s %s %s %s', 'delimiter', ',', 'EmptyValue', -Inf);
fclose(fid);
env = params{1,1};
env = env{1,1};
traffic_type = params{1,2};
traffic_type = traffic_type{1,1};
wsn_app_period_sec = params{1,3};
wsn_app_period_sec = str2num(wsn_app_period_sec{1,1});
pdr_timeout = params{1,4};
pdr_timeout = str2num(pdr_timeout{1,1});

% Get wsn data period from project-conf.h
fid = fopen('../wsp-mac/apps/app/project-conf.h');
A = textscan(fid, '%s', 'delimiter', '
');
fclose(fid);
[m, n] = size(A{1,1});
for i = 1:m
    line = A{1,1}{i,1};
    line = strsplit(line, ' ');
    [k, n] = size(line);
    if n >=3 && strcmp(line(2), 'NUM_PDR_SAMPLES_PER_WINDOW')
        val = line(3);
        pdr_window_size = str2num(val{1,1});
        %break;
    end
end

% Application parameters
n_nodes = 5;

% Other parameters
day_id = 1;
hour_id = 7;
wsn_ch = 18;

txt = sprintf('%s, %d, period-%d sec, pdr_timeout %d samples, pdr window %d samples, d-%d, h-%d
', ...
env, traffic_type, wsn_app_period_sec, pdr_timeout, pdr_window_size, day_id, hour_id);
disp(txt);

% Socket configurations
port = 60000; host_type = 'client';
%port = 1234; host_type = 'server';

% Slot
slot_len_sec = 0.05; %seconds

% Seed
seed = 227;

% Number of predictions per request
max_slot_predictions = 2;

% Open sockets for motes and waiting for requests
fprintf('Start listening...
');
global config;
global predictions_db;

% load prediction database
fname = ['predictions_db/predictions_db_', env, '_ch-', num2str(wsn_ch), '.mat'];
try
load(fname);
catch
fprintf('s loading error
', fname);
end

% Sim start time
started_at = datestr(datetime, 'yyyyymmd_HHMMSS');
display(started_at);

for node_id = 1:n_nodes
% Find node type: this depends on the scenario and location of 2 jammers
nodetype = node2type(node_id);
% file to write dbg msgs
fid = fopen([started_at, '_node_', num2str(node_id), '.csv'], 'w');
fprintf(fid, txt);

% Corresponding model information
model_info = struct(
    'env', env, ...
    'wsn_ch', wsn_ch, ...
    'wsn_app_period_sec', wsn_app_period_sec, ...
    'day_id', day_id, ...
    'hour_id', hour_id, ...
    'traffic_type', traffic_type, ...
    'nodetype', nodetype, ...
    'node_id', node_id, ...
    'slot_len_sec', slot_len_sec, ...
    'max_slot_predictions', max_slot_predictions ... \#continuous predictions per req.
);

% file_id to save dbg msg
config(node_id).fid = fid;
config(node_id).last_slot_id = 0;

% gmmhmm models global
gmmhmm = load_gmmhmm_model(model_info);

% Create socket and wait for requests from cooja %
t = tcpip('localhost', port+node_id, 'NetworkRole', host_type);
t.BytesAvailableFcn = {@coojaCallback_pre_predictions, gmmhmm, model_info};
open_connections{node_id} = t;
fclose(t);
end

clear t slot_data node_id fid
B.2.2 Cooja Callback Function

The following callback function is triggered when Cooja sends data to MATLAB.

```matlab
function [] = coojaCallback_pre_predictions(data_buffer, event, gmmhmm, model_info)
    global config;
    global predictions_db;

    node_id = model_info.node_id;
    wsn_app_period_sec = model_info.wsn_app_period_sec;
    slot_len_sec = model_info.slot_len_sec;
    traffic_type = model_info.traffic_type;
    day_id = model_info.day_id;
    hour_id = model_info.hour_id;

    if strcmp(traffic_type, 'peak')
        type_id = 1;
    else
        type_id = 2;
    end

    app_period_index = dataperiod2index(wsn_app_period_sec);
    if app_period_index == -1
        fprintf('check wsn_app_period_sec\n');
        return;
    end

    T = wsn_app_period_sec/slot_len_sec;
    init_slot = 71; %81;

    data_read = fscanf(data_buffer);

    if length(data_read) < 4
        fprintf('Data not recognized!: %s\n', data_read);
    elseif strcmp(data_read(1:4), 'CTRL')
        % Save to file
        fprintf(config(node_id).fid, '%s', data_read);
    end
```

% Own model request
% Send own model to node
send_model(data_buffer, gmmhmm, node_id);
elseif strcmp(data_read(1:4), 'CSP: ')
    
    % Save to file
    fprintf(config(node_id).fid, '%s', data_read);
    t = datestr(now, '"HH:MM:SS.FFF'');
    fprintf(config(node_id).fid, '%s\t', [t]);
    fprintf(config(node_id).fid, '%s', [data_read]);

    % Parent free slot request
    % Send a predicted slot to the node
    % CSP: xx:y:z -> xx=node_id, y=flag:change model (if 1 -> change model), z=csi
    dst_node = str2double(data_read(5:6));
    model_change_flag = str2double(data_read(8));
    current_slot_id = str2double(data_read(10:end-1));

    if(isnan(dst_node) || dst_node <= 0)
        fprintf('No valid dest node id!\n');
        fprintf(data_buffer, 'PN:%02d=-1\n', node_id);
        fprintf(config(node_id).fid, 'PN:%02d=-1\n', node_id);
        return;
    end

    if current_slot_id > 864000
        disp('Slot id is out of bound!');
        fprintf(config(node_id).fid, 'Slot %d is out of bound!\n',
            current_slot_id);
        return
    end

    % Select correct traffic model: peak or offpeak
    if model_change_flag == 1
        if type_id == 1
            type_id = 2;
        else
            type_id = 1;
        end
end

% Map node to node type
dst_nodetype = node2type(dst_node);

% find correct time based on received slot
corresponding_slot_in_current_hour = mod(current_slot_id, 
    floor(3600/slot_len_sec));

hour_id = hour_id + 
    floor(current_slot_id/floor(3600/slot_len_sec));

day_id = day_id + floor(hour_id/24);

slot_index = slot2index(corresponding_slot_in_current_hour, 
    init_slot, T);

try
    % Find white space 
    rfsi = 
        predictions_db(dst_nodetype).period(app_period_index... 
            ).type(type_id).day(day_id).hour(hour_id... 
            ).slot(slot_index).rfsi;

    fprintf(data_buffer, 'PN:%02d:%02d:%d\n', [rfsi(1), rfsi(2), 
        current_slot_id]);

t = datestr(now, 'HH:MM:SS.FFF');

    fprintf(config(node_id).fid, '%s\t', [t]);

    fprintf(config(node_id).fid, 'PN:%02d:%02d:%d\n', [rfsi(1), 
        rfsi(2), current_slot_id]);

    catch ME1
        fprintf('No prediction (PN) for slot %d in d-%d h-%d slot %d, index %d\n',...
            [current_slot_id, day_id, hour_id, 
            corresponding_slot_in_current_hour, slot_index]);

        fprintf(config(node_id).fid, 'No prediction (PN) for slot %d in d-%d h-%d slot %d, index %d\n',...
            [current_slot_id, day_id, hour_id, 
            corresponding_slot_in_current_hour, slot_index]);
    end

elseif strcmp(data_read(1:4), 'CSO:')
% Save to file
fprintf(config(node_id).fid, '\n%s', data_read);
t = datestr(now, 'HH:MM:SS.FFF');
fprintf(config(node_id).fid, '\n%s\t', [t]);
fprintf(config(node_id).fid, '%s', [data_read]);

% Own node free slot request
% Send a predicted slot to the node
% CSO:xx:y:z -> xx=node_id, y=flag:change model (if 1 -> change model), z=csi
dst_node = str2double(data_read(5:6));
model_change_flag = str2double(data_read(8));
current_slot_id = str2double(data_read(10:end-1));

if isnan(dst_node) || dst_node <= 0
    fprintf('No valid dest node id!\n');
    fprintf(data_buffer, 'ON:%02d:-1\n', node_id);
    fprintf(config(node_id).fid, 'ON:%02d:-1\n', node_id);
    return;
end

if current_slot_id > 864000
    disp('Slot id is out of bound!');
    fprintf(config(node_id).fid, 'Slot %d is out of bound!\n',
            current_slot_id);
    return;
end

% Select correct traffic model: peak or offpeak
if model_change_flag == 1
    if type_id == 1
        type_id = 2;
    else
        type_id = 1;
    end
end

% Map node to node type
dst_nodetype = node2type(dst_node);
B.2 MATLAB

% find correct time based on received slot
 corresponding_slot_in_current_hour = mod(current_slot_id, floor(3600/slot_len_sec));
 hour_id = hour_id + floor(current_slot_id/floor(3600/slot_len_sec));
 day_id = day_id + floor(hour_id/24);
 slot_index = slot2index(corresponding_slot_in_current_hour, init_slot, T);

try
    % Find white space 
    rfsi = predictions_db(dst_nodetype).period(app_period_index... 
        ).type(type_id).day(day_id).hour(hour_id...
        ).slot(slot_index).rfsi;

    fprintf(data_buffer, 'ON:%02d:%02d:%02d\n', [rfsi(1), rfsi(2), current_slot_id]);
    t = datestr(now, 'HH:MM:SS.FFF');
    fprintf(config(node_id).fid, 's\n', [t]);
    fprintf(config(node_id).fid, 'ON:%02d:%02d:%02d\n', [rfsi(1), rfsi(2), current_slot_id]);

catch ME2
    fprintf('No prediction (ON) for slot %d in d-%d h-%d slot %d, index %d\n', ...
            [current_slot_id, day_id, hour_id, 
                corresponding_slot_in_current_hour, slot_index]);
    fprintf(config(node_id).fid, 'No prediction (ON) for slot %d in d-%d h-%d slot %d, index %d\n', ...
            [current_slot_id, day_id, hour_id, 
                corresponding_slot_in_current_hour, slot_index]);

end

else
    disp(data_read);
end
B.2.3 Train White Space Model

The white space models used in this work were trained using a toolbox from [122, 121], for which the following code was used.

```matlab
%% Script to train models
slot_len_ms = 50;
slot_len_sec = slot_len_ms*1e-3;
tot_duration_sec = 3600;
day_id = 1;
hour_id = 10;
tot_duration = floor(tot_duration_sec/slot_len_sec);
start_clock_time = datetime('2018-01-08 07:00:00');
end_clock_time = datetime('2018-01-09 07:00:00');
[y, m, d] = ymd(start_clock_time);
test_start_clock_time = datetime(y, m, d + day_id - 1) +
    hours(hour_id); %hours(start_hour + hour_id - 1);
test_end_clock_time = test_start_clock_time +
    hours(tot_duration_sec/3600);
if (test_start_clock_time < start_clock_time) || (test_end_clock_time >
    end_clock_time)
    return;
end
cliping_start_slot_sec = seconds(test_start_clock_time - ...
    start_clock_time);
cliping_start_slot = cliping_start_slot_sec/slot_len_sec + 1;
for env = {'office', 'home'}
    for type = {'peak', 'offpeak'}
        for nodetype = 1:3
            fprintf('training model for %s, %s, ntype %d
', env{1},
                type{1}, nodetype);
            fname = ['slotted_data/slot_data-', num2str(slot_len_ms), ...
                'ms_', env{1}, '_' , type{1}, '_nodetype', ...
```
num2str(nodetype), ’.mat’);
try
    fprintf(’%s loaded\n’, fname);
    load(fname); % truncated trace
catch
    fprintf(’No file found: %s\n’, fname);
    return;
end

slotted_data_1hrs = slot_data(:, :, cliping_start_slot:
    cliping_start_slot ...
    + tot_duration - 1);
train_model(slotted_data_1hrs, env{1}, type{1}, nodetype);
end
end

%% Function definition
function [gmmhmm] = train_model(train_data, env, traffic_type, nodetype)

%% Model parameters
rng(227, ’twister’);
Q = 2; ’nstates’
M = 7; ’nmix’
O = 2; ’ndim’
T = 1; ’seq_len’
cov_type = ’diag’;
n_iter = 10;
thresh = 1e-6;
pkt_count_threshold = 5;
slot_len_sec = 0.05;
slot_len_ms = slot_len_sec*1000.0;
iat_threshold = 8.512;

%% Start training
[train_features_busy, train_features_free] =
    cluster_data(train_data, iat_threshold, pkt_count_threshold,
                 slot_len_ms, 'train');

n_busy_train = length(train_features_busy);
n_free_train = length(train_features_free);
n_train_samples = n_busy_train + n_free_train;
observed_train = [train_features_busy; train_features_free];
hidden_train = [ones(n_busy_train, 1); ones(n_free_train, 1)*2];
[observed_train, hidden_train] = shuffleRows(observed_train,
                                            hidden_train);
observed_train = reshape(observed_train, 1, 1, n_train_samples);
train_data = cell2mat(observed_train);
train_labels = hidden_train;

%Find all model parameters with fully observed data
transmat1 = transmat_train_observed(train_labels, Q);

% transform labels: 1-->0 (Busy), 2-->1 (Free)
labels = zeros(size(train_labels));
labels(train_labels==1) = 0;
labels(train_labels==2) = 1;
try
    % what if all training set says channel is BUSY or FREE? Then,
    % kmeans algorithm fails to cluster.
    mixgauss = mixgauss_classifier_train(train_data, labels, M, ...
                                         'cov_type', cov_type, 'max_iter', n_iter, 'thresh', thresh,
                                         'method', 'kmeans');
catch
    fprintf('no model trained!\n');
    return;
end
mu1(:, 1, :) = mixgauss.neg.mu;
mu1(:, 2, :) = mixgauss.pos.mu;
Sigma1(:, :, 1, :) = mixgauss.neg.Sigma;
Sigma1(:, :, 2, :) = mixgauss.pos.Sigma;
prior1 = flip(mixgauss.priorC)';
mixmat1(1, :) = mixgauss.neg.prior';
mixmat1(2, :) = mixgauss.pos.prior';
B.2 MATLAB

100  gmmhmm = struct( ... 
101      'mu', mu1, ...  % means
102      'Sigma', Sigma1, ...  % covariance
103      'mixmat', mixmat1, ...  % mixture weights
104      'prior', prior1, ...  % state prob.
105      'transmat', transmat1 ...  % transition prob.
106    );
107
108  if exist('1hrs_models.7mix', 'dir')~=0
109      mkdir('1hrs_models.7mix');
110  end
111  model_name = ['1hrs_models.7mix/1hrs_model_', 
112                 num2str(slot_len_ms),'ms_','env','_','traffic_type','_nodetype', 
113                 num2str(nodetype),'.mat'];
114  save(model_name, 'gmmhmm');
115

B.2.4 Pre-compute White Spaces

The following code was used to store predicted white spaces.

% instrfind
% cd cooja-matlab-connection

% Parameters
env = 'home';
wsn_ch = 18;

% Slot
slot_len_sec = 0.05;

% Seed
seed = 227;
rng(seed, 'twister'); %the initial status of RNG
rng_init = rng;

% Number of predictions per request
max_slot_predictions = 2;

% Constants
n = 3600/slot_len_sec;
init_slot = 71; %31;

% Predictions start time
started_at = datestr(datetime, 'yyyymmd_HHmmss');
display(started_at);

for wsn_app_period_sec = [5, 10, 30, 60]
    T = wsn_app_period_sec/slot_len_sec;

    for nodetype = 1:3
        % all nodes starts with the same random state
        rng(rng_init);

        for day_id = 1:2
            for hour_id = 1:24
                for traffic_type = {'peak', 'offpeak'}
                    if strcmp(traffic_type, 'peak')
                        type_id = 1;
                    else
                        type_id = 2;
                    end

                % Corresponding model information
                model_info = struct(...
                    'env', env, ...
                    'wsn_ch', wsn_ch, ...
                    'wsn_app_period_sec', wsn_app_period_sec, ...
                    'day_id', day_id, ...
                    'hour_id', hour_id, ...
                    'traffic_type', traffic_type, ...
                    'node_id', nodetype, ...
                    'slot_len_sec', slot_len_sec, ...
                    'max_slot_predictions', max_slot_predictions ...
                        %#continuous predictions per req.
                );
            end
        end
    end
end
app_period_index = dataperiod2index(wsn_app_period_sec);
if app_period_index == -1
    fprintf('check wsn_app_period_sec\n');
    return;
end

%load model
gmmhmm = load_gmmhmm_model(model_info);

%load slotted trace
slot_data = load_trace(model_info);
if isempty(slot_data)
    continue;
end

fprintf('predicting for node-%d day-%d hour-%d %s traffic\n', ...
    [nodetype, day_id, hour_id, traffic_type{1}]);

% Predictions
for slot_id = init_slot:T:n 1:T:n
    slot_index = slot2index(slot_id, init_slot, T);
    rfsi = get_rfsi(slot_id, gmmhmm, model_info,
        slot_data);
    predictions_db(nodetype).period(app_period_index...
        ).type(type_id).day(day_id).hour(hour_id...
        ).slot(slot_index).rfsi = rfsi;
end

try
    % Save the db every hour
    save(['predictions_db/predictions_db_', env, '_ch-',
        num2str(wsn_ch), '.mat'], 'predictions_db');
    fprintf('\tDB saved at %s\n', datestr(datetime,
        'yyyymmdd_HHMMSS'));
catch
    %
save([predictions_db/predictions_db_, env, '_ch-', num2str(wsn_ch), '.mat'], 'predictions_db');

fprintf('	DB saved at %s!\n', datestr(datetime, 'yyyyymmdd_HHMMSS'));

catch

end

B.2.5 Obtain Relative Free Slot Index

The following function is used to obtain predictions from the white space model. The model predict a sequence of BUSY and FREE slots and the slots are marked relative to the current data period. This function returns the relative positions of the predicted FREE slots.

function [rfsi] = get_rfsi(current_slot_id, gmmhmm, model_info, slot_data)

%% Here we go through the slot data
[l, m, n] = size(slot_data);
current_slot_id = mod(current_slot_id, n);
if current_slot_id == 0
    current_slot_id = 1;
end

%% Model info
wsn_app_period_sec = model_info.wsn_app_period_sec;
slot_len_sec = model_info.slot_len_sec;
MAX_SLOT_PREDICTIONS = model_info.max_slot_predictions;
slot_len_ms = slot_len_sec*1e3;
n_past_samples = wsn_app_period_sec/(slot_len_ms*1e-3);
n_future_samples = n_past_samples;

BUSY = 0;
FREE = 1;

%% Model parameters
mu1 = gmmhmm.mu;
Sigma1 = gmmhmm.Sigma;
mixmat1 = gmmhmm.mixmat;
prior1 = gmmhmm.prior;
transmat1 = gmmhmm.transmat;

%% Past samples
if current_slot_id > n_past_samples
    past_data = slot_data(:, :, current_slot_id-1: -1: current_slot_id - n_past_samples);
elseif current_slot_id > 1 && current_slot_id <= n_past_samples
    past_data = slot_data(:, :, 1:current_slot_id-1);
else
    past_data = [];
end

%% Current sample
current_sample = slot_data(:, :, current_slot_id);
obslik = mixgauss_prob(current_sample, mu1, Sigma1, mixmat1);
current_state = viterbi_path(prior1, transmat1, obslik)';

%% Future samples
future_data = mhmm_sample_adis(1, n_future_samples, prior1, ... 
    transmat1, mu1, Sigma1, mixmat1, current_state);
data_set = cat(3, past_data, current_sample, future_data);

%% Find FREE slot
if size(past_data, 3) == 1
    tot_n_samples = n_future_samples + 1;
else
    tot_n_samples = size(past_data, 3) + n_future_samples + 1;
end
obslik = zeros(2, tot_n_samples);
for i = 1: tot_n_samples
    \% calculate observation likelihood of each sample
    obslik(:, i) = mixgauss_prob(data_set(:, i), mu1, Sigma1, mixmat1);
end
\% obtain the corresponding state sequence
\% 0: Busy
\% 1: Free
state_seq = viterbi_path(prior1, transmat1, obslik)';
state_seq(state Seq==1)=0;
state_seq(state Seq==2)=1;
future_state_seq = state_seq(size(past_data, 3)+2: end);
\% Find the most likely, the closest MAX_SLOT_PREDICTIONS white spaces
relative_free_slot_pos = -1*ones(1, MAX_SLOT_PREDICTIONS);
free_slot_cnt = 0;
for i = 1:length(future_state_seq)
    if future_state_seq(i) == FREE
        if free_slot_cnt >= MAX_SLOT_PREDICTIONS
            break;
        end
        free_slot_cnt = free_slot_cnt + 1;
        relative_free_slot_pos(1, free_slot_cnt) = i;
    end
end
rfsi = relative_free_slot_pos;
end
Appendix C

JamLab Settings

JamLab was used to inject noise into the simulation environment using the following source code. In this work, JamLab was used to generate RF noise according to a Probability Density Function (PDF) of the Time Between Noise Signals (TBNS). This PDF is extracted from the collected real-world traces depending on the environment (HOME/OFFICE) and traffic type (Peak/Off-peak). Because two jammers were used in the simulations, there were eight different PDFs defined in the JamLab configurations with the variable P[] in the following source code. Statistically, the two jammers generate noise according to the TBNS distribution of the collected real-world traces. To make the noise generation more realistic, two jamming durations (i.e., 0.413 ms and 7.278 ms) were borrowed from the original JamLab implementation that emulates short and long jamming signals with 0 dBm jamming power.

```c
/*
 * Copyright (c) 2014, TU Graz, Austria.
 * Copyright (c) 2010, Swedish Institute of Computer Science (SICS), Sweden.
 * All rights reserved.
 * Redistribution and use in source and binary forms, with or without modification, are not permitted.
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS 'AS IS' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* Author: Carlo Alberto Boano <cboano@tugraz.at>
* Thiemo Voigt <thiemo@sics.se>
*
* Modified: Indika S. A. Dhanapala
*
* Description: Settings for JamLab’s interference emulation.
*
*/

#include "settings_jamlab.h"
#include "clock.h"

// Model-parameters
#define NR 101
#define NRANDS 1100

// Debugging active?
#define JAMLAB_DEBUG_ON 0

// Global variables
static uint16_t tmp_cnt_1 = 0;
static uint16_t tmp_cnt_2 = 0;
static float r;
static int off_slots;
static float sump = 0;
static float P[NR] = {0.000f};
static float cdf[NR];
static float rands[NRANDS];

// Used externally
#define NUM_COUNTS 5000
static uint16_t stop_counter = 0;

/**-----------------------------------------------*/

/////////////-------------------------------------
// Radio settings for jamming //
/////////////-------------------------------------

/**-----------------------------------------------*/

// Reset the interferer back to normal mode (setting back the registers
to their original value)
void reset_jammer(uint8_t carrier){
  if(carrier){
    SPI_SET_MODULATED(0x0500);
  }
  else{
    SPI_SET_UNMODULATED(0x0000,0x0000,0x0500,0x0010);
  }
  ENERGEST_OFF(ENERGEST_TYPE_TRANSMIT);
  ENERGEST_ON(ENERGEST_TYPE_LISTEN);
}

/**-----------------------------------------------*/

// Starting the interferer (0 = unmodulated carrier, !0 = modulated
carrier)
void set_jammer(uint8_t carrier){
  if(carrier){
    // The CC2420 has a built-in test pattern generator that can
generate pseudo random sequence using the CRC generator.
// This is enabled by setting MDMCTRL1.TX_MODE to 3 and issue a
STXON command strobe. The modulated spectrum is then available
on the RF pins.

// The low byte of the CRC word is transmitted and the CRC is
updated with 0xFF for each new byte.

// The length of the transmitted data sequence is 65535 bits. The
transmitted data-sequence is then: [synch header]
SPI_SET_MODULATED(0x050C);
}
else{

// An unmodulated carrier may be transmitted by setting
MDMCTRL1.TX_MODE to 2, writing 0x1800 to the DACTST register
and issue a STXON command strobe.

// The transmitter is then enabled while the transmitter I/Q DACs
are overridden to static values.

// An unmodulated carrier will then be available on the RF output
pins.
SPI_SET_UNMODULATED(0x1800,0x0100,0x0508,0x0004);
}
ENERGEST_OFF(ENERGEST_TYPE_LISTEN);
ENERGEST_ON(ENERGEST_TYPE_TRANSMIT);
}

/***************************************************************************/
// Setting the transmission power to pow
void power_jammer(uint8_t pow){

// 0xa0ff is the initial value of the CC2420_TXCTRL register
// measured by me
SPI_SET_TXPOWER((0xa0ff & 0xffe0) | (pow & 0x1f));
}

/***************************************************************************/
void assign_model_parameters(uint8_t emulated_interference_type,
uint8_t loc) {

if(loc == 1){

// Peak traffic in the office
if(emulated_interference_type == JL_OFFICE_PEAK){
P[0] = 0.2961;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.3290;
P[10] = 0.0130;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.0724;
P[20] = 0.0068;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0534;
P[30] = 0.0073;
P[31] = 0.0000;
P[32] = 0.0000;
P[33] = 0.0000;
P[34] = 0.0000;
P[35] = 0.0000;
P[36] = 0.0000;
P[37] = 0.0000;
P[38] = 0.0000;
P[39] = 0.0363;
P[40] = 0.0070;
P[41] = 0.0000;
P[42] = 0.0000;
P[43] = 0.0000;
P[44] = 0.0000;
P[45] = 0.0000;
P[46] = 0.0000;
P[47] = 0.0000;
P[48] = 0.0000;
P[49] = 0.0101;
P[50] = 0.0025;
P[51] = 0.0000;
P[52] = 0.0000;
P[53] = 0.0000;
P[54] = 0.0000;
P[55] = 0.0000;
P[56] = 0.0000;
P[57] = 0.0000;
P[58] = 0.0000;
P[59] = 0.0267;
P[60] = 0.0084;
P[61] = 0.0000;
P[62] = 0.0000;
P[63] = 0.0000;
P[64] = 0.0000;
P[65] = 0.0000;
P[66] = 0.0000;
P[67] = 0.0000;
P[68] = 0.0000;
P[69] = 0.0311;
P[70] = 0.0121;
P[71] = 0.0000;
P[72] = 0.0000;
P[73] = 0.0000;
P[74] = 0.0000;
P[75] = 0.0000;
P[76] = 0.0000;
P[77] = 0.0000;
P[78] = 0.0000;
P[79] = 0.0360;
P[80] = 0.0173;
P[81] = 0.0000;
P[82] = 0.0000;
P[83] = 0.0000;
P[84] = 0.0000;
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0146;
P[90] = 0.0082;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0116;
}
// Off-peak traffic in the office
else if(emulated_interference_type == JL_OFFICE_OFFPEAK){
P[0] = 0.2651;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.2627;
P[10] = 0.0107;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.0927;
P[20] = 0.0081;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0656;
P[30] = 0.0092;
P[31] = 0.0000;
P[32] = 0.0000;
P[33] = 0.0000;
P[34] = 0.0000;
P[35] = 0.0000;
P[36] = 0.0000;
P[37] = 0.0000;
P[38] = 0.0000;
P[39] = 0.0431;
P[40] = 0.0081;
P[41] = 0.0000;
P[42] = 0.0000;
P[43] = 0.0000;
P[44] = 0.0000;
P[45] = 0.0000;
P[46] = 0.0000;
P[47] = 0.0000;
P[48] = 0.0000;
P[49] = 0.0731;
P[50] = 0.0184;
P[51] = 0.0000;
P[52] = 0.0000;
P[53] = 0.0000;
P[54] = 0.0000;
P[55] = 0.0000;
P[56] = 0.0000;
P[57] = 0.0000;
P[58] = 0.0000;
P[59] = 0.0439;
P[60] = 0.0137;
P[61] = 0.0000;
P[62] = 0.0000;
P[63] = 0.0000;
P[64] = 0.0000;
P[65] = 0.0000;
P[66] = 0.0000;
P[67] = 0.0000;
P[68] = 0.0000;
P[69] = 0.0262;
P[70] = 0.0102;
P[71] = 0.0000;
P[72] = 0.0000;
P[73] = 0.0000;
P[74] = 0.0000;
P[75] = 0.0000;
P[76] = 0.0000;
P[77] = 0.0000;
P[78] = 0.0000;
P[79] = 0.0127;
P[80] = 0.0061;
P[81] = 0.0000;
P[82] = 0.0000;
P[83] = 0.0000;
P[84] = 0.0000;
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0096;
P[90] = 0.0051;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0156;
}
// Peak traffic in the home
else if(emulated_interference_type == JL_HOME_PEAK){
    P[0] = 0.5228;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.2107;
P[10] = 0.0088;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.0643;
P[20] = 0.0058;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0428;
P[30] = 0.0057;

P[31] = 0.0000;

P[32] = 0.0000;

P[33] = 0.0000;

P[34] = 0.0000;

P[35] = 0.0000;

P[36] = 0.0000;

P[37] = 0.0000;

P[38] = 0.0000;

P[39] = 0.0454;

P[40] = 0.0087;

P[41] = 0.0000;

P[42] = 0.0000;

P[43] = 0.0000;

P[44] = 0.0000;

P[45] = 0.0000;

P[46] = 0.0000;

P[47] = 0.0000;

P[48] = 0.0000;

P[49] = 0.0370;

P[50] = 0.0091;

P[51] = 0.0000;

P[52] = 0.0000;

P[53] = 0.0000;

P[54] = 0.0000;

P[55] = 0.0000;

P[56] = 0.0000;

P[57] = 0.0000;

P[58] = 0.0000;

P[59] = 0.0231;

P[60] = 0.0071;

P[61] = 0.0000;

P[62] = 0.0000;

P[63] = 0.0000;

P[64] = 0.0000;

P[65] = 0.0000;

P[66] = 0.0000;

P[67] = 0.0000;

P[68] = 0.0000;
P[69] = 0.0055;
P[70] = 0.0023;
P[71] = 0.0000;
P[72] = 0.0000;
P[73] = 0.0000;
P[74] = 0.0000;
P[75] = 0.0000;
P[76] = 0.0000;
P[77] = 0.0000;
P[78] = 0.0000;
P[79] = 0.0005;
P[80] = 0.0002;
P[81] = 0.0000;
P[82] = 0.0000;
P[83] = 0.0000;
P[84] = 0.0000;
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0000;
P[90] = 0.0000;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0001;
}

// Off-peak traffic in the home
else if(emulated_interference_type == JL_HOME_OFFPEAK){
    P[0] = 0.4464;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.1669;
P[10] = 0.0005;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.0381;
P[20] = 0.0012;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0246;
P[30] = 0.0035;
P[31] = 0.0000;
P[32] = 0.0000;
P[33] = 0.0000;
P[34] = 0.0000;
P[35] = 0.0000;
P[36] = 0.0000;
P[37] = 0.0000;
P[38] = 0.0000;
P[39] = 0.0218;
P[40] = 0.0036;
P[41] = 0.0000;
P[42] = 0.0000;
P[43] = 0.0000;
<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>0.0000</td>
</tr>
<tr>
<td>45</td>
<td>0.0000</td>
</tr>
<tr>
<td>46</td>
<td>0.0000</td>
</tr>
<tr>
<td>47</td>
<td>0.0000</td>
</tr>
<tr>
<td>48</td>
<td>0.0000</td>
</tr>
<tr>
<td>49</td>
<td>0.0161</td>
</tr>
<tr>
<td>50</td>
<td>0.0082</td>
</tr>
<tr>
<td>51</td>
<td>0.0000</td>
</tr>
<tr>
<td>52</td>
<td>0.0000</td>
</tr>
<tr>
<td>53</td>
<td>0.0000</td>
</tr>
<tr>
<td>54</td>
<td>0.0000</td>
</tr>
<tr>
<td>55</td>
<td>0.0000</td>
</tr>
<tr>
<td>56</td>
<td>0.0000</td>
</tr>
<tr>
<td>57</td>
<td>0.0000</td>
</tr>
<tr>
<td>58</td>
<td>0.0000</td>
</tr>
<tr>
<td>59</td>
<td>0.0123</td>
</tr>
<tr>
<td>60</td>
<td>0.0116</td>
</tr>
<tr>
<td>61</td>
<td>0.0000</td>
</tr>
<tr>
<td>62</td>
<td>0.0000</td>
</tr>
<tr>
<td>63</td>
<td>0.0000</td>
</tr>
<tr>
<td>64</td>
<td>0.0000</td>
</tr>
<tr>
<td>65</td>
<td>0.0000</td>
</tr>
<tr>
<td>66</td>
<td>0.0000</td>
</tr>
<tr>
<td>67</td>
<td>0.0000</td>
</tr>
<tr>
<td>68</td>
<td>0.0000</td>
</tr>
<tr>
<td>69</td>
<td>0.0111</td>
</tr>
<tr>
<td>70</td>
<td>0.0104</td>
</tr>
<tr>
<td>71</td>
<td>0.0000</td>
</tr>
<tr>
<td>72</td>
<td>0.0000</td>
</tr>
<tr>
<td>73</td>
<td>0.0000</td>
</tr>
<tr>
<td>74</td>
<td>0.0000</td>
</tr>
<tr>
<td>75</td>
<td>0.0000</td>
</tr>
<tr>
<td>76</td>
<td>0.0000</td>
</tr>
<tr>
<td>77</td>
<td>0.0000</td>
</tr>
<tr>
<td>78</td>
<td>0.0000</td>
</tr>
<tr>
<td>79</td>
<td>0.0000</td>
</tr>
<tr>
<td>80</td>
<td>0.0000</td>
</tr>
<tr>
<td>81</td>
<td>0.0000</td>
</tr>
<tr>
<td>82</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
\[\text{else if}(\text{loc} == 2)\{
\]

\[\text{// Peak traffic in the office}
\]

\[\text{if}(\text{emulated interference type} == \text{JL OFFICE PEAK})\{
\]

\[\text{P}[0] = 0.4129;
\]

\[\text{P}[1] = 0.0000;
\]

\[\text{P}[2] = 0.0000;
\]

\[\text{P}[3] = 0.0000;
\]

\[\text{P}[4] = 0.0000;
\]

\[\text{P}[5] = 0.0000;
\]

\[\text{P}[6] = 0.0000;
\]

\[\text{P}[7] = 0.0000;
\]

\[\text{P}[8] = 0.0000;
\]

\[\text{P}[9] = 0.3265;
\]

\[\text{P}[10] = 0.0134;
\]

\[\text{P}[11] = 0.0000;
\]

\[\text{P}[12] = 0.0000;
\]

\[\text{P}[13] = 0.0000;
\]

\[\text{P}[14] = 0.0000;
\]

\[\text{P}[15] = 0.0000;
\]

\[\text{P}[16] = 0.0000;
\]

\[\text{P}[17] = 0.0000;
\]
P[18] = 0.0000;
P[19] = 0.1379;
P[20] = 0.0122;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0400;
P[30] = 0.0054;
P[31] = 0.0000;
P[32] = 0.0000;
P[33] = 0.0000;
P[34] = 0.0000;
P[35] = 0.0000;
P[36] = 0.0000;
P[37] = 0.0000;
P[38] = 0.0000;
P[39] = 0.0216;
P[40] = 0.0039;
P[41] = 0.0000;
P[42] = 0.0000;
P[43] = 0.0000;
P[44] = 0.0000;
P[45] = 0.0000;
P[46] = 0.0000;
P[47] = 0.0000;
P[48] = 0.0000;
P[49] = 0.0119;
P[50] = 0.0030;
P[51] = 0.0000;
P[52] = 0.0000;
P[53] = 0.0000;
P[54] = 0.0000;
P[55] = 0.0000;
P[56] = 0.0000;
P[57] = 0.0000;
P[58] = 0.0000;
P[59] = 0.0060;
P[60] = 0.0021;
P[61] = 0.0000;
P[62] = 0.0000;
P[63] = 0.0000;
P[64] = 0.0000;
P[65] = 0.0000;
P[66] = 0.0000;
P[67] = 0.0000;
P[68] = 0.0000;
P[69] = 0.0020;
P[70] = 0.0009;
P[71] = 0.0000;
P[72] = 0.0000;
P[73] = 0.0000;
P[74] = 0.0000;
P[75] = 0.0000;
P[76] = 0.0000;
P[77] = 0.0000;
P[78] = 0.0000;
P[79] = 0.0001;
P[80] = 0.0001;
P[81] = 0.0000;
P[82] = 0.0000;
P[83] = 0.0000;
P[84] = 0.0000;
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0000;
P[90] = 0.0000;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0000;
}
// Off-peak traffic in the office
else if(emulated_interference_type == JL_OFFICE_OFFPEAK){
  P[0] = 0.3642;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.3558;
P[10] = 0.0144;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.1091;
P[20] = 0.0099;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0319;
P[30] = 0.0045;
P[31] = 0.0000;
\[ P[32] = 0.0000; \]
\[ P[33] = 0.0000; \]
\[ P[34] = 0.0000; \]
\[ P[35] = 0.0000; \]
\[ P[36] = 0.0000; \]
\[ P[37] = 0.0000; \]
\[ P[38] = 0.0000; \]
\[ P[39] = 0.0192; \]
\[ P[40] = 0.0035; \]
\[ P[41] = 0.0000; \]
\[ P[42] = 0.0000; \]
\[ P[43] = 0.0000; \]
\[ P[44] = 0.0000; \]
\[ P[45] = 0.0000; \]
\[ P[46] = 0.0000; \]
\[ P[47] = 0.0000; \]
\[ P[48] = 0.0000; \]
\[ P[49] = 0.0256; \]
\[ P[50] = 0.0064; \]
\[ P[51] = 0.0000; \]
\[ P[52] = 0.0000; \]
\[ P[53] = 0.0000; \]
\[ P[54] = 0.0000; \]
\[ P[55] = 0.0000; \]
\[ P[56] = 0.0000; \]
\[ P[57] = 0.0000; \]
\[ P[58] = 0.0000; \]
\[ P[59] = 0.0231; \]
\[ P[60] = 0.0075; \]
\[ P[61] = 0.0000; \]
\[ P[62] = 0.0000; \]
\[ P[63] = 0.0000; \]
\[ P[64] = 0.0000; \]
\[ P[65] = 0.0000; \]
\[ P[66] = 0.0000; \]
\[ P[67] = 0.0000; \]
\[ P[68] = 0.0000; \]
\[ P[69] = 0.0164; \]
\[ P[70] = 0.0063; \]
P[71] = 0.0000;
P[72] = 0.0000;
P[73] = 0.0000;
P[74] = 0.0000;
P[75] = 0.0000;
P[76] = 0.0000;
P[77] = 0.0000;
P[78] = 0.0000;
P[79] = 0.0015;
P[80] = 0.0006;
P[81] = 0.0000;
P[82] = 0.0000;
P[83] = 0.0000;
P[84] = 0.0000;
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0000;
P[90] = 0.0000;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0000;

} // Peak traffic in the home
else if(emulated_interference_type == JL_HOME_PEAK){
    P[0] = 0.6425;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
\begin{verbatim}
  P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.1682;
P[10] = 0.0089;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.0489;
P[20] = 0.0048;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0295;
P[30] = 0.0043;
P[31] = 0.0000;
P[32] = 0.0000;
P[33] = 0.0000;
P[34] = 0.0000;
P[35] = 0.0000;
P[36] = 0.0000;
P[37] = 0.0000;
P[38] = 0.0000;
P[39] = 0.0237;
P[40] = 0.0045;
P[41] = 0.0000;
P[42] = 0.0000;
P[43] = 0.0000;
P[44] = 0.0000;
P[45] = 0.0000;
\end{verbatim}
\[ \begin{align*}
\text{P}[46] &= 0.0000; \\
\text{P}[47] &= 0.0000; \\
\text{P}[48] &= 0.0000; \\
\text{P}[49] &= 0.0158; \\
\text{P}[50] &= 0.0036; \\
\text{P}[51] &= 0.0000; \\
\text{P}[52] &= 0.0000; \\
\text{P}[53] &= 0.0000; \\
\text{P}[54] &= 0.0000; \\
\text{P}[55] &= 0.0000; \\
\text{P}[56] &= 0.0000; \\
\text{P}[57] &= 0.0000; \\
\text{P}[58] &= 0.0000; \\
\text{P}[59] &= 0.0114; \\
\text{P}[60] &= 0.0036; \\
\text{P}[61] &= 0.0000; \\
\text{P}[62] &= 0.0000; \\
\text{P}[63] &= 0.0000; \\
\text{P}[64] &= 0.0000; \\
\text{P}[65] &= 0.0000; \\
\text{P}[66] &= 0.0000; \\
\text{P}[67] &= 0.0000; \\
\text{P}[68] &= 0.0000; \\
\text{P}[69] &= 0.0062; \\
\text{P}[70] &= 0.0022; \\
\text{P}[71] &= 0.0000; \\
\text{P}[72] &= 0.0000; \\
\text{P}[73] &= 0.0000; \\
\text{P}[74] &= 0.0000; \\
\text{P}[75] &= 0.0000; \\
\text{P}[76] &= 0.0000; \\
\text{P}[77] &= 0.0000; \\
\text{P}[78] &= 0.0000; \\
\text{P}[79] &= 0.0052; \\
\text{P}[80] &= 0.0022; \\
\text{P}[81] &= 0.0000; \\
\text{P}[82] &= 0.0000; \\
\text{P}[83] &= 0.0000; \\
\text{P}[84] &= 0.0000; \\
\end{align*} \]
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0049;
P[90] = 0.0021;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0076;
}

// Off-peak traffic in the home
else if(emulated_interference_type == JL_HOME_OFFPEAK){
    P[0] = 0.6509;
P[1] = 0.0000;
P[2] = 0.0000;
P[3] = 0.0000;
P[4] = 0.0000;
P[5] = 0.0000;
P[6] = 0.0000;
P[7] = 0.0000;
P[8] = 0.0000;
P[9] = 0.1798;
P[10] = 0.0054;
P[11] = 0.0000;
P[12] = 0.0000;
P[13] = 0.0000;
P[14] = 0.0000;
P[15] = 0.0000;
P[16] = 0.0000;
P[17] = 0.0000;
P[18] = 0.0000;
P[19] = 0.0320;
P[20] = 0.0019;
P[21] = 0.0000;
P[22] = 0.0000;
P[23] = 0.0000;
P[24] = 0.0000;
P[25] = 0.0000;
P[26] = 0.0000;
P[27] = 0.0000;
P[28] = 0.0000;
P[29] = 0.0195;
P[30] = 0.0017;
P[31] = 0.0000;
P[32] = 0.0000;
P[33] = 0.0000;
P[34] = 0.0000;
P[35] = 0.0000;
P[36] = 0.0000;
P[37] = 0.0000;
P[38] = 0.0000;
P[39] = 0.0141;
P[40] = 0.0018;
P[41] = 0.0000;
P[42] = 0.0000;
P[43] = 0.0000;
P[44] = 0.0000;
P[45] = 0.0000;
P[46] = 0.0000;
P[47] = 0.0000;
P[48] = 0.0000;
P[49] = 0.0110;
P[50] = 0.0024;
P[51] = 0.0000;
P[52] = 0.0000;
P[53] = 0.0000;
P[54] = 0.0000;
P[55] = 0.0000;
P[56] = 0.0000;
P[57] = 0.0000;
P[58] = 0.0000;
P[59] = 0.0087;
P[60] = 0.0026;
P[61] = 0.0000;
P[62] = 0.0000;
P[63] = 0.0000;
P[64] = 0.0000;
P[65] = 0.0000;
P[66] = 0.0000;
P[67] = 0.0000;
P[68] = 0.0000;
P[69] = 0.0062;
P[70] = 0.0031;
P[71] = 0.0000;
P[72] = 0.0000;
P[73] = 0.0000;
P[74] = 0.0000;
P[75] = 0.0000;
P[76] = 0.0000;
P[77] = 0.0000;
P[78] = 0.0000;
P[79] = 0.0059;
P[80] = 0.0038;
P[81] = 0.0000;
P[82] = 0.0000;
P[83] = 0.0000;
P[84] = 0.0000;
P[85] = 0.0000;
P[86] = 0.0000;
P[87] = 0.0000;
P[88] = 0.0000;
P[89] = 0.0066;
P[90] = 0.0047;
P[91] = 0.0000;
P[92] = 0.0000;
P[93] = 0.0000;
P[94] = 0.0000;
P[95] = 0.0000;
P[96] = 0.0000;
P[97] = 0.0000;
P[98] = 0.0000;
P[99] = 0.0376;
}
}

/************************************************************************
// Simple periodic pattern (like the one from microwave ovens)

void periodic_jammer(uint16_t period_on, uint16_t period_off, uint8_t
txpower, uint8_t carrier) {
    // Enable carrier
    CC2420_SPI_ENABLE();
    set_jammer(carrier);
    for(stop_counter=0; stop_counter<NUM_COUNTS; stop_counter++) {
        // ON cycle
        #if JAMLAB_RANDOM_POWER
            uint8_t random_power = (random_rand() % txpower);
            power_jammer(random_power);
        #else
            power_jammer(txpower);
        #endif
        clock_delay(period_on);
        // OFF cycle
        power_jammer(JAMLAB_LOWEST_POWER);
        clock_delay(period_off);
    } 
    CC2420_SPI_DISABLE();
}

/************************************************************************
void jamlab_emulation(uint8_t radio_channel, uint8_t
    interference_power, uint8_t interference_type, uint8_t
carrier_type, uint8_t loc) {
    random_init(227);
    // Setting transmission power and radio channel
    cc2420_set_channel(radio_channel);
    cc2420_set_txpower(interference_power);
if((carrier_type != JAMLAB_CARRIER_TYPE_MODULATED) && (carrier_type 

  != JAMLAB_CARRIER_TYPE_UNMODULATED)){
  printf("Invalid carrier type %u. JamLab cannot start!\n", 
        carrier_type);
  return;
}

#ifdef JAMLAB_DEBUG_ON
  printf("Starting JamLab, emulation mode for interference type %u\n", 
        interference_type);
#endif

// Stop the watchdog
watchdog_stop();

if(interference_type <= 7) {

  // No interference
  if(interference_type == JL_NOINT){
    // Disable carrier
    CC2420_SPI_ENABLE();
    power_jammer(JAMLAB_LOWEST_POWER);
    reset_jammer(carrier_type);
    CC2420_SPI_DISABLE();
  }

  // Continuous carrier
  else if(interference_type == JL_CONTN){
    // Enabling carrier
    CC2420_SPI_ENABLE();
    power_jammer(interference_power);
    set_jammer(carrier_type);
    CC2420_SPI_DISABLE();
  }

  // CDF-based interference
  else{
assign_model_parameters(interference_type, loc);

for(tmp_cnt_1=0;tmp_cnt_1<NRANDS;tmp_cnt_1++) {
    rands[tmp_cnt_1] = (float)(random_rand()%32768)/(32768);
    if (rands[tmp_cnt_1]<0 || rands[tmp_cnt_1]> 1.0) {
        printf("== ERROR ==\n");
    }
    if (tmp_cnt_1<20) {
        #if JAMLAB_DEBUG_ON
            printf("rands[%d] %dn", tmp_cnt_1, (int)(rands[tmp_cnt_1]*100));
        #endif
    }
}

for(tmp_cnt_1=0;tmp_cnt_1<NR;tmp_cnt_1++) {
    sump += P[tmp_cnt_1];
    cdf[tmp_cnt_1] = sum;
    #if JAMLAB_DEBUG_ON
        printf("cdf(%d) \n", (int)(100*cdf[tmp_cnt_1]));
    #endif
}

// Starting the carrier
CC2420_SPI_ENABLE();

tmp_cnt_1 = 0;
//for(stop_counter=0;stop_counter<NUM_COUNTS;stop_counter++){
while(1){
    // Beginning of interference

    // Turning on interferer
    #if JAMLAB_RANDOM_POWER
        uint8_t random_power = (random_rand() % interference_power);
        power_jammer(random_power);
    #else
        power_jammer(interference_power);
    #endif
```c
set_jammer(carrier_type);

r = rands[tmp_cnt_1];
tmp_cnt_1++;

// We oscillate between small and larger packet sizes
if (r<0.5) { //if (r<0.5) {
    // very short jammer on
clock_delay(2571); //(146); //(2088); // 2332 - 244 //adis
} else {
clock_delay(146); //(146); //(2571); // 2815 - 244 //adis
}

for(tmp_cnt_2=0;tmp_cnt_2<NR;tmp_cnt_2++) {
    if (r < cdf[tmp_cnt_2]) {
        off_slots = tmp_cnt_2;
        break;
    }
}

// Turning off interferer
power_jammer(JAMLAB_LOWEST_POWER);
set_jammer(carrier_type);

// one off_slot is 27 us, gives (27-4)/0.78=29.4 clock_delays
//clock_delay(off_slots*1282); // 1 MILLISECOND
clock_delay((off_slots*(720 - 16))); //(off_slots*(720 - 64));
    //((720-64)); //(720); //(656)

if (tmp_cnt_1 == (NRANDS-1)){
tmp_cnt_1 = 0;
printf("#round\n");
}

// Disable the carrier
CC2420_SPI_DISABLE();
```
else{
    printf("ERROR: unknown interference type!\n");
}
return;
Appendix D

Packet Formats

For disseminating control commands, i.e., periodic time synchronisation and model re-parametrisation, throughout the network, the packet format depicted in Figure D.1 is used. MHR and MFR fields of the packet are dedicated for the MAC header and the footer. Command type distinguishes the control packet. The sequence number field specifies the sequence identifier of the control packet. The base slot indicates the ID of the first slot of the target data period. Re-parametrisation and parameter type define whether the parameter set should be swapped and the corresponding parameter set, respectively. Age, authoritative level and timestamp fields are used for time synchronisation.

<table>
<thead>
<tr>
<th>Octets:</th>
<th>17</th>
<th>1</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHR</td>
<td>Cmd. Type</td>
<td>Sequence Number</td>
<td>Base Slot</td>
<td>Re-param.</td>
<td>Param. Type</td>
<td>Age</td>
<td>Auth. Level</td>
<td>Timestamp</td>
<td>MFR</td>
</tr>
</tbody>
</table>

Fig. D.1 MAC control packet format.

For exchanging the white space model with neighbours, the packet format shown in Figure D.2 is used. The command type field defines the type of the control packet. The model data field carries parameters of the white space model. The remaining fields are as same as that of the control packet illustrated in Figure D.1.

<table>
<thead>
<tr>
<th>Octets:</th>
<th>17</th>
<th>1</th>
<th>variable</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHR</td>
<td>Cmd. Type</td>
<td>Model Data</td>
<td>Sequence Number</td>
<td>Timestamp</td>
<td>MFR</td>
<td></td>
</tr>
</tbody>
</table>

Fig. D.2 White space model exchange packet format.
Appendix E

Simulation Log

The following is a snippet from the simulation log of node 2 when it was executing the 10 seconds periodic data collection application. Each line starts with the timestamp in microseconds since the node had booted. Text that starts with APP marks the beginning of a data period, while r_on and r_off denote radio on and off status. The node requests a list of free receive/transmit slots by sending its ID, the flag that carries the information regarding the change of the set of parameters of the white space model, and the first slot ID of each data period for which the predictions are required. The corresponding log lines start with CSO and CSP, respectively for receive and transmit slots. The predictions from the model are logged in the lines starting with ws, which include the first slot ID of the data period, whether the predictions are for transmit-/receive-slots, and the relative slot ID of the predicted FREE slots from the start of the data period.

1 90673525 APP seq 0
2 90681453 ws 871 P1,1,2
3 90769012 r_on
4 90773447 ws 971 P0,15,26
5 90868573 r_on
6 90886910 CSO:02:0:1171
7 90975439 r_off lr
8 91386574 CSO:02:0:1171
9 91886574 CSO:02:0:1171
10 92386574 CSO:02:0:1171
11 92886574 CSO:02:0:1171
12 95886678 CSP:01:0:1171
13 96386576 CSP:01:0:1171
97386712 CSO:02:0:1271
97886575 CSO:02:0:1271
98669868 r_on
98767753 r_off lr
100673540 APP seq 1
100681774 ws 971 P1,15,26
102164392 r_on
102169125 ws 1071 P0,37,60
102267390 r_off
102386821 CSO:02:0:1371
102886764 CSO:02:0:1371
103267042 r_on
103373222 r_off lr
105886691 CSP:01:0:1271
106386577 CSP:01:0:1271
108669868 r_on
108767753 r_off lr
110673540 APP seq 2
110682117 ws 1071 P1,37,60
114362920 r_on
114367601 ws 1171 P0,28,31
114386945 CSO:02:0:1471
114467270 r_off lr
114886794 CSO:02:0:1471
115386766 CSO:02:0:1471
115886766 CSO:02:0:1471
116386766 CSO:02:0:1471
116665479 r_on
116705575 ws 1171 P1,28,31
116772051 r_off
117386827 CSP:01:0:1371
117886702 CSP:01:0:1371
118386702 CSP:01:0:1371
118669997 r_on
118767400 r_off
118886716 CSP:01:0:1371
120673540 APP seq 3
123464365 r_on
123468952 ws 1271 P0,24,27
123569081 r_off
123767028 r_on
123873167 r_off lr
123886525 CSO:02:0:1571
125886612 CSP:01:0:1471
126386510 CSP:01:0:1471
126886510 CSP:01:0:1471
127386510 CSP:01:0:1471
127886617 CSP:01:0:1471
128386574 CSP:01:0:1471
128669925 r_on
128767753 r_off lr
128886588 CSP:01:0:1471
129386574 CSP:01:0:1471
129886574 CSP:01:0:1471
130386574 CSP:01:0:1471
130673597 APP seq 4
130682169 ws 1271 P1,24,27
130886746 CSP:01:0:1471
131386703 CSP:01:0:1471
131886703 CSP:01:0:1471
132386703 CSP:01:0:1471
133062196 r_on
133066729 ws 1371 P0,1,33
133165937 r_off
133365338 r_on
133386828 CSO:02:0:1671
133471100 r_off lr
133886506 CSO:02:0:1671
134386506 CSO:02:0:1671
134886506 CSO:02:0:1671
135386506 CSO:02:0:1671
135886506 CSO:02:0:1671
136386506 CSO:02:0:1671
136886506 CSO:02:0:1671
137886627 CSP:01:0:1571
138386513 CSP:01:0:1571
138669718 r_on
138767698 r_off lr
138886527 CSP:01:0:1571
139386513 CSP:01:0:1571
139886513 CSP:01:0:1571
140386513 CSP:01:0:1571
140673509 APP seq 5
Appendix F

Simulation Results

This section presents the performance of LUCID, CRYSTAL, and ContikiMAC in terms of PDR, duty-cycle, and energy efficiency ($= PDR/duty-cycle$).
### Table F.1 Comparison of PDR with the SoA.

<table>
<thead>
<tr>
<th>Network Size</th>
<th>Data Period (seconds)</th>
<th>Env Type</th>
<th>Mean PDR %</th>
<th>Standard Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LUCID (2FS)</td>
<td>LUCID (3FS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Crystal</td>
<td>Contiki MAC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Contiki MAC (retx=3)</td>
<td>Contiki MAC (retx=3)</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>HOME</td>
<td>Peak</td>
<td>90.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>92.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>93.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>94.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>HOME</td>
<td>95.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
<td>96.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>56.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>95.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>97.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.28</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>HOME</td>
<td>Peak</td>
<td>92.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>92.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>94.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>94.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>HOME</td>
<td>94.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
<td>97.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>55.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>97.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>97.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.67</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>HOME</td>
<td>Peak</td>
<td>87.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>90.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>89.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>92.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>HOME</td>
<td>90.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
<td>91.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>95.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>90.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>94.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>HOME</td>
<td>89.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
<td>92.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>90.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>53.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>90.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>91.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60</td>
<td>HOME</td>
<td>90.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
<td>93.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>95.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OFFICE</td>
<td>Peak</td>
<td>91.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
<td>95.46</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>96.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>88.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.50</td>
</tr>
</tbody>
</table>

- LUCID: Location Update Control and Information Distribution
- Crystal: CrystalMAC
- Contiki MAC: Contiki MAC with retransmission limit of 3
Table F.2 Comparison of *duty-cycle* with the SoA.

<table>
<thead>
<tr>
<th>Network Size</th>
<th>Data Period (seconds)</th>
<th>Env Type</th>
<th>Mean Duty-cycle %</th>
<th>Standard Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LUCID (2FS)</td>
<td>LUCID (3FS)</td>
</tr>
<tr>
<td>5</td>
<td>10 HOME OFFICE</td>
<td>Peak</td>
<td>0.84</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>0.84</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>0.84</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>0.84</td>
<td>0.57</td>
</tr>
<tr>
<td>60</td>
<td>10 HOME OFFICE</td>
<td>Peak</td>
<td>0.16</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>0.16</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>0.16</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>0.16</td>
<td>0.11</td>
</tr>
<tr>
<td>16</td>
<td>10 HOME OFFICE</td>
<td>Peak</td>
<td>1.26</td>
<td>1.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>1.26</td>
<td>1.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>1.26</td>
<td>1.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>1.26</td>
<td>1.76</td>
</tr>
<tr>
<td>60</td>
<td>10 HOME OFFICE</td>
<td>Peak</td>
<td>0.23</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>0.23</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Peak</td>
<td>0.23</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Off-peak</td>
<td>0.23</td>
<td>0.31</td>
</tr>
</tbody>
</table>
Table F.3 Comparison of energy efficiency (in terms of $PDR/duty-cycle$) with the SoA.

<table>
<thead>
<tr>
<th>Network Size</th>
<th>Data Period (seconds)</th>
<th>Env Type</th>
<th>Mean Energy Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LUCID (2FS)</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>HOME</td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>HOME</td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>HOME</td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>HOME</td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Peak</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off-peak</td>
</tr>
</tbody>
</table>