Eating of the pudding

Supporting the development life-cycle of
wireless sensor networks for
environmental monitoring scientists and ecologists

Kui Zhang

Eating of the pudding

Supporting the development life-cycle of wireless sensor
networks for environmental monitoring scientists and
ecologists

Kui Zhang

Graduation committee:

Chairman:	Prof.dr.ir. Peter M.G. Apers
Promoter:	Prof.dr.ing. Paul J.M. Havinga
Co-promoter:	Dr.ir. Nirvana Meratnia

Members:

Prof.dr.ir. Gerard J. M. Smit	University of Twente
Prof.dr. Andrew K. Skidmore	University of Twente
Prof.dr. Kirk Marinez	University of Southampton
Prof.dr. Antonio Liotta	Technical University of Eindhoven

CTIT

CTIT Ph.D. Thesis Series No. 15-378

Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands

This work is supported by the FP7 CLAM, Veiligheid op de Werkvloer and SaxShirt project.

ISBN: 978-90-365-3995-1

ISSN: 1381-3617

DOI: 10.3990/1.9789036539951

<http://dx.doi.org/10.3990/1.9789036539951>

Cover design: Fatjon Seraj

Printed by CPI - Koninklijke Wöhrmann

Copyright © Kui Zhang

EATING OF THE PUDDING

**SUPPORTING THE DEVELOPMENT LIFE-CYCLE OF
WIRELESS SENSOR NETWORKS FOR
ENVIRONMENTAL MONITORING SCIENTISTS AND
ECOLOGISTS**

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus
Prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Friday 20 November 2015 at 12:45

by

Kui Zhang

Born on 1 December 1985
In JingShan, China

Dit proefschrift is goedgekeurd door:

Promotor: Prof.dr.ing. Paul J.M. Havinga

Assistent-Promotor: Dr.ir. Nirvana Meratnia

Acknowledgements

In 2010, I was happy with the special job offer from IBM and Tencent after my got my master degree. We had a big party with all friends, classmates before I made a choose of my job offer. During the party all people were talking about their future plans, I was really shocked that 90% chose to go for a PhD abroad. I confused and struggled for one week and then made a decision I should also go for a PhD study abroad. Right after I made a decision there was the advertisement looking for PhD for underwater communication from PS group in the tccc mailing list. I did the skype interview and also came to Enschede for the final face-to-face interview. Time flies, now I am finalizing my PhD thesis.

During these years, many people helped me for my research and my life. My deepest gratitude goes first and foremost to my promotor Prof. Paul Havinga for his constant valuable guidance and suggestion. Although he is very busy, he is always willing to spend time to supervise and support me. In addition, I deeply appreciate that Prof. Paul Havinga also translate the abstract of the thesis into Dutch. I would like to express my heartfelt gratitude to my supervisor Dr. Nirvana Meratnia. She had spent a lot of time for helping me to revise my papers, to improve my writing skills. Without their patient instructions, insightful criticism and expert guidance, the completion of this thesis would not be possible.

I would like to express my gratitude to Dr. Peng Guo for his support and fruitful discussions and collaborations during these years.

To all (ex)members of PS group, I leave here my gratitude for providing such a wonderful working environment. Special thanks goes to Yang who helped me a lot with all kinds of issues. I also would like to thank Fatjon Seraj, Okan Turkes, Dr. Nirvana Meratnia and Prof. Paul Havinga for helping me designing the cover of the thesis, having fun together and giving valuable suggestions.

Special thanks go to my two strong paranimf Zhang Sheng and Fatjon Seraj. Thank you for your help and support.

Next, I would like to thank all partners in CLAM Veiligheid op de Werkvloer and SaxShirt project. It was fun and productive to work with most of them.

Also I want to thank Prof.dr.ir. Gerard J. M. Smit, Prof. dr. Andrew K. Skidmore, Prof.dr. Kirk Martinez and Prof.dr. Antonio Liotta for being part of my graduation committee. I really feel honored to have such experts in my graduation committee.

Thanks for all my wonderful friends here: xiao gang, xiao li, zhong hua, jing wei, hong lin, min min, hai rong, xiao bin, lei lei, mei nan, wang rong, xiao lin, zou yue tian, xiao hua, ke nan, yi jian, liang rong, gao qi, lan tian, du ying, dong fang, zou yu xin, su yu bo, dang wen qi, xiao yan, jia hui, fu yao, li guo, jia jie, hai shan, wei tao, mi yue, zhang sheng(my paranimf), chen ning, bo wen, xue long, zheng ding, xiao chao, gao peng, wu yingzhe, wang yi, ai jie, and... Thank you all for having fun together and making me feel at home.

The most special thanks to my great and lovely parents, thank you for your unconditional love, support and understanding. Thank you for providing me a friendly, lovely family atmosphere.

Finally, I want to thank my beautiful and kind fiancée xibo xiong. Thank you for your endless support and all kinds of nice food you prepared.

Kui Zhang

Enschede, October 2015

Abstract

In this thesis we present design and tooling solutions as well as network protocols to support application experts in the entire development life-cycle of wireless sensor networks. The complete life-cycle of wireless sensor networks starts with the user/application requirement analysis. It then goes through the following steps: hardware schematic design, PCB design, system software design, network protocol design, application software design, testing, debugging, and re-programming. In addition to being error-prone and time consuming tasks, all these steps require expertise, experience, in-depth knowledge about electronics and computer science, which application experts often do not possess. Instead of large quantities of WSN platforms with perfect design, application experts often require limited number of custom-designed wireless sensor node platforms, often not commercially available, for fast prototyping. They are interested in platforms that can be easily used and experimented with and can easily be adapted into their/applications requirements to find the best and most appropriate setup (software and hardware wise). As such availability of accurate and easy-to-use support tools, which can be used by application experts to translate their requirements easily and efficiently into a hardware schematic and associated software code and to make the entire process of hardware and software design easier, faster, and more efficient would greatly help bridge the gap between the demands of application experts and their WSN's expertise and knowledge.

The main focus of this thesis is on providing tools and mechanisms for application experts in general, and environmental and ecological scientists in particular, to support the entire life-cycle of wireless sensor nodes, facilitating and accelerating the process. To this end, the main research question addressed in this thesis is:

Can the entire process of design and development of wireless sensor networks be made efficient,

easy, fast, less error-prone and open to application experts in the field of environmental monitoring and ecology who have no a priori knowledge about WSNs?

This question relates to both hardware/software design of WSN platforms and network protocol design to make the network operational. As such, this thesis is organized into two design and tooling chapters (Chapter 2-3) and one network protocol chapter (Chapter 4). The contribution of this thesis can be summarized as:

- a hardware schematic design tool to facilitate the automatic generation of the schematic draft satisfying the systematic requirements formulated by the application experts and its evaluation of our automatic hardware schematic design tool in terms of effectiveness and efficiency using a proof-of-concept (Chapter 2). The main advantage offered by our contribution in this chapter includes reducing the design errors, since the generated hardware schematic contains more rich information than just a drawing of the hardware components and their pin assignments. It includes (i) type and functionality of each pin and (ii) type of software needed for CPU instantiation and device drivers. By doing so, hardware components cannot be left unconnected (which is one of the most common design errors) as the tool checks them all. The tool also makes the entire WSN hardware design easier for application experts since to use the tool no WSN knowledge and expertise is required. The tool is tested by application experts and WNS experts and the required time as well as the generated designs are compared with the output of the tool.
- tools to facilitate generation, debugging, testing, and re-programming of software system and to make these tasks easier, faster, more flexible, resource-aware, and less error prone (Chapter 3). To this end, we present (i) an automatic code generation tool for the software system template consisting of CPU initialization, device driver, service loop, event handler, and interrupt handler, based on the generated hardware schematics of Chapter 2 and user inputs, (ii) an online debugging tool to help easily modify and test the software system on the fly, and (iii) a re-programming tool to allow partial update of the binary image of the software system to reduce the re-programming time and effort. Evaluation of the tools has been done using a proof-of-concept.
- achieving a tradeoff between energy-efficiency and high packet delivery rates on the one hand and fast data dissemination on the other hand in large-scale low duty cycle WSNs (Chapter 4). Our target applications of environmental

monitoring and ecology have often large-scale static backbone infrastructures. To ensure that the network runs for very long time, a low duty cycle MAC is often utilized. This brings with itself the risk of either missing a packet sent by another node or delaying the message delivery and dissemination. To this end, in this chapter, we present three cross-layer MAC and routing protocols to achieve a good tradeoff between energy-efficiency, reliability, and delay for (i) static chain-based Wireless Sensor Networks (WSNs) with unreliable links, (ii) mobile WSNs interacting with none time-critical static backbone infrastructure, and (iii) large-scale time-critical event detection applications of WSNs.

Samenvatting

In dit proefschrift presenteren wij een keten van ontwerp- en ontwikkelhulpmiddelen voor draadloze sensor netwerken. Een typisch ontwerptraject start met een analyse van de gebruikers- en toepassingseisen. Vervolgens kunnen de volgende fases worden doorlopen: schematisch hardware ontwerp, printplaat ontwerp, systeemsoftware ontwikkeling, netwerkprotocol ontwerp, ontwikkelen van de toepassing, en vervolgens testen en debuggen. Al deze stappen in het ontwerpproces zijn foutgevoeling en kosten veel tijd. Daarnaast vereist het ontwerp specifieke kennis en ervaring op het gebied van elektrotechniek en informatica, iets wat een normale gebruiker van de technologie meestal niet (voldoende) heeft. Bovendien heeft een typische gebruiker vaak zeer specifieke wensen voor het ontwerp, waar commercieel verkrijgbare systemen niet aan kunnen voldoen. Wat zij feitelijk nodig hebben is een systeem waarmee ze makkelijk en snel een prototype kunnen maken, en er eenvoudig mee kunnen experimenteren. Het ontwerp moet eenvoudig kunnen worden ontworpen en ontwikkeld, en toegespitst op de specifieke eisen van hun toepassing. Een dergelijk set van gereedschappen voor ontwerp en ontwikkeling dat zowel de hardware als de software aspecten voor een groot deel uit handen kan nemen, is zeer waardevol, en kan de brug slaan tussen de wensen van de (eind)gebruiker en de vereiste kennis op het vlak van draadloze sensor netwerk platformen en netwerkprotocollen.

De kern van dit proefschrift beslaat dan ook de hiervoor genoemde aspecten, en omvat het onderzoek en ontwerp van een keten van ontwerpgeredschappen die in principe geschikt zijn voor een brede groep van gebruikers van draadloze sensor netwerken. In dit proefschrift richten we ons met name op gebruikers op het gebied van milieu, ecologie en biologie. De onderzoeksvraag dat in dit proefschrift wordt behandeld is dan ook:

Hoe kan het proces van ontwerp en ontwikkeling van een draadloos sensor netwerk voor gebruik in toepassingen voor het milieu, biologie en ecologie worden ondersteund met tools en methodieken dusdanig dat een (eind) gebruiker van dergelijke systemen deze eenvoudig, foutloos en efficiënt kan maken?

Deze onderzoeksvraag omvat zowel de hardware en software aspecten van draadloze sensor netwerk systemen, alsmede de netwerk protocollen om deze platformen efficiënt en effectief aan elkaar te kunnen koppelen. Dit proefschrift omvat dan ook twee delen. Het eerste deel omvat twee hoofdstukken over de benodigde ontwerp- en ontwikkelingsgereedschappen (Hoofdstuk 2 en 3), en het tweede deel betreft het onderzoek naar geschikte netwerkprotocollen (Hoofdstuk 4). De concrete bijdragen van dit proefschrift zijn:

- De ontwikkeling en evaluatie van een set ontwerpgereedschappen voor hardware-ontwikkeling waarmee op een semi-automatische wijze een schematisch ontwerp kan worden gegenereerd aan de hand van de specifieke eisen van de gebruiker (Hoofdstuk 2). De ontwikkelde methodiek waarbij het ontwerp extra meta-data bevat, zorgt ervoor dat ontwerpfouten worden geminimaliseerd. De meta-data betreft naast de specifieke functionaliteit van de componenten en aansluitingen ook de benodigde systeemsoftware en device drivers. De gebruiker van deze gereedschappen hebben hierdoor weinig specifieke technische kennis nodig. Het systeem is ontwikkeld, getest en geëvalueerd op een variëteit van ontwerpen en gebruikers.
- De ontwikkeling en evaluatie van gereedschappen waarmee op een effectieve, efficiënte en flexibele wijze de software van de ontworpen hardwareplatformen (ook wel genoemd sensor nodes) kan worden gegenereerd, getest, en hergeprogrammeerd. In Hoofdstuk 3 worden daartoe de volgende concepten gepresenteerd: 1) een semi-automatisch gereedschap voor het ontwerp van een raamwerk behorende bij het gegenereerde hardware ontwerp (uit Hoofdstuk 2) met daarin de essentiële systeemsoftware zoals initialisatie, device drivers, en event- en interrupt handlers; 2) een methode waarmee het systeem op een eenvoudige en snelle wijze kan worden getest en aangepast; en 3) een methode om specifieke gedeeltes van de programmacode van het systeem te vervangen, waardoor veel tijd en moeite kan worden bespaard tijdens het ontwikkeltraject. Al deze concepten en methodieken zijn getest aan de hand van een proof-of-concept.

- Een set van netwerkprotocollen voor data-disseminatie in draadloze grootschalige sensornetwerken waarbij er een uitgekende balans is tussen energie-efficiëntie en data transport. Om energie te besparen wordt meestal een lage duty-cycle MAC protocol gebruikt, wat tot gevolg kan hebben dat een bericht gemist kan worden, of dat er grote vertraging op kan treden. In Hoofdstuk 4 worden drie netwerkprotocollen besproken die de balans tussen efficiëntie, betrouwbaarheid, en vertraging maken voor i) een netwerk van een ketting van sensor nodes met onbetrouwbare verbindingen, ii) een statisch netwerk waarin mobiele sensor nodes bewegen die hun data aan het statische netwerk willen doorgeven, en iii) grootschalige netwerken voor toepassingen waarbij de betrouwbare detectie van events belangrijk is.

Contents

1 Introduction	15
1.1 The problem and target users	18
1.2 Research question	20
1.2.1 Hypothesis	21
1.3 Contributions of this thesis	21
1.4 Thesis organization	23
2 From WSN Application Requirements to Schematic Design: A System Approach	27
2.1 Hardware components of wireless sensor nodes	29
2.1.1 Specifications of hardware interfaces	31
2.1.2 Specifications of power supply	32
2.2 Automatic Hardware Schematic design Tool (AHST)	34
2.2.1 Overview	34
2.2.2 Modeling hardware components	34
2.2.3 Modeling requirements of hardware components and users	39
2.2.3.1 Modeling requirements of hardware components	39
2.2.3.2 Modeling user-defined requirements	41
2.2.4 Parsing	43
2.2.5 Finding optimal set of hardware components	43
2.2.6 Designing the hardware schematic	45
2.3 Proof-of-concept	47
2.3.1 Hardware design of an implant	47
2.3.1.1 User requirements $RE(S, t, p, r, d)$	49
2.3.1.2 Schematics	49

2.3.2 Hardware design of a Birdtag	49
2.3.2.1 User requirements $RE(S, t, p, r, d)$	49
2.3.2.2 Schematics	51
2.3.3 Evaluation of design time and design errors	52
2.4 Summary	52
3 Computer aided WSN software design	55
3.1 Challenges	57
3.1.1 Challenges faced by coding	57
3.1.2 Challenges faced by debugging	60
3.2 Software components in wireless sensor nodes	61
3.2.1 Programming and code generation	62
3.2.2 Debugging	65
3.2.3 Re-programming	66
3.3 Automatic code generation tool	67
3.3.1 Generated software modules	68
3.4 Easy and correct debugging	74
3.4.1 Debug communication	77
3.5 Graphical user interfaces	78
3.6 Partial (Re)-Programming	78
3.7 Proof-of-concept	80
3.8 Summary	83
4 Efficient network protocol design	87
4.1 A cross-layer MAC and routing protocol for low duty cycle static WSN with unreliable links and a linear topology	88
4.1.1 Model and assumptions	91
4.1.2 An opportunistic energy-efficient scheduling and routing for linear WSNs (OPS)	92
4.1.2.1 Scheduling of nodes with OPS	95
4.1.2.2 Optimization of OPS	99
4.1.3 Performance evaluation using simulation	102
4.1.3.1 Simulation setup	102
4.1.3.2 Simulation results	103
4.1.4 Performance evaluation using experiments	105

4.2 A cross-layer Media Access Control (MAC) and routing protocol for interacting mobile and static rendezvous	107
4.2.1 Applicability of pipeline scheduling for random data gathering . .	108
4.2.2 An energy-efficient and fast network protocol (MobiBone)	109
4.2.3 Performance evaluation using simulation	115
4.2.3.1 Simulation setup	115
4.2.3.2 Simulation results	116
4.3 A reliable and fast cross-layer MAC and routing protocol for large-scale time-critical event detection applications of WSNs	118
4.3.1 Problem formulation	121
4.3.2 A reliable and fast scheduling (RealFast)	122
4.3.2.1 Establishment of the traffic paths	122
4.3.2.2 Design of the wake up patterns	126
4.3.3 Performance evaluation using simulation	127
4.3.3.1 Simulation setup	127
4.3.3.2 Simulation results	129
4.4 Summary	133
5 Conclusion and future direction	135
5.1 Conclusions and lessons learned	137
5.2 Future work	138
Appendices	141
Bibliography	161

CHAPTER 1

Introduction

Deploying thousands or even millions of tiny and cheap sensors with wireless communication capability in an unattended and harsh environment to monitor physical phenomena and to detect events of interest in a self-organizing manner being reliably and robustly operational for many years has been the vision and the dream of many researchers in the field of wireless sensor networks (WSNs). Despite their tremendous efforts, this vision and dream has unfortunately not yet been fully realized due to various technical and scientific challenges. The most important challenges include:

- **Heterogeneity:** heterogeneity in wireless sensor networks can present itself in different forms ranging from heterogeneity in sensor types to heterogeneity of deployment areas, network scales, node densities, hardware platforms, and application domains.
- **Lifetime:** wireless sensor nodes are often battery-powered. Their operational lifetime is strongly dependent on the type of tasks they need to perform, the frequency by which these tasks should be performed, and the performance of protocols they run. Having larger batteries or battery capacities is not a proper option, as this will increase their size and cost.
- **Dynamicity:** the environment in which wireless sensor networks are deployed in is quite dynamic. This by itself introduces many changes in the network topology and network link quality. Moreover, a single network can be used by different users, each of which imposing different requirements on the network performance, data quality, and application quality of service requirements.

- Application dependency: wireless sensor networks are application dependent. This implies that methods and protocols designed for a specific type of application may not (directly) be applied to another domain. The same applies for hardware platforms and hardware components of the platform, which if well suited for a specific type of application does not guarantee their appropriateness for another application. This brings with itself the issue of re-design of the hardware platform and software systems.
- Hardware/software co-design: limited resources available on the wireless sensor network hardware platform requires a thorough understanding of hardware details and specification to design efficient and high performance software system and applications. This knowledge is often missing by the software application developers resulting in solutions that work well in simulation environments but fail in real world settings.

Despite of these challenges, wireless sensor networks have found their way into many application domains including:

- Environmental monitoring: was the very first public application of wireless sensor networks. The objective of environmental monitoring is to help human to understand the environment thoroughly and to give indications of what influences the environment. The most typical sensors used for this purpose are temperature, light, humidity, and CO₂ sensors. There are many sub-domains under the environmental monitoring application of WSNs including forest monitoring [15], climate monitoring [48], animal monitoring [26], and pollution monitoring [33,45]. Environmental monitoring applications have traditionally utilized a centralized data gathering scheme, in which sensor data has been periodically sent to a server. A great challenge faced by environmental monitoring applications is the unattended and often harsh environment, in which the network should function. The large scale of the environment to be covered necessitates utilizing a multi-hop network for this type of applications. Network density in this type of applications is rather low.
- Agriculture and farming: WSNs can be deployed in a farm to monitor environmental data such as temperature, light, soil moisture, pH value, strength of the wind, and soil chemical substances, which are all important for the grow of crops. There are many success stories about use of wireless sensor networks in agriculture and farming [19,39,50]. For example [11] showed in 2002 that Intel

established the world's first wireless sensor vineyards in Oregon. Sensor nodes were deployed in every corner of the vineyard to measure the soil temperature, humidity in the farm in a 1Hz frequency. In the end, the researchers found that the subtle climate change in the vineyard will greatly influence quality of the wine. The large scale of the environment to be covered necessitates utilizing a multi-hop network for this type of applications. Similar to environmental monitoring applications, network density in this type of applications is rather low.

- **Structure health monitoring:** Relatively low-cost, small size, and ease of deployment of wireless sensor nodes have made them a suitable alternative to costly, bulky, wired sensing devices and often visual inspection methods used traditionally for monitoring of structures and infrastructures [47]. The objective of structural health monitoring is to monitor the integrity of structures, to detect and to pinpoint occurrence and the locations of any possible damage. Typical sensors used for this purpose are temperature, accelerometers, and strain gauges. At certain periods of time, the sensor nodes transmit their data to a central sever for further analysis. In this case, sensors are required to work at high sampling frequency ($>$ hundreds of Hz), which results in generation of a large amount of data (about thousands or even tens of thousands bytes for each sensor in a single round of data collection process). All this data should be transmitted to a central location, which consumes a lot of energy. An alternative is to enable the sensor nodes to process the data locally and report only potential events, which does not occur very often in this type of applications. The large scale of the environment to be covered necessitates utilizing a multi-hop network for this type of applications, while in some cases single-hop communication between clusters of nodes and the central server has been used. While the network can have high density at critical points, the usual node density is rather low.
- **Health and well-being:** wireless sensor nodes can also be attached to different parts or implanted inside the human being to collect various types of data such as temperature, blood pressure, heart rate, breath rate, and motion [6]. This data is further processed to reason about persons' health condition, well-being, and activity level. Different types of sensors used for this application poses different requirements in terms of sampling frequency, reliability, data quality, and processing capability. The fact that this data is often required to be collected by yet another device carried/worn by people (like a smartphone)

requires compatibility of wireless communication module between the wireless sensor nodes and the device. Unlike the previously mentioned applications, the network scale in this application is small, though the dynamicity (due to human movement) is quite high.

- Underwater monitoring: wireless sensor networks have recently been also used for monitoring water quality and health and condition of underwater infrastructures such as pipelines [66]. The underwater world is much harsher than the air and by itself brings many different challenges ranging from water-proof packaging to resilience against pressure, extreme dynamic communication channel, and extremely low data rate. Typical sensors used in these type of applications are acoustic sensors, motion, pressure, and pH. The sensor data is usually transmitted to a surface buoy via acoustic channel. Unlike applications on air, underwater monitoring applications use acoustic signals instead of radio signals as the acoustic signal can reach further distance than the radio signal underwater. The power consumption of the acoustic-based communication is, however, much higher than the radio-based wireless communication. Moreover, the signal of one acoustic transmitter only can cover a certain angle, which means multiple transmitters should be used in order to cover a wide angle.

1.1 The problem and target users

From what was presented above, one can see that diversity of wireless sensor network applications imposes different requirements on sensing, communication, and processing capabilities of the wireless sensor nodes. Even within one application domain, various requirements need to be satisfied since there often exist different sub-domain applications under each application domain. To clarify this issue, let us consider the environmental monitoring application with animal monitoring as its sub-domain application. Monitoring the mobile animals and the environment in which they live introduces the following diversities and conflicting issues:

- Data gathering: periodic (e.g. from the environment and the animal), aggregated-based (e.g. from the environment), or event-based (e.g. from the environment in case of occurrence of an environmental hazardous or an animal passing by);
- Node density: high (e.g. around animal nests or feeding stations), low (e.g. in overall habitat), or mixed;

- Data resolution: high (e.g. in case of events), low, mixed;
- Data/network reliability: high (e.g. in case of events), low, mixed;
- Delay: tolerant (e.g. in case of periodic monitoring), real-time (e.g. in case of events), mixed;
- Scale: large (e.g. ground environmental network), small (e.g. network of mobile animals), hybrid;
- Mobility: mobile (network of mobile animals), static (ground environmental network), hybrid;
- Data traffic: periodic, bursty (event-based), random;
- Power source: battery (mobile tags), energy harvested, AC powered (e.g. gateways);

Application diversity also implies diversity of users, each of which having different expertise, background, and competence. Users of wireless sensor networks can be experts and non-experts. The difficulty here is that expert users from the application point of view (ecologists, biologists, environmental scientists, farmers, physicians, patients, etc) are often non-expert users from the WSNs point of view (those who have little or no in-depth knowledge about the (internal) mechanisms and fundamentals of WSNs).

Communication between WSN experts and application experts does not always go smoothly and easily and therefore translation of application/user requirements to specific hardware and software designs is not always flawless. The complete life-cycle of wireless sensor networks starts with the user/application requirement analysis. It then goes through the following steps: hardware schematic design, PCB design, system software design, network protocol design, application software design, testing, debugging, and re-programming. Being knowledgeable about all these steps is difficult even for WSN experts as it requires an all-inclusive approach ranging from electronic-domain specific knowledge to software engineering and network design up to application development. Rapid advancement of technology and availability of many different hardware components for sensing, wireless communication, and processing, makes it difficult to choose appropriate components from a wide range of choices. This requires having extensive insight on the one hand and reading many different datasheets on other hand. Normally the minimum number of datasheets to be read is about five per design. Reading datasheets of all commercially available

hardware components is of course not practical, leave only not being feasible. After reading these documents, 'best' option are decided by the designer according to their own experiences or certain constraints such as cost, size, and energy efficiency. However, no one can guarantee that these 'best' choice will remain the best in practice. After selection of these components, the real problem is how to power these components, how to connect them, how to configure them and how to access them. Unlike many other computing platforms, operation of software applications in WSNs is to a great extent restricted by the hardware platform capability and resources. As such many of solutions and techniques designed by researchers, although being successfully tested in simulation environments, fail in practice. Programming of wireless sensor nodes is difficult and requires special expertise and knowledge about low-level hardware knowledge not found in all high-level programmers. Moreover, it is time consuming and often cannot be done on the fly. Clearly all these steps require expertise, experience, in-depth knowledge about electronics and computer science, which application experts often do not possess.

The bottom line is that application experts often require custom-designed wireless sensor node platforms that are not commercially available. They also require fast prototyping and as such they are not willing to spend a lot of money and effort into making large quantities and perfect design. Instead, they are interested in platforms that can be easily used and experimented with and can easily be adapted into their/applications requirements to find the best and most appropriate setup (software and hardware wise). As such availability of accurate and easy-to-use support tools, which can be used by application experts to translate their requirements easily and efficiently into a hardware schematic and associated software code and to make the entire process of hardware and software design easier, faster, and more efficient would greatly help bridge the gap between the demands of application experts and their WSN's expertise and knowledge. This is the very goal of this thesis.

1.2 Research question

The main focus of this thesis is on providing tools and mechanisms for application experts in general, and environmental and ecological scientists in particular, to support the entire life-cycle of wireless sensor nodes, facilitating and accelerating the process. To this end, the main research question to be addressed is:

Can the entire process of design and development of wireless sensor networks be made efficient,

easy, fast, less error-prone and open to application experts in the field of environmental monitoring and ecology who have no a priori knowledge about WSNs?

This question can be broken down into various aspects of fast prototyping related to hardware design, system software design, network protocol design, and testing (including debugging and re-programming). To this end, in this thesis we answer the following sub-questions:

- Question 1: Can application experts be completely shielded from hardware details and be offered an automatic generated hardware schematic that completely fulfills their requirements?
- Question 2: Can system software for a specific hardware design be generated automatically, efficiently, and accurately and online be debugged without the need for extra hardware?
- Question 3: Can conflicting requirements (in terms of scale, dynamicity, duty cycle and dynamic data traffic) of environmental and ecology monitoring applications of WSNs integrating static large-scale WSNs and random mobile WSNs be simultaneously met with ?

1.2.1 Hypothesis

Our starting point to answer questions Q1 and Q2 is the following hypothesis:

- H1: *It is possible to support application experts in fast prototyping of WSNs platforms if hardware and software design are tightly coupled and hardware schematics, pin allocations and their connectivity, as well as code generation are done in an automatic and all-inclusive manner requiring little interaction from application experts.*

The hypothesis to answer Q3 is that:

- H2: *An integrated and cross-layer MAC and routing protocol design can fulfill conflicting requirements of static and mobile WSNs used in environmental monitoring and ecology applications.*

1.3 Contributions of this thesis

From the research questions stated above, one can clearly see that this thesis follows two stream lines, i.e., (i) design and tooling (Chapter 2 and 3), and (ii) research on

network protocols (Chapter 4). To this end, the contribution of this thesis can be summarized as:

- Chapter 2: modeling of characteristics and performance of major hardware components of wireless sensor hardware platforms, presenting a hardware schematic design tool to facilitate the automatic generation of the schematic draft satisfying the systematic requirements formulated by the application experts, and evaluation of our automatic hardware schematic design tool in terms of effectiveness and efficiency using a proof-of-concept. The key challenges to be addressed in this chapter include how to accurately and efficiently model the hardware components, their requirements, and the user-specified requirements, how to find the optimal solution satisfying the input requirements, and how to automatically generate the schematic? The main advantage offered by our contribution in this chapter includes reducing the design errors, since the generated hardware schematic contains more rich information than just a drawing of the hardware components and their pin assignments. It includes (i) type and functionality of each pin and (ii) type of software needed for CPU instantiation and device drivers. It includes the hardware components and their pin assignments. By doing so, hardware components cannot be left unconnected (which is one of the most common design errors) as the tool checks them all. The tool also makes the entire hardware design easier for application experts since to use the tool no WSN knowledge and expertise is required. The tool is tested by application experts and WSN experts and the required time as well as the generated designs are compared with the output of the tool.
- Chapter 3: tools to facilitate generation, debugging, testing, and re-programming of software system and to make these tasks easier, faster, more flexible, resource-aware, and less error prone. To this end, we present (i) an automatic code generation tool for the software system template consisting of CPU initialization, device driver, service loop, event handler, and interrupt handler, based on the generated hardware schematics of Chapter 2 and user inputs, (ii) an online debugging tool to help easily modify and test the software system on the fly, and (iii) a re-programming tool to allow partial update of the binary image of the software system to reduce the re-programming time and effort. Evaluation of the tools has been done using a proof-of-concept. Fast prototyping requirement of application experts implies that many of traditional challenges of WSN hardware design including interoperability, security, service discovery, memory management, and Internet connectivity managed by the operating

system are not required. As such, design, use, and porting OS, which is a relatively large task, for which application experts neither have the knowledge nor the expertise, and is not feasible for dedicated systems used for environmental and ecological monitoring is not required. The main advantages offered by the unique tight coupling between hardware design and code generation presented in this chapter include reducing complexity of software design, reducing the software errors, and speeding up life-cycle of software design.

- Chapter 4: achieving a trade-off between energy-efficiency and high packet delivery rates on the one hand and fast data dissemination on the other hand in large-scale low duty cycle WSNs. Our target applications of environmental monitoring and ecology have often large-scale static backbone infrastructures. To ensure that the network runs for very long time, a low duty cycle MAC is often utilized. This brings with itself the risk of either missing a packet sent by another node or delaying the message delivery and dissemination. Both of these situations can often occur in our target applications, in which an event (e.g. fire in a forest area) can occur for instance or presence of an animal should disseminated through the network as a trigger to wake up the sleeping nodes. Ensuring reliability, energy-efficiency, and fast data dissemination, a cross-layer design integrating MAC and routing is needed. To this end, in this chapter, we present three cross-layer MAC and routing protocols to achieve a good trade-off between energy-efficiency, reliability, and delay for (i) static static WSNs with unreliable links and a linear topology, (ii) mobile WSNs interacting with none time-critical static backbone infrastructure, and (iii) large-scale time-critical event detection applications of WSNs.

1.4 Thesis organization

This thesis is organized as follows shown in Figure 1.1. In Chapter 2, we present our support tool for the automatic hardware schematic design. To evaluate its performance, two generated hardware schematics designs, i.e. for an implant and a Birdtag together with the design time taken by two WSN experts and three application experts and the tool is presented. In Chapter 3, we present our support tool for automatic system software design, debugging, and re-programming. The tool is evaluated using a proof-of-concept. In Chapter 4, we present three different network protocols showing how a good trade-off between energy-efficiency, high packet delivery rate, and dissemination delay in low duty cycle large-scale hybrid (static and

mobile) wireless sensor networks can be archived. Performance of these protocols is evaluated through various rounds of simulations. A number of concluding remarks are presented in Chapter 5 to conclude the thesis. Figure 1.1 illustrates the relationship among different chapters.

Figure 1.1: Thesis organization

Part I: Design and tooling

CHAPTER 2

From WSN Application Requirements to Schematic Design: A System Approach

By focusing on our user target groups, i.e., application experts in the fields of environmental monitoring and ecology, in this chapter we present an automatic hardware schematic design tool, which is easy to use, reduces the time and complexity of hardware design process, and is less error-prone. Our tool shields the application experts from the low-level hardware details and supports them by enabling fast prototyping of WSNs dedicated to a special application domain (thus not full-fledged traditional wireless sensor nodes). The tool not only selects the best suited hardware components based on user/application requirements but also generates the hardware schematics and more importantly assigns the pins for each components. By doing so, it components cannot be left unconnected, which is one of the most common errors occurred during hardware design phase.

High-level software development is little, if any, challenged by the hardware architecture. In contrary, the hardware of wireless sensor nodes and its associated constraints impose restrictions on the performance of the high-level designed algorithms. This necessitates having at least some in-depth knowledge about the hardware design. This leads to a gap between high-level algorithmic design and available hardware resources. The rich electronic-domain knowledge required for the hardware/firmware design is often a great barrier for the application experts and contributes to enlarging this gap. Hardware design is the first step in the life-cycle of sensor platform design. WSNs are often application-dependent. This implies that different applications require different hardware platform and configuration. While some parts of a hardware design for a specific application (e.g. power management of a processor) can be re-used, some parts may need to be re-designed from scratch. To speed up the design

process, on the one hand, and to make the design process easy for (none)experts, there is a need for a tool capable of automatic generation of the hardware design directly from the user requirements.

While such a full-fledge tool does not currently exist, several tools for automatic *analog* circuit design are in use to reduce required design time and labor cost. IDAC [16], OASYS [27], OPASYN [52], ISAID [57] and DARWIN [38] are examples of automatic circuit design tools, to name but a few. All these tools make use of pre-defined libraries of (sub)-modules being given before the circuit design process. Techniques used in these tools to select filter component sizes and topologies and collaborators on analog circuit synthesis for automatic analog circuit design are often heuristics-based, knowledge-based and genetic algorithm-based [64].

Contributions of this chapter include:

- modeling of characteristics and performance of typical and major hardware components of wireless sensor hardware platforms.
- a hardware schematic design tool, which facilitates automatic generation of the schematic draft satisfying the system requirements formulated by the end-users.

In this chapter we introduce the automatic hardware schematic design tool and its performance in terms of effectiveness and efficiency using a proof-of-concept.

After requirement analysis and architecture design, the hardware design process starts. It consists of two tasks, i.e., (i) the schematic design (logic design) and (ii) the Printed Circuit Board (PCB) design. Schematic design is the blue print and structure of the circuit graph, which is drawn with pre-defined symbols of hardware components. It represents the structure and the functionality of the circuit system. Schematic design is a process of finding the proper choices of all hardware components and connecting them. Based on the schematic, it is easy to verify the correctness of the design and to analyze the behavior of the hardware before the actual hardware platform is manufactured. A correct schematic design is characterized by a circuit, which follows the circuit theory [30]. The PCB design, the other task of the hardware design, is a guideline of physical location of the actual electronic circuits/components. It represents the same logic as the schematic but the distribution of the components is quite different. This is because the main concern of the schematic design is to ensure the correctness of functionality and behavior of the circuits, while the concern of the PCB design is to reduce the overall size of the platform and the heat dissipation, to increase reliability, and to prevent anti-coupling, among other things. Despite the fact that there are many different designs, they all share a common design methodology,

which includes: (i) analyzing user requirements, (ii) selecting hardware components based on user requirements, (iii) design and verification of schematic design, and (iv) PCB design and manufacturing.

2.1 Hardware components of wireless sensor nodes

Our automatic hardware schematic design tool for wireless sensor network platform is based on the fact that most wireless sensor nodes have somewhat similar high-level hardware architecture, whose building blocks can be chosen from variety of hardware modules. Since more and more hardware circuits are being integrated into a single chip, functionality of an individual module may be found in a single chip. Since every chip has certain function, interface, and usage definition, the choice of appropriate chip is quite important.

All hardware platforms of wireless sensor networks share a set of the following basic hardware components:

Figure 2.1: Typical components of a WSN device

- Sensor is an important component of WSNs, which interacts with the physical phenomena. The choice of sensors is application-dependent. The power consumption of some sensors (e.g. chemical sensors) are often the largest source of power consumption in WSNs. Several aspects such as range, resolution (i.e., the minimum changes that can be captured by the sensor), power consumption, and interfaces need to be taken into consideration while selecting sensors.
- Wireless communication emerges today in the form of different technologies including WiFi, Bluetooth, Zigbee, sub-GHz radio, GPRS, GSM, infrared, acoustic. Not all these technologies are equally mature. All except acoustic communication have standards at the physical level, while only some (i.e., WiFi, Bluetooth,

Zigbee, and Infrared) have standard protocol stack. All these wireless communication technologies can be used in WSNs, but each has its own advantages and disadvantages in terms of energy consumption, range, throughput, to name but a few. It is therefore important to know features of each communication technology and requirements of the application at hand.

To select a wireless communication component, which meets application requirements, knowing the media type, frequency, bandwidth, communication range, and data rate is important.

- Processor is a large integrated circuit which is the core for computation and control of a computing device. The main components of the processor are mainly Arithmetic and Logic Unit (ALU) and Control Unit (CU). Additionally, the processor includes a set of registers, high speed cache, data bus, control bus and status bus. An embedded processor is different from a normal processor in terms of its structure and combination of register, cache, etc. There are three basic types of embedded processors, i.e., (i) MicroProcessorUnit (MPU), (ii) MicroControlUnit (MCU), and (iii) Embedded Digital Signal Processor (EDSP).

Processor consists of:

- Random Access Memory (RAM) exchanges data directly with the MCU. Since, it can be read or written on at any time and it is fast, it is used to store intermediate data for running programs. As it will be described in Chapter 3, every program has at least two segments, i.e., (i) the data segment and (ii) the stack segment being stored on the RAM. As size of the stack segment changes dynamically during program execution, RAM should have enough space to prevent the MCU from entering an unpredictable status if the stack overflows. If size of the RAM of the processor is not sufficient for WSNs applications, it should be replaced. However replacement of a RAM means replacing the entire MCU, since the processor used in WSNs is integrated with the RAM.
- Internal flash is a long lasting non-volatile memory. As it will be described in Chapter 3, all instructions are stored on internal flash and are read from internal flash during program execution. Internal flash can generally only be programmed via a certain flash programmer, but nowadays more and more MCUs provide the capability to program the internal flash.

The choice of the processor has to be made after sensors, wireless communication, and size of the internal flash have been decided. This is because to choose

an appropriate processor, computation capacity, number of supported interfaces, minimum required amount of internal flash, and power consumption need to be taken into account. The maximum speed of executing the software system by the processor is expressed by the maximum system clock of the processor on the datasheet. As a general rule of thumb, the more powerful the processor, the higher the energy consumption.

- Local storage is required by many, if not all, WSNs applications. Large storage spaces are in the form of external flash memory or Secure Digital (SD) card. Flash memory can be directly connected to the processor via Serial Peripheral Interface (SPI) or Inter-Integrated Circuit (I2C), while SD card normally needs special chip to support the SD card logic. The principle for choosing the storage devices depends on the amount of data, which needs to be stored. The capacity of the SD card is much larger than the flash memory. What storage types are required are often specified by the user.

Design of both hardware and software of wireless sensor nodes put forward many considerations and requirements. Hardware design is not just about selecting several hardware components and connecting them together. Various types of interfaces have their particular communication (e.g., universal asynchronous receiver/transmitter (UART), I2C) and electronic standards (e.g., voltage and current), which make some hardware components incompatible to be connected with each other. Special low-level programs for example setting timers and clock need to be specially developed for the designed hardware so that the application-level programs can be supported.

In this section, we first analyze the most typical specifications and requirements of developing wireless sensor node hardware platforms and then show how our proposed automatic hardware schematic design tool takes these specifications into account and conforms to the requirements.

2.1.1 Specifications of hardware interfaces

Once all hardware components are selected, a standard way is needed to connect them together. Defining a standard physical (connectors) and/or logical (software) connections between different components is called interfacing. Software interfaces are needed since there may not be enough physical interfaces available on the PCB. In this case, if enough General-purpose input/output (GPIO) pins are available, a particular physical interface can be emulated using a software interface. There exist

several different types of interfaces, out of which UART, SPI, I2C, Universal Serial Bus (USB), Ethernet, and analog, are the most commonly used in WSNs.

2.1.2 Specifications of power supply

The hardware components described above may have different requirements in terms of (i) power supply and (ii) communication interface.

Different chips being used for a wireless sensor node platform have different power supply requirements. Some chips need 3V, while others need 5V for example. The power source in wireless sensor networks can only provide one power level. Therefore, to accommodate different power supply requirements on a single platform, a power converter is needed.

As it can be seen in Figure 2.2, there are three basic types of power converters:

- In the first type of converters, shown in Fig 2.2a, the input is either GND or V_{cc1} . The output will then be GND or V_{cc} , respectively. The advantages of this kind of voltage converter are (i) simplicity and (ii) bi-directionality which means that it can convert from low voltage to high voltage and vice versa. The disadvantage of this type of converter is, however, that it takes a relative large space in the PCB layout.
- In the second type of converters, shown in Fig 2.2b, the input is either V_{cc1} or GND , while the output voltage is V_{cc} or V_{D1} , respectively. V_{D1} is the forward voltage drop of the diode D_1 . Similar to the first type, this type of converter is also simple but there are two constraints: firstly V_{cc1} must be greater than V_{cc} and secondly the low voltage threshold of the circuits should be greater than V_{D1} .
- The third type of converters, shown in Fig 2.2c, is actually a buffering circuit. For this type of converters, V_{cc1} must be greater than V_{cc} . The buffering circuit can be the voltage converter in a certain range since the input of the buffering circuit is only Transistor–transistor logic (TTL) or Complementary metal–oxide–semiconductor (CMOS) circuits.

Figure 2.2: Power converter

Figure 2.3: Overview of automatic hardware schematic design tool (AHST)

2.2 Automatic Hardware Schematic design Tool (AHST)

2.2.1 Overview

Figure 2.3 presents the main concept behind our automatic hardware schematics design tool. As it can be seen, first a model library (to be described in Section 2.2.2) will be created, which contains models of various off-the-shelf hardware components, their functionality, their specification, and performance. This model library together with the user-specified application requirements described using our descriptive model (to be explained in Section 2.2.3) are input for the optimization algorithm (to be described in Section 2.2.5) that aims to find a set of optimal solutions from the model library. Members of the selected set satisfy the input (i.e., user-specified application) requirements with the minimum hardware cost. After that, the optimal solution as well as the datasheet information of the corresponding selected hardware components will be used as the input to the automatic hardware schematic design tool, which automatically generates the schematic of the input systematic solution of the wireless sensor devices. The key challenges to successfully do so are addressing the following questions: (i) how to accurately and efficiently model the hardware components, their requirements, and the user-specified requirements?, (ii) how to find the optimal solution satisfying the input requirements?, and (iii) how to automatically generate the schematic? In what follows we answer these questions and provide a detailed description of various phases of our automatic hardware schematics design tool.

2.2.2 Modeling hardware components

We define a descriptive way to model each hardware component. Our model consists of description of each component in the form of key-value pairs. The mandatory key-value pairs for each component are:

- Name: product name of the component;
- Type: component type, to be selected from Sensor, Processor, Wireless communication, Storage, and Actuator;
- Accuracy: only applicable to sensor types ;
- Power: is the required power by the component and has the following fields:
 - Voltage: voltage range that can be applied to the component;

- Current: current range that can be applied to the component;
- Pin configuration: the following fields need to be specified per pin:
 - Index: index of the pin in the component;
 - Type: type of the pin, which can be one of the following types. The default types can be extended if needed (e.g. USB).
 - * VCC: voltage of the circuit;
 - * VS: supply voltage;
 - * VDD: internal working voltage of the component;
 - * GND: common ground for the circuits;
 - * DGND: digital ground;
 - * AGND: analog ground;
 - * SGND: signal ground;
 - * UARTTXD: UART transmitter;
 - * UARTRXD: UART receiver;
 - * SPIMISO: 4 wire SPI master-input slave-output;
 - * SPIMOSI: 4 wire SPI master-output slave-input;
 - * SPIDATA: 3 wire SPI data;
 - * SPICLK: SPI clock;
 - * IIC_SCL: I2C clock;
 - * IIC_SDA: I2C data;
 - * GPIO: GPIO;
 - * ANALOG: analog pin;
 - * RESERVED: reserved pin;
 - * NC: not connected internally;
 - * CS: chip select;
 - * INT: interrupt;
 - * JTAGTDI: test Data In for Joint Test Action Group (JTAG);
 - * JTAGTDO: test Data Out for JTAG;
 - * JTAGTCK: test Clock for JTAG;
 - * JTAGTMS: test Mode Signal for JTAG;
 - * JTAGTRST: test Reset for JTAG;
 - Name: function name, which specifies the core functionality of the pin;

- Direction: direction of the pin (i.e., in, out);
- Flotation: an indication whether the pin should be float;
- Power: required power by the pin and has the following fields:
 - * Voltage: voltage range that can be applied to or provided by the pin;
 - * Current: current range that can be applied to or provided by the pin;

List 2.1 illustrates an example of our model for ADXL345 motion sensor. As it can be seen, each pin can have different functions, which all need to be described in the model.

Listing 2.1: Filled-in model template for ADXL345 motion sensor

```
1 {
2   "Name": "ADXL345",
3   "Type": "Sensor",
4   "Voltage": "2.0V- 3.6V",
5   "Current": "0.1uA - 23uA",
6   "Pins": [
7     {"Index": "1", "Functions": [{"Type": "VDD", "Description":
8       "Digital Interface Supply Voltage"}]},
9     {"Index": "2", "Functions": [{"Type": "GND", "Description":
10       "Must be connected to the ground"}]},
11     {"Index": "3", "Functions": [{"Type": "Reserved",
12       "Description": "Must be connected to Vs or left open"}]},
13     {"Index": "4", "Functions": [{"Type": "GND", "Description":
14       "Must be connected to the ground"}]},
15     {"Index": "5", "Functions": [{"Type": "GND", "Description":
16       "Must be connected to the ground"}]},
17     {"Index": "6", "Functions": [{"Type": "VS", "Description":
18       "Supply Voltage"}]},
19     {"Index": "7", "Functions": [{"Type": "CS", "Description":
20       "Chip select"}]},
21     {"Index": "8", "Functions": [{"Type": "INT", "Description":
22       "Interrupt output 1"}]},
23     {"Index": "9", "Functions": [{"Type": "INT", "Description":
24       "Interrupt output 2"}]},
25     {"Index": "10", "Functions": [{"Type": "NC", "Description":
26       "Not connected"}]},
27   ]
28 }
```

```

17  {"Index": "11", "Functions": [{"Type": "Reserved",
    "Description": "Must be connected to ground or left
    open"}]},
18  {"Index": "12", "Functions": [{"Type": "SPIMISO",
    "Description": "Serial data output for \gls{spi} 4
    wires"},
19    {"Type": "ALT ADDRESS", "Description": "Alternate
    \gls{i2c} address select"}]},
20  {"Index": "13", "Functions": [{"Type": "IICSDA",
    "Description": "Serial data for \gls{i2c}"},
21    {"Type": "SPIMOSI", "Description": "Serial data input for
    \gls{spi} 4wires"},
22    {"Type": "SPIDATA", "Description": "Serial data input and
    output for \gls{spi} 3wires"}]},
23  {"Index": "14", "Functions": [{"Type": "IICSCL",
    "Description": "Serial communication for clock
    \gls{i2c}"},
24    {"Type": "SPICLK", "Description": "Serial communication
    clock for \gls{spi}"}]}],
25 }

```

Each hardware component can be modeled either by the user by filling the above model template or by directly parsing the datasheet. As shown in Figure 2.3, the model of all hardware components should be added to their corresponding model library. Regardless of how the model is made, each model should be translated to an internal data structure. We call this step ‘parsing’. All the functions of a specific pin and all pins in a hardware component are managed as a double linked list. The functions to handle the double linked list is show in Appendix 5.2. We define the format of the internal data structure as:

Listing 2.2: Core internal structure for hardware component

```

1  struct list_head {
2      struct list_head *next, *prev;
3  };
4
5  typedef struct func {
6      struct list_head func_list;

```

```
7     char *name;
8     int function_type;
9     int direction;
10    char *description;
11 } func_t;
12
13 typedef struct pin {
14     int index;
15     func_t funcs;
16     int flotation;
17     float voltage[2];
18     float current[2];
19     struct list_head pin_list;
20 } pin_t;
21
22 typedef struct hwcomponent {
23     char *name;
24     char *model;
25     int pin_total;
26     pint_t pins;
27     float voltage_range[2];
28     float current_range[2];
29     char *datasheet_url;
30     int rating;
31     char *feedback;
32     struct list_head component_list;
33 }hwcomponent_t;
34
35 typedef struct hwdesign {
36     char *name;
37     hwcomponent_t components;
38 }hwdesign_t;
```

Structure of each pin specified in the List 2.2, has the following important fields:

- Index: index of the pin of the hardware component. It will be used for drawing the hardware component itself and for the schematic design;
- Functions: function field contains three types of information:
 - Name: name of the function;

- Type: type of the function which can for example be GPIO, Clock (clock), I2C;
- Direction: direction of the pin;
- Flotation: flotation of pin;
- Voltage and current: voltage and current provided or applied to the pin;

Structure of component specified in the listing 2.2, has the following important fields:

- Name and model: the combination of name and model specifies a certain hardware;
- Total number of pins: total number of pins on the hardware component;
- pins: header of the double linked list described above;
- Power characteristics: the voltage and current requirements of the hardware component;

Optional fields of the structure component are: (i) 'url of the component's datasheet', (ii) 'rating', and (iii) feedback based on user experience. To select the best hardware component from both technical specification point of view and actual experience of the users, 'rating' and 'feedback' fields are foreseen. Rating is defined as a value between 0 and 10, the higher the value the most satisfactory the component is from the user experience point of view. The actual experience of the user can be described in the feedback field. This information will be used at later stage to make the final selection of the hardware components for the schematic design. Parsing process checks whether all mandatory fields of the internal data structure are (correctly) filled in. If so, the model will be sent, verified and stored in the back-end database.

2.2.3 Modeling requirements of hardware components and users

2.2.3.1 Modeling requirements of hardware components

One of the inputs required for the automatic hardware schematic design tool is the model of functionality, specification, and performance of the hardware components as well as their operational requirements. We group various hardware components of a wireless sensor node into sensing, processing, and wireless communication. In what follows, we describe how to model each of these three hardware components and their operation requirements.

- Modeling sensors: Different types of sensors may be needed for a single WSN platform, each of which requiring different interface resources from the processor. We define the model of the sensing module SE as:

$$SE : SE_i(A, p, c, n, m)$$

, where $SE_i(A)$ denotes a set of sensing capabilities such as temperature sensing, light sensing, and pressure sensing. $SE_i(p)$, $SE_i(c)$, $SE_i(n)$, and $SE_i(m)$ denote power consumption, hardware cost, number of required hardware interfaces, and number of required pins of the sensor, respectively;

- Modeling processors: The processor has various interface resources and processing/computation capabilities. The WSNs platform requires the processor to be low power. Therefore we focus on the interface resources of the processor instead of variety of its computation capabilities. Consequently, we define the model of the processor PR as:

$$PR : PR_i(p, c, n, m)$$

, where $PR_i(p)$, $PR_i(c)$, $PR_i(n)$, and $PR_i(m)$ denote power consumption, hardware cost, number of interface resources, and number of available pins of the processor, respectively;

- Modeling wireless communication: Parameters for modeling wireless communication are power, cost, transmission rate, transmission range, and interface resources. The transmission range usually depends on the power of the selected wireless communication component, its frequency and the receipt sensitivity. We define the model of the wireless communication RA as:

$$RA_i(p, c, r, f, s, n)$$

, where $RA_i(p)$, $RA_i(c)$, $RA_i(r)$, $RA_i(f)$, $RA_i(s)$, and $RA_i(n)$ denote power consumption, hardware cost, transmission rate, radio frequency, receipt sensitivity, number of interface resources, respectively. Power consumption is an approximation of the transmission and reception powers. Given $RA_i(p)$, $RA_i(f)$, $RA_i(s)$, we can calculate the transmission range d of the wireless communication component as follows:

$$d(RA_i(p), RA_i(f), RA_i(s)) = 10^{(RA_i(p) - RA_i(s) - 20 \lg RA_i(f) - 32.44) / 20} \quad (2.1)$$

2.2.3.2 Modeling user-defined requirements

The user-defined requirements are usually related to expected sensing capabilities (sensors), transmission rate, transmission range, and the overall power consumption of the wireless sensor devices. To obtain a uniform formulation of the user-defined requirements, we define the RE as:

$$RE(S, t, p, r, d)$$

, where $RE(S)$, $RE(t)$, $RE(p)$, $RE(r)$, and $RE(d)$ denote expected set of sensing capabilities, radio type, power consumption, transmission rate, and transmission range, respectively. One should note that the end-user may require multiple sensing capabilities (sensors) and even multiple sensors for the same sensing capability. Therefore, $RE(S)$ may contain multiple elements with the same value.

For each of hardware components described in Sections 2.1.1, users are given the option to specify application/user requirements. The user requirements should be described using the descriptive language proposed below. To do so, the user can choose between (i) a script template, an example of which is illustrated in List 2.3, or (ii) a GUI, an example of which is illustrated in Figure 2.4.

Figure 2.4: Graphical user interface for requirement

Listing 2.3: Script template for defining user requirements for an implant

```

1  Sensor accelerometer;
2  accelerometer.Type: digital;
3  accelerometer.Axis: 3;

```

```
4  accelerometer.Resolution: 1mg;
5  accelerometer.Range: -8g ~ 8g;
6  accelerometer.Size: 4 * 5 * 2;
7
8  Sensor temperature;
9  temperature.Type: analog;
10 temperature.Accuracy: 0.3;
11 temperature.Range: -10 ~ 100;
12
13 Radio radio;
14 radio.Type: 433MHz;
15 radio.Range: 20m;
16 radio.Speed: 200kbps;
```

Regardless of which option the user chooses, he will be given the following options to describe requirements for each hardware component:

User-defined requirements for the sensors are:

- Accuracy: is a measure indicating how close a sensor measurement is to the real physical value. Accuracy is an application dependent requirement. This means that a single sensor type (e.g. temperature) may be required to have high accuracy in a certain application (e.g. fire detection) and a low/average accuracy in another (structural health monitoring);
- Range: expected value range of the sensor output;
- Resolution: is the minimum change which can be captured by the sensor;
- Sampling frequency: required frequency at which sensor measurements need to be made;

User-defined constrained for the wireless communication component are:

- Type: which can be Zigbee, WiFi, Bluetooth, GSM, etc;
- Communication range: expected range between two wireless communication component. For this constraint, user specify the point-to-point communication distance he would like to bridge;
- Data rate: expected data rate of the wireless communication component;

- **Compatibility:** if there is already another communication technology in use, with which the 'to be designed' platform should be able to communicate, user needs to specify the compatibility constrain. To do so, he can choose the type of wireless communication technology to be compatible with, which can be WiFi, Zigbee, Bluetooth, and sub_GHz;

Response time is an important factor for the user. It relates to how fast the processor can perform its tasks. This depends on (i) main data type to be sensed, communicated, and processed, and (ii) intensity/complexity of tasks being executed. The latter highly depend on the sensor and is expressed by number of bits required to represent a single data instance. The former highly relates to the speed of the processor. The processor's speed also depends on the main data type and the word length of the processor. The word length of the processor is the maximum number of bits that a processor can process in a cycle. If the word length is shorter than the main data type, the processor will perform slower as it requires more cycles to handle the data. User-defined constrained for the processor are:

- **Main data type:** user can choose between data types of 8-bits (e.g. temperature, humidity) and 16-bits (e.g. motion). One should note that 32-bits have not been included because of it is considered as over-done in terms of both power consumption and data type support for environmental monitoring and ecology applications of WSNs;
- **Processor speed:** user can specify how fast the processor should run;

2.2.4 Parsing

After user/application requirements are specified, they should be translated to an internal data structure readable and understandable by the tool. To do so, a parsing tree will be created based on the specified user requirements. To create the parsing tree, first a lexical analysis will be performed using which the user defined requirements will be divided into tokens. Thereafter, a syntax analysis will be performed on the derived list of tokens to arranges them in a tree structure. Figure 2.5 illustrates an example of such parsing tree.

2.2.5 Finding optimal set of hardware components

After modeling hardware components and user-defined requirements, the next step is to find the optimal set of hardware components, which satisfy both user-defined

Figure 2.5: An example of parse tree

requirements and hardware design requirements, and have the minimum hardware cost. One should note that the term optimal here refers to the most optimal set of hardware components available in the database. To this end, the problem at hand is:

Given: requirements $RE(S, p, r, d)$, and model library SE, PR, RA , where $SE = SE_i | 1 \leq i \leq N$, $PR = PR_j | 1 \leq j \leq M$, $RA = RA_k | 1 \leq k \leq K$ are the library model of sensing, processing, and wireless communication component, respectively, we need to

$$\text{Minimize. } [\sum SE_i(c) + PR_j(c) + RA_k(c) * \frac{RE(r)}{RA_k(r)}] * \frac{RE(d)}{d(RA_i(p), RA_i(f), RA_i(s))} \quad (2.2)$$

$$\text{S.T. } \left\{ \begin{array}{ll} \Theta \subseteq SE & (1) \\ RE(A) \subset \bigcup_{SA_i \in \Theta} SA_i(A) & (2) \\ \sum_{SA_i \in \Theta} SA_i(p) \leq RE(p) & (3) \\ \sum_{SA_i \in \Theta} SA_i(n) \leq PR_j(n) & (4) \\ \sum_{SA_i \in \Theta} SA_i(m) \leq RR_j(m) & (5) \end{array} \right. \quad (2.3)$$

The second constraint in the problem means that the selected set of sensors denoted by Θ should have all sensing capabilities (sensors) defined in $RE(A)$. The last two constraints in the problem mean that the interface resources of the selected processor should satisfy the requirement of the selected sensors.

To solve the problem, we use a brute force algorithm which will have a polynomial time when the set of sensors Θ is given. That is to say, when selecting any Θ satisfying

constraints (1) and (2), we can search processor denoted by PR_j in the model library of processors of PR and wireless communication component denoted by RA_k in the model library RA . The optimal processor and the optimal wireless communication component for a given Θ are those satisfying constraints (3), (4), and (5) with the minimum cost (computed based on the objective function above). In other words, for any Θ , there are unique candidates corresponding the optimal PR_j and RA_k . For the sake of convenience, we denote the corresponding minimum objective cost as $f(Re, \Theta)$. It can be seen that the time complexity of finding the optimal set of PR_j and RA_k for the a given Θ satisfying RE is $O(M * K)$.

Now we only need to solve a sub-problem, which is: **how to find the optimal $\Theta \subseteq S$ satisfying $RE(A) \in \cup_{SE_i \in \Theta} SE_i(A)$, with which $f(Re, \Theta)$ is minimized?**

This is a typical set cover problem [35], which is NP-complete. To solve this problem, we propose a dynamic programming algorithm. The basic idea of this dynamic programming algorithm is that when selecting sensors, processors, and wireless communication components to satisfy $RE(A)$, we always search the sensors in SE for those components which have the maximum sensing capabilities defined in $RE(A)$ and have the minimum cost. This idea is based on the observation that integrated sensors having more sensing capabilities is usually cheaper and requires less interface resources than a number of individual sensors. We define our search algorithm based on dynamic programming as:

If the search results in more than one appropriate set of hardware components satisfying user requirements, they all will be provided to the user who finally needs to make a choice. To help the user make the best choice, 'rating' and 'feedback' fields as described in the hardware model will be provided to the user. In case, no matching hardware is found but the user knows one, he has the possibility to create and upload the model of this component.

2.2.6 Designing the hardware schematic

All components selected should be connected to the processor via certain interfaces. The pins of selected processor are represented as a tree structure as shown in Figure 2.6.

There should be enough interfaces for all hardware components being connected to the processor. Connecting components to the processor should be done in an efficient way, which implies to use as many as possible hardware interfaces instead of software emulated solutions. One should note, however, that choosing software emulated interfaces offer the advantage of reducing the size, which some applications may

Input: RE and model library of sensors SE
Output: Optimal set of sensors Θ

$\Theta = \Phi$;
 $F(RE(A), |RE(A)|)$;
Function $F(C, t)$ is defined as follows:
if $B = \{SE_i | S_i \in SE, SE_i(A) = C\} \neq \Phi$ **then**
 Search the element $SE_* \in B$ with minimum $f(RE, \Phi \cap SE_*)$;
 $\Theta = \Theta \cap \{SE_*\}$;
else if $B = \{SE_i | S_i \in SE_*, |SE_i(A)| = t - 1\} \neq \Phi$ **then**
 Search the optimal elements of $SE_* \in B$ as well as the corresponding
 optimal elements of $SE'_* \in SE$ and $SE'_*(A) \supseteq C \cap \overline{SE_*(A)}$ having the
 minimum $f(RE, \Theta \cap \{SE_*, SE'_*\})$;
 $\Theta = \Theta \cap \{SE_*, SE'_*\}$;
else
 for ($j = t - 2; j \geq 1; j--$) **do**
 if $B = \{SE_i | SE_i \in SE, |SE_i(A)| = j\} \neq \Phi$ **then**
 Search the elements of $SE_* \in B$ having the minimum cost, and
 $\Theta = \Theta \cap SE_*$;
 Execute $F(C \cap \overline{SE_*(A)}, t - j)$;
 end
 end
end

Algorithm 1: Searching for optimal set of sensors Θ

Figure 2.6: An example of tree model for MSP430F5438A processor

require. On the other hand, hardware interfaces have in general better performance in terms of speed and real-time execution. To determine how to connect hardware components to each other, we define an 'interface scheduler'. The interface scheduler first searches all possible interfaces for each hardware component and then it connects each interface of a certain hardware component to its corresponding interface on the processor.

In case there is a conflict between two different hardware components in terms of using the same pin, the following three solutions may be utilized:

- Switch one of the components to another connection found by the interface scheduler;
- If there is no alternative found, a bus solution (e.g. SPI, I2C) can be used to be connected to multiple components;
- If none of the above works, either the processor should be changed or a software emulated interface should be used;

2.3 Proof-of-concept

As a proof-of-concept for our automatic hardware schematic design tool, in this section, we present two examples, i.e. an implant and a Birdtag. The reason to choose these two designs is that both are used by our target application experts to monitor two types of animals. Activity level of badgers together with their body temperature are monitored using the implant node, while motion and activity level of snake-eagles are monitored using the Birdtag. Compared with the Birdtag, the hardware design of the implant node has to fulfill a different set of requirements both in terms of performance and hardware components.

2.3.1 Hardware design of an implant

Applications of implantable sensor networks in the health-care and animal monitoring domains have increased tremendously over the last decade. However the in-body communication mechanisms are not fully understood. To be able to model the communication channel in and around the body in a realistic setting, we designed and developed an implant, shown in Figure 2.7. In what follows, we present user requirements for this design and output of our automatic hardware schematic design tool.

Figure 2.7: Implant platform PCB (left), with coating (middle), and inside meat during experiment (right)

Figure 2.8: Birdtag hardware platform PCB

2.3.1.1 User requirements $RE(S, t, p, r, d)$

- Sensors (S): temperature and motion;
- Radio type (t): 433Mhz;
- Processor (p): 12 MHz (to process motion data online);
- Transmission rate (r): 200kbps;
- Transmission range (d): long range;

2.3.1.2 Schematics

Below the generated schematic design of the required hardware components and their connections for the implant node is illustrated. The figure contains (a) the processor integrated with the radio, (b) temperature sensor, (c) accelerometer sensor, (d) gyroscope, and (e) the power management. As it can be seen, the generated design matches the user requirements in terms of hardware component and CPU performance.

2.3.2 Hardware design of a Birdtag

To enable online tracking of snake-eagles traveling across Europe and from Europe to Africa and back, we design and developed Birdtag, shown in Figure 2.8. In what follows, we present user requirements for this design and output of our automatic hardware schematic design tool.

2.3.2.1 User requirements $RE(S, t, p, r, d)$

- Sensors (S): GPS and motion;
- Radio type (t): GSM;
- Processor (p): 12 MHz (to process motion and GPS data online);
- Transmission rate (r): 100kbps;
- Transmission range (d): long range;

Figure 2.9: Generated Implant schematic

2.3.2.2 Schematics

Below the generated schematic design of the required hardware components and their connections for the Birdtag node is illustrated. The figure contains (a) the processor, (b) GPS sensor, (c) GSM module, and (d) power management. As it can be seen, the generated design matches the user requirements in terms of hardware component and CPU performance.

Figure 2.10: Generated Birdtag schematic

2.3.3 Evaluation of design time and design errors

To evaluate the effectiveness of our tool, we have asked a number of WSN experts and application experts to complete the design of the two hardware platforms (i.e., the implant and the Birdtag). Results of these evaluations are shown in Table 2.1 and Table 2.2 for the design time (expressed in days) and the design errors made, respectively. As it can be seen, using AHST, a time saving of approximately 100% and 68%, 74% can be achieved by the application experts and WSN experts to design the implant and Birdtag WSN hardware platform. In fact, application experts could not finish the design. Additionally, while the design of both WSN experts fulfilled the user requirements in terms of the required hardware components (even though the type of hardware component being selected were not the same), their design contained some errors. These errors include forgetting (i) to connect chip select pins and (ii) to add pull up or down resistors. The AHST tool improves the design accuracy of the implant and Birdtag in average by 75% and 50%, respectively, for the two WSN experts.

2.4 Summary

The rich electronic-domain knowledge required for the hardware/firmware design is often a great barrier for application experts. To enable this group of users to easily develop, prototype, and experiment with wireless sensor nodes without having extensive electronic-domain knowledge, in this chapter, we presented an automatic hardware schematic design tool. This tool allows easy translation of system requirements formulated by the application experts to the schematic draft. Finding the optimal hardware design based on input requirements and characteristics of the hardware components is an NP-complete optimization problem. To solve this optimization problem, we utilized a greedy algorithm, which utilizes some domain-context information. Application experts can interact with the tool through either a scrip template or the graphical user interface. As a proof-of-concept, the hardware schematic for two different platforms, namely, a Birdtag and an implant, have been designed by the tool, based on which prototypes were manufactured, tested, and are currently in different projects in use. As other performance evaluation metrics, the time required to make these designs and the errors made during the design by two WSN experts, two application experts, and the tool were reported, which show the effectiveness, ease of use, less error prone characteristics of our tool.

Table 2.1: Comparison of time (expressed in days) taken to design schematics for the implant and the Birdtag nodes

	Expert index	Implant device	Birdtag device
WSN expert	1	3	4
	2	5.5	6
	3	2	3
	4	3	3.5
	5	3	4
	6	2.5	5
	7	2	4
	8	5	3
	9	2.5	4
	10	3	2
Application Expert	1	∞	∞
	2	∞	∞
	3	∞	∞
	4	∞	∞
	5	∞	∞
	6	∞	∞
	7	∞	∞
	8	∞	∞
	9	∞	∞
	10	∞	∞
	AHST	1	1

Table 2.2: Comparison of design errors made during the schematic design for the implant and the Birdtag nodes

	Expert index	Implant device	Birdtag device
WSN expert	1	2	0
	2	0	1
	3	0	0
	4	1	0
	5	0	0
	6	1	0
	7	0	1
	8	0	0
	9	0	0
	10	0	0
Application Expert	1	∞	∞
	2	∞	∞
	3	∞	∞
	4	∞	∞
	5	∞	∞
	6	∞	∞
	7	∞	∞
	8	∞	∞
	9	∞	∞
	10	∞	∞
	AHST	1	1

CHAPTER 3

Computer aided WSN software design

Wireless sensor network platforms are becoming more heterogeneous, supporting many different components, and performing various tasks. Whereas in the early days all sensor node designs were derivations from the good-old mote from Berkeley, current sensor node designs show a plurality of processors, wireless communication, and sensing devices. Porting the various OS for sensor nodes is a relatively large task, and is not feasible for most dedicated systems, especially those that we target in this thesis (i.e., environmental monitoring and ecology). This makes the role of the software system even more essential since the power and capability of the software system determines the actual functionality of the WSNs system.

Different applications of wireless sensor networks have different requirements in terms of cost, size, power consumption, quality of service, etc, on both hardware platform and software system of the WSNs. The WSN *system software* is responsible to operate and control the WSNs hardware platform, to provide resources for the application software, and to manage its execution. This is often referred to as the 'kernel'. Similar to the PCs, in which the operating system takes care of operation of hardware devices (e.g. video card, sound card, WiFi card) and memory management, the WSNs system software performs the same task. This means that the system software of WSNs is closely linked to both the hardware platform and the application software running on top of the system software. The system software of WSNs should be designed modular so that it can be easily used for different WSN hardware platforms, have small size and low resource consumption, and should offer reconfiguration possibility.

As discussed in Chapter 2, from hardware point of view, there are many different design choices for selecting processors, wireless communication components, and sensors. The system software should support the hardware components selected

during the hardware design phase and should be designed accordingly. The AHST tool presented in Chapter 2 finds the most suited set of hardware components satisfying user/application requirements and assigns pin to each and every hardware component. By doing so, it minimizes the possibility of making hardware design errors and also reduces the time and complexity of the hardware design. Automatic generation of device drivers and their initialization is the topic of Chapter 3. The wide range of hardware choices requires a thorough understanding of each component and its compatibility with the other components. It also requires extensive coding for each of many selected hardware components. The fact that the coding is highly prone to errors, makes it an overwhelming process for even those with many years of experience leave alone our user target group, i.e. application experts with no or little experience and knowledge about hardware and software design of WSN platforms. To this end, a computer aided software design tool, which supports the entire development life-cycle of the software system is essential. Since the fast prototyping requirement of application experts implies that many of traditional challenges of WSN hardware design including inter-operability, security, service discovery, memory management, and Internet connectivity managed by the operating system are not required, the software system as addressed in this chapter is different from the traditional operating systems of WSNs. This difference relates to that fact that compared with traditional WSN operating systems, our software system is more resource-lean because (i) there is no memory management and file system, and (ii) partial (re)programming and debugging is possible on the WSN platform through simplified management of the program layout and the debugging information. The complete program layout and the debugging information is shifted to the PC side.

Similar to Chapter 2, this chapter is a design and tooling chapter. The main question to be addressed here is what hardware/software co-design mechanisms are needed to allow code generation matching the hardware design specification, its initialization, debugging, and (re)programming of WSNs platform less error-prone, faster, and easier for application experts with no a priori knowledge about hardware/software design. To this end, contributions of this chapter include:

- an automatic code generation tool for the system software template consisting of CPU initialization, device driver, service loop, event handler, and interrupt handler. The tool allows easy, flexible, fast, and correct automatic code generation tool based on hardware schematics and user inputs. This technique will take care of generation of device drivers and their initialization for all the hardware components of the hardware schematic generated in Chapter 2. By

doing so, it enriches the hardware design.

- an online debugging tool, which helps the application experts easily modify and test the software system on the fly. This enables ease of debugging and code adaption.
- a (re)programming tool, which allows partial update of the binary image of the software system to reduce the re-programming time and effort.
- evaluation of the tools using a proof-of-concept.

In this chapter we introduce the automatic code generation tool, an online debugging tool and a (re)programming tool and the evaluation of these tools using a proof-of-concept.

3.1 Challenges

In this section, we briefly describe the most common challenges faced during the design and development of the WSN system software.

3.1.1 Challenges faced by coding

The processor used in WSNs of our target applications is in general of small size, has low power consumption, and is highly integrated as it contains the flash (flash), RAM, and occasionally the wireless module. The WSNs software is normally stored on the internal storage of the processor to ensure its fast and reliable execution. Small size of the WSNs hardware platform necessitates efficient design of both hardware and software system without having redundancy while ensuring high performance. To support the long life-cycle of the WSNs applications on the one hand, and to support our target users who have no or little experience and knowledge about software design of WSNs and therefore can make many errors and have changing requirements on the other hand, the software system may need to be often updated. As such the software system should be capable to support application and hardware requirements changes. It is nearly impossible for the user to change the functionality of the system after the design phase. Therefore, a dedicated development tool is needed for generation and debugging of the software system for WSNs.

The software system generally has the following building blocks:

- Core system software: includes codes for *CPU initialization, event handling, and interrupt handling*. It manages the system resources including processor execution time, RAM, and hardware timers. The actual management of system resources depends on the actual processor used since different processors have different hardware configuration, RAM layout, hardware timer modules, etc. Core system should provide basic and platform-dependent functions such as initializing clocks for different modules in the system and timer as well as task scheduling. The clock dividers in the MCU can be used to generate different frequency clocks based on the clock source. In WSNs, there are often two main different types of tasks, i.e., (i) *time-based tasks* such as reading sensors in a certain interval, and (ii) *event-based tasks* such as triggering an alarm upon occurrence of a special/certain event. Normally several hardware timers are available on the processor. The challenge is then how to effectively manage these hardware timers. This is because some hardware timers may need to be used to capture or compare sensor signals, which means that there will not be enough hardware timers. In this case virtual software timers are needed to be able to generate the timer functionality based on a certain hardware timer. The core system software can be either Operating System (OS) dependent or independent. The functions required to be implemented in the core system include, watchdog configuration, IO ports configuration, crystal configuration, clock configuration, and driver framework initialization. Scheduling of tasks to be performed can be done in (i) non-preemptive First-In-First-Out (FIFO), (ii) Earliest-Deadline-First (EDF), and (iii) event-based.
- Device drivers: which provide *initialization of the device and device operation* functions such as read, write, and configure. Device drivers are also different from system to system as the device can be connected to processor in different ways. Some drivers are platform-dependent while others are platform-independent. Operations needed to be supported by the device driver include hardware power control, software power control, device interface initialization, read and write, and power profiling.
- User-defined application: which has the code for the *service loop* to provide an environment for storage of user-defined parameters/requirements and execution of user-defined applications.

The normal process of developing the WSNs system software template consisting of CPU initialization, device driver, service loop, event handler, and interrupt handler

by system developers is:

- to understand the entire system design, especially the hardware schematics;
- to understand the exact functionality and interface of different hardware components used in hardware design. The current practice to acquire such understanding is reading extensive datasheets;
- to design and implement the *core system software*, which by itself is a complicated process consisting of the following steps:
 - providing default vector table: the vector table provides the jump address if a specific interrupts occurs. All entries of the vector table can be overwritten. The address of the vector table can also be changed to a customized address.
 - providing default interrupt handler: which normally sets a flag to trigger an event and to clear the interrupt flag;
 - stopping watchdog: during the start process the watchdog should be turned off, otherwise it can reset the processor in many cases;
 - initializing the .data segment: this segment is the place on the RAM to store static, global, and static local variables;
 - setting the .bss segment to zero: this segment contains all global and static variables that are initialized to zero or do not have explicit initialization in source code;
 - initializing the stack pointer: the stack is to provide space for function calls, intermediate variables, etc. The stack pointer also has a default value per processor and can be changed;
 - initializing clock system: after all the above steps are performed, the clock system should be configured in order to make the processor run as expected. Configuration of the clock system requires many detailed calculation to set the registers to achieve the required frequency of the clock;
 - configuring timers: this is to configure the hardware timer, watchdog timer, and possibly the software timer system which is using the hardware timer. Configuration of the hardware timers requires many detailed calculation to set the registers to achieve the required timer interval;

- to design and implement the *device driver*: this is to (i) initialize the communication interface between the processor and the peripheral devices and (ii) provide operation function for the peripheral devices;
- to provide an environment for user-defined applications using the *service loop*: which takes care of execution of event loops being triggered by interrupts and periodic tasks such as periodic sampling. Network protocol stack to enable wireless communication and creation and execution of application software are placed in the service loop.

The common problems faced by designing software for the above-mentioned tasks include: (i) use of different clock sources by the Central Processing Unit (CPU). This will make calculation of the expected clock frequency difficult; (ii) if the scheduling is done from scratch, the amount of details to be taken care of for each platform may be overwhelming. If on the other hand a specific OS is used, the problem is the lack of generality, which means that the knowledge and expertise of one OS and platform cannot be applied to another; (iii) either a hardware interface or a software-emulated driver is required for initializing devices. Diversity of devices used in wireless sensor networks challenges correct, easy, and fast implementation of the required drivers; and (iv) similar to the problem faced by the device initialization task, diversity of networking protocols challenges correct, easy, and fast implementation. The scheduler and the network protocol stack can be used as common modules for different platforms as they are hardware independent.

As it can be concluded from the above observations, writing the software system codes is a complex and error prone task. As it will be discussed in Section 3.3, this process can be made faster and more reliable through a computer-aided software design tool.

3.1.2 Challenges faced by debugging

Application users of the WSNs, who require fast prototyping and easy experimental tools, often need to use the network for different application scenarios, sensor configurations, geographic locations, etc. This necessitates frequent modification and testing of the system software, network protocols, and applications, to ensure correct and robust performance of the programs. To help testing the logic of the programs and to find possible errors and bugs therein, a debugging tool which monitors and controls (i.e., suspends, continues, stops, checks registers and memory values) the execution of the programs at run-time is used. The debugging tools are executed on a Personal

Computer (PC). The current debugging approaches for software system of the WSNs require special hardware and frequent erase/write of the flash. This means that before the actual debugging process starts, the image file (i.e. the compiled version of the software system converted from binary format to compatible format of the processor) should be written into the internal flash of the processor using an external hardware programmer/debugger (device). This dedicated hardware interface has typically no other use after the debugging is done. After that, the processor will load instructions from the flash and will execute the instructions under the control of the hardware debugger. Meanwhile, some internal state information of the hardware platform, e.g., state of registers, stack information, will be sent to the debugger via the debugging interface (e.g. JTAG), which in turn will send this state information to the debugging PC. During this process, inconvenience and difficulties to debug the software system using the debugger are often experienced. This is because each time that the code needs to be modified, the device should be re-programmed. This is a time consuming task because firstly the flash should be erased before it can be re-written on and secondly the re-programming itself takes quite some time. Moreover, frequent erase/write steps shorten the flash lifetime and possibly damage its sectors. As it will be discussed in Section 3.4, this process can be done in a faster and online manner. One should note that the focus of our debugging is on the system software and device driver codes. Methods already used for application-level debugging in distributed or deployed WSNs do not relate to this discussion. This is because the application-level debugging requires a wireless communication between the debugging PC and the target WSN node to wirelessly access the WSN nodes already deployed. Before nodes can be wirelessly accessed, they need to have correct and verified system software and device drivers.

3.2 Software components in wireless sensor nodes

Software is a set of instructions and data, which represent the user logic in a computer understandable way. As software represents the same logic as the human, the creation of software is a translation from user logic to computer logic. Normally, this translation is done via a certain programming language and a compiler. The programming language defines the formal way to express the human logic in the programming language and the compiler translates the human readable logic written according to the programming language defined rules to the computer understandable form, also known as instructions. As explained previously, these instructions are stored in the

storage devices such as flash, hard disk and SD card, and the intermediate data are stored in the RAM. Software execution has certain requirements in terms of storage, processing time, and RAM, which are collectively called 'run time' requirements. Generally the software run time is calculated by the OS, which manages and monitors all resources available on a platform.

3.2.1 Programming and code generation

Programming of WSNs is typically very close to the OS, therefore it requires programmers to focus on the low-level system development. On the one hand, this low-level programming distracts the programmer from the real application requirements and logic, while on the other hand, it requires in-depth knowledge and expertise about low-level hardware and software design. One of the commonly used approaches for WSNs programming is the macro-programming [21,23], which is in principle a network programming technique. It uses high-level programming languages to specify network operations. While the communication does not need to be explicitly specified, it is part of the execution of the language-specific instructions. This means that their focus is not on the individual network devices, but rather on the network as a whole. This implies that there is an underlying assumption that devices have already the basic functionality in terms of resource management, device drivers, and network protocol stack.

Developers of high-level programs do not need to think about where to store the code and how the code will be loaded and executed. In WSNs programming, however, the layout of the code and in particular the following components should be carefully taken into consideration:

- Interrupt vector table, which describes the address of each interrupt handler to be executed when the interrupt is triggered. It is stored in the internal flash and the addresses vary from MCU to MCU;
- Instructions, which are translated and generated by the compiler and stored in the internal flash;
- Run-time intermediate data, which is generated and used by variables during the execution of the program. It remains in the RAM in the heap, stack, or register spaces.

The operating system manages issues such as where the program should be loaded, when, where, and how the program should be executed, and how much

memory can be used by the program. The essential functionalities of an OS are scheduling, memory management, network protocol support, device management, and file system support. OS architecture can be in the form of monolithic, micro-kernel, virtual machine, and layered. In monolithic architecture, OS provides services together with their interfaces to be used by other services in the system. Multiple services can be bundled together into a single system image. This will reduce the OS memory footprint. All services of the OS are implemented in the same address space which results in more system efficiency compared with implementing in different address spaces. Running on a single address space has the risk that the whole system crashes even if only a bug in a single service occurs. Micro kernel only provides basic services for such as process management, IPC (Inter-Process-Communication), and memory management. All services including device drivers are executed in the user mode which is the same as any other applications running in the system. This is because each service has its own address space, which makes all services being separated from each other. The micro kernel architecture is easy to extend and simplifies the application development work. The user application only needs to run the required service. This can reduce the requirement of the storage and ram. The disadvantage of this architecture is, however, that the performance can be low as all user applications need to switch to kernel context as the user applications need to use the service provided by the OS. Virtual machine provides services and their interfaces in an hardware-dependent fashion. The advantages of this architecture is its portability while its main disadvantage is the poor system performance in terms of high memory footprint, high CPU utilization, and execution time. The layered OS architecture implements all services in the layers form. It offers a good resource management functionality in the cost of inflexibility of the architecture. The scheduler functionality of the OS should provide fairness in terms of using system resources, maximizing the overall performance while minimizing the system response time. Memory management refers to the strategy used to allocate or de-allocate memory for different tasks which can be processes or threads. The existing memory management techniques are either static or dynamic. The static memory management means that the memory is allocated before the system execution while the dynamic memory management occurs during the run-time. In addition to the memory allocation, memory protection is also important. It refers to the protection of process's address space from one process to another. In the OS context, communication refers to inter-process communication within the system as well as among different devices in a network. Therefore, the OS should provide API that can enable application program

to facilitate network communication.

The OS used in wireless sensor networks should be efficient by itself and small in size and its resource consumption. The most typical operating systems used in wireless sensor networks are:

- TinyOS [40] which is a monolithic OS. It provides a way of programming with provider and consumer components in the system. The provider components provide certain functionalities and can be used by the consumer components. The user components can use the available provider components and provides feedback to them. Early version of TinyOS supported non-preemptive First-In-First-Out scheduling algorithm [40]. This implied that it did not support real-time applications. The kernel of TinyOS executes in the order of FIFO manner. Therefore, an important feature of the TinyOS is 'run to completion', which means that all tasks cannot be preempted. The FIFO scheduling can be unfair to the tasks inserted to the queue later. This is not efficient in situations in which a short task has to wait for longer tasks to finish. In [34], a Earliest-Deadline-First (EDF) scheduling algorithm is added to TinyOS to support real-time applications. The EDF scheduling algorithm does not produce a feasible schedule when tasks compete for resources.
- Contiki [17] which is a lightweight OS implemented in pure C for wireless sensor nodes. It is highly portable and the kernel is an event-driven kernel. It provides multitasking and can be used at an individual process level. It follows the modular architecture. The kernel follows the event driven model but it provides optional threading facilities to individual process. In the kernel there is a lightweight event scheduler which dispatches events to running processes. Process execution is triggered by events dispatched by the kernel to the processes or by a polling mechanism. The schedule event always runs to completion but the event handler can use internal mechanism for preemption. Two different types of events are supported by Contiki, i.e., the asynchronous and synchronous events. The difference between the two is that the synchronous events are dispatched immediately to the target process that causes it to be scheduled while the asynchronous events act like deferred procedure calls that are en-queued and dispatched later to the target process. The polling mechanism can be seen as high-priority events that are scheduled in between two asynchronous events. All polled processes will execute the poll handler in order of their priorities. Contiki supports preemptive multi-threading which is implemented as a library on top of the event-driven kernel. The library can

be linked to those applications which need multi-threading support. Internally, Contiki implement the multi-threading by using protothreads [18] which is designed for resource limited devices because it shares the same stack for different thread and very lightweight (two bytes per thread).

WSN applications should be kept simple to be comply with WSN resource constraints. Additionally to ensure energy-efficiency, WSN software system should directly interact, control, and manages the hardware resources. As such, use of full-fledged operating system which creates a layer of abstraction is not always needed. In this thesis, we focus on those applications that are dedicated, relatively simple and do not benefit from a full-fledged OS .

3.2.2 Debugging

Although software engineering is a very well developed discipline and there are many well-documented procedures for all its aspects ranging from requirement analysis up to implementation and programming, it is still difficult to guarantee that all functional and non-functional requirements are fully covered. Even after a careful design process, the designed software may still have some errors or bugs in particular due to deployment issues which did not occur in a lab-test or even the user/application requirements may still change. These are the main motivations for an on site debugging. Here by errors, we refer to the software run-time errors, which are mainly man-made or are caused because of not occurrence of an un-expected or unforeseen situation.

To find and correct errors and bugs occurring during programming of WSNs, many useful tools are nowadays available for PC based software debugging such as Visual Studio for Windows-based softwares, GDB for Linux-based softwares, Xcode for MAC OS-based software, and Java tools for all platforms. All of these tools not only provide the basic functionalities of the debugging such as set breakpoints, watch variables, monitoring stack, but also support more advanced features such as power profiling, performance checking, and bottleneck detection. These debugging tools are OS dependent. OS handles the memory management for the program and the low-level operations. Debugging for the WSNs software system is however quite different. First of all, the limited memory does not allow running a debugger together with the software system on the WSNs hardware platform. Secondly, the OS of WSNs is not mature by itself and may itself need debugging. Debugging of WSNs software is usually done in a separated manner, which means that the debugging software runs

on a computer while the actual instructions are executed on the MCU of the WSNs device. The debugging computer sends instructions to the MCU and receives back the status information from the MCU. Debugging in WSNs is done either hardware-based or software-based.

JTAG is the standard IEEE 1149.1, which defines the standard debugging techniques for processors. It is the most commonly-used debugger/protocol for the WSNs software programming and debugging and is mainly used for internal testing of chips. Most of the modern chips such as DSP, FPGA, ARM, and micro-controller support JTAG. JTAG is not only defined for testing the electronic characteristics of the chip but also can be used for debugging or programming the flash devices. The internal registers of the CPU could be accessed via the JTAG if the clock signal is correct and if the CPU has the JTAG debug module.

The main disadvantages of current hardware-based debugging techniques are: (i) the need for extra hardware for re-programming and debugging, which has no use after debugging is done, and (ii) the need for the erasing/writing the binary image needs from/to the flash before the debugging process, which reduces life time of the device.

Bootloader is a program to load kernels and to write the image files to the flash and to execute the code on it. The Bootloader setups the environment for the kernel execution and copies the kernel into memory. It also transfers the control from the Bootloader to the kernel after the boot and provides all information required for the kernel to work properly. There are many different kinds of bootloaders in the field of embedded systems including Redboot [4], ARMboot [1], and U-Boot [5]. All these programs boot the OS and support file-system and user applications. The bootloader is highly hardware dependent and therefore it is not possible to create a universal bootloader for all embedded devices. Using the Bootloader, the binary images can be written to the flash or the SD-card via UART, USB, or Ethernet.

The advantage offered by software-based debugging techniques is that there is no need to use any hardware debugger which makes the JTAG pins free for other tasks such as connecting to sensor, actuator, and radio devices.

3.2.3 Re-programming

Re-programming deals with changing the logic of the software system of a WSN device. One should note that here we do not deal with network re-programming that involves re-programming of an already deployed WSN. It is the process of writing the binary image of the software system to the device flash memory. Re-programming

of WSNs devices is done using (i) full image replacement, (ii) diff-based approaches, and (iii) ELF-based dynamic load approach.

Flash memory has three types of operations: read, write and erase. Different from magnetic disks, flash memory must be erased before being re-written. The basic unit to be erased on the flash memory is a sector, which can be several hundreds of bytes or several kilobytes.

The MCU used in WSNs devices is often integrated with the NOR flash as the storage for binary images. If the binary image needs to be updated, the NOR flash needs to be first erased and then reprogrammed. In the most common way of updating a new version of image, the flash image is entirely replaced. However, it is usually only a small part of the software system image that needs to be changed due to fixing bugs. The minor difference between the new version and the old version of the binary image can be updated via the diff-base approaches such as [32] [37] [51]. Most of WSNs operating systems including TinyOS [40], Contiki [17], BTNut [2], and MANTIS [7] are statically-linked, which means that all functions and variables are copied into an stand-alone image file. The OS, however, should have the capability to load dynamically-linked programs. Therefore, the OS should be able to locate functions from the dynamic-linked libraries. Based on this concept, existing re-programming approaches are either based on erasing and re-writing the entire image file (i.e., statically linked) or based on Executable and Linkable Format (ELF) [62] (i.e. dynamically linked). In the ELF-based approaches, the ELF-loader runs on the processor and parses the image to locate all the symbols and functions. The drawback of these approaches is that every image should include symbols and function information including addresses. This makes the image quite large. Another drawback of ELF-based approach is that they only support re-programming of the application software but not the low-level re-programming.

3.3 Automatic code generation tool

To tackle the challenges faced by coding, as mentioned in Section 3.1.1, we propose a generic technique to allow easy, flexible, fast, resource-aware, and correct computer-based code generation. The generated code is processor-centric and supports all the interfaces of the processor which are used by various components of the wireless sensor node shown in Figure 2.1. By doing so, diversity of external devices and radio does not impose any challenge to the design of the software system and the generated code remains generic. A unique feature of our automatic code generation

technique is the tight coupling it offers between the hardware schematic design and the device driver generation and initialization. By doing so, the hardware schematic design generated in Chapter 2 will be enriched by extra information related to the device-drivers, which are mcu-dependent.

As shown in Figure 3.3, the input sources for our automatic code generator are (i) the generated schematics from Chapter 2, (ii) the hardware database (described in Chapter 2), and (iii) user configuration. The user configuration relates to the user choice of system clock source, running frequency of the processor, hardware interface or software emulated interface and their respective configuration (interface type, pins, speed, polarity and phase) for device driver, interrupt pins, MAC protocols (basic TDMA vs. CSMA), and the supported OS (optional). As described in Chapter 2, the current practice in hardware design is that after the hardware schematic is designed, it is read by the software system developer to understand the connection between different hardware components and then to design the software system based on this information. In contrary to this current practice, the output of our schematic design tool is not only the hardware scheme illustrating the connections between hardware components, but also computer-readable information that can be parsed by our code generator and automatically translated into the software system.

3.3.1 Generated software modules

The automatic code generation tool generates the following four software modules: system initialization module, device driver, the service loop, and event-triggered task. Figure 3.1 illustrates the logic of the automatic generated code.

1. System initialization module: Using the generated hardware schematics and the hardware database of Chapter 2, the first step is to extract processor information from the schematics. Extracted information include processor model and processor's memory layout, which will in turn be used to query the hardware database to get (i) information about the toolchain to be used for compiling the source code and for converting the binary file to the processor compatible image file, (ii) system clock library, and (iii) device interface library. The schematic design is parsed and a device tree as shown in Figure 3.4 is generated. The device tree illustrates all the hardware components in the schematic design and their connections. Figure 2 in and Figure 1, Appendix 5.2, show all the connections in terms of power control pin, communication interface, and corresponding pins of the hardware components connected to the processor of the MCUs of the two

Figure 3.1: Logic of the automatic generated code

hardware schematic designs (i.e., Figure 1 and Figure 2) mentioned in Chapter 2. The system clock library has *ConfigureClock(runningfrequency, clocksource)* format. The device interface library has five hooks in the following formats:

Open(), *Configure(interfacetype, pins, speed, polarityandphase)*, *Read(address)*, *Write(address, value)*, and *Close()*.

After the code generation engine parses the device tree in a specific design, the code design tools will be used to interact with software developers to make several choices. The two choices to be made by the developer are related to the system clock source and expected clock frequency. The system clock source can be a low precision digital crystal integrated with the MCU or an external high precision crystal. To be able to decide which of these two options to select, the power consumption and precision needed by the system should be taken into account. The system clock frequency determines how fast the processor can perform the calculation therefore it should be selected considering power consumption, response requirement of the system, and clock requirement of the external devices.

Using all three aforementioned input sources, the next step is start-up code related to the processor. It will handle the interrupt vector table, stack, and heap initialization. This will be followed by generating the code for the system clock management. The system clock library is the framework of the system clock management. The input from user configuration related to the system clock source together with the running frequency of the processor are input to the system clock library and will instantiate the code for system clock management through *ConfigureClock* hook. Figure 3.2a (left) illustrates the vector table of the interrupt handler. The code generated for the system clock of the two hardware schematic designs (i.e. Figure 1 and Figure 2 mentioned in Chapter 2) can be found in Appendix 5.2.

2. Device driver: Using schematics and the user configuration input related to the device interfaces, code will be generated either based on hardware interface or the software emulated interface. For the hardware interface, the device interface library extracted in system initialization module and the corresponding user configuration will instantiate the code template. For the software-emulated interface, the user configuration input related to pins and

the type of interface will instantiate the code. Instantiation of the code template for both hardware and software interfaces will be done through *Open()*, *Configure(interface type, pins, speed, polarity and phase)*, *Read(address)*, *Write(address, value)* and *Close()* hooks. The driver interface for debugging purpose will be initiated. Figure 3.2b (middle) illustrates the vector table of the communication library for the device driver. The code generated for the device driver interface of the two hardware schematic designs (Figure 1 and Figure 2) mentioned in Chapter 2 can be found in Appendix 5.2.

3. Service loop: The service loop is the place holder for the software application. The logic of the software application will be placed within this *while* loop, which will never end. Figure 3.1 illustrates the service loop.
4. Event-triggered task: To enable event-triggered tasks, the code generator should also support interrupt handlers. To do so, processor information extracted in Step 1 together with the user configuration related to interrupt pins will be used to instantiate the interrupt handler hook. The interrupt handler hook has *SetFlag(flag)* format. Figure 3.2c (right) illustrates the timer-triggered event handler.

Figure 3.2: Logic of the generated code

The flow chart of our automatic code generation tool is illustrated in Figure 3.3.

After the code is generated, it should be compiled and converted to the processor compatible format to be executed. Figure 3.7 shows this process, which is done using

Figure 3.3: Flow chart of the automatic code generation tool

Figure 3.4: Device tree for the Birdtag device mentioned in Chapter 2

Figure 3.5: Device tree for the implant device mentioned in Chapter 2

the open source gcc toolchain. We choose to use gcc because of its extended support for macro and interrupt handling as well as its full support for various hardware architectures including MSP430, AVR, and ARM. The pre-processing step extends all macros defined by using `#define` directive, handles conditional compilation directives such as `#if`, `#elif`, `#else`, `#endif`, handles all `#include` directives, deletes all comments, generates line number of debug symbols and warning/errors, generates file description, and handles `#pragma` directive. The compiling step generates the assembly code via lexical analysis, syntactic analysis, semantic analysis and optimization. Next step is the assembling step, which converts the assembly code to the machine instructions which in turn will generate the object file(`*.o`). In the linking step all object files(`*.o`) in the project and other libraries(`*.a` for static link and `*.so` for dynamic link) will be used to generate the executable file(`a.out`). The order and organization of the object and library files should be specified in the link script, which indicates the memory allocation for the code addresses and variables. In the last step, the executable file(`a.out`) is converted into the processor compatible binary format.

Figure 3.7: Coding flow, *arch* indicates the gcc toolchain ported on a given hardware architecture

3.4 Easy and correct debugging

An alternative to the traditional debugging tools is our on-line debugging tool, which considers the debugging PC as the external flash and helps the developers to easily modify and test the software system on the fly. It does not require the image file of the entire software system to be loaded on the internal flash of the processor. Instead, it will load only the part of the binary image that needs to be tested to the RAM. Subsequently, it enables execution of both instruction-by-instruction and group instruction as they are received from the debugging PC. The instructions are being executed from the RAM until execution involves instructions outside the loaded code. During execution of this binary image, the GUI provides data on registers and stack information. From the moment this binary image code is exited, a new binary image to be tested can be loaded. The amount of code to be loaded depends on the available RAM. The RAM can be used by both the binary image code and all global variables. The address of all global variables are stored in the debugging PC. Because of sending online instructions directly to the device MCU RAM for execution, our tool does not need to frequently erase or program the flash memory. It, therefore, saves development time and lifetime of the flash

memory. Moreover, by utilizing a graphical user interface, the on-line debugging tool makes the development of the WSNs software system transparent. In addition, by enabling simultaneous connection of several WSNs hardware platforms to the debugging PC, it facilitates group debugging. It does not require any extra hardware to implement the debugging, while current debugging heavily relies on the choice of hardware debuggers. Moreover, unlike the current debugging practices, which require different hardware debugger for different processor, our online debugging technique provides a standard way to debug all processors. Fig. 3.8a illustrates how different online debugging concept is from the current debugging practices. Our online debugging tool can support break points, register status without using Jtag programmer. Moreover, it does not need to reprogram the flash every time. It only needs to program the flash at the very beginning and at the end when the validation process of the software system is done.

Figure 3.8: Comparison of current debugging techniques and our online debugging technique

The online debugging starts by connecting the device to the debugging PC via a debugging interface. Then our specially designed firmware, called μ Boot, will be programmed into the flash memory of the device. The concept behind the μ Boot is similar to the msp430bsl [3] program from Texas Instrument (TI), which communicates with the MSP430 series processor via serial link UART to erase/write the internal Flash. Bootstrap Loader (BSL) can only be used for re-programming the device and not for the debugging. In μ Boot, complexities are moved from the target device to the debugging PC. μ Boot extends the limited functionality of BSL with enabling (i) debugging drivers, protocol stack, and software application, (ii) full-fledged re-programming via radio, and (iii) partially re-programming, via which only part of the binary image will be updated. The μ Boot is automatically generated (using the automatic code generation procedure explained previously) based on the following parameters:

- Processor type: which is extracted from the hardware schematic generated by our AHST tool;
- Clock source type: which can be external high/low frequency crystal, internal Digitally Controlled Oscillator (DCO), etc. This information is also extracted from the hardware schematic generated by our AHST tool. In case that more than one option is available, the user should select one;
- Debugging interface: which can be UART, SPI, Inter-Integrated Circuit (*IIC*), or USB. This information is provided by the user;
- Radio interface: which is extracted from the hardware schematic generated by our AHST tool and confirmed by the user.

The μ Boot cannot be modified during the debugging. It guides the device to load sequential instructions from the debugging PC. The instructions will be executed on the device MCU upon their arrival. At the same time, the internal state information of the device, including its registers' states, will be sent back to the debugging PC for analysis.

As mentioned before, all hardware components are connected to the device MCU via a certain interface. The μ Boot supports these device interfaces through the following interface operation primitive instead of the register operation:

- UART read: reads data from the UART port and sends it to the debugging machine (PC or ios devices);
- UART write: writes the data received from the UART port;
- SPI read: reads data from the SPI port and sends it to the debugging machine (PC or ios devices);
- SPI write: writes the data received from the SPI port;
- *IIC* read: reads data from the *IIC* port and sends it to the debugging machine (PC or ios devices);
- *IIC* write: writes the data received from the *IIC* port.

In the debugging process, μ Boot accepts instructions/commands from the debugging machine and stores the status/information on the MCU RAM. The μ Boot provides the hook for the hardware interrupts and therefore the debugging machine can bind and use the interrupts.

Size of the code to be loaded and debugged depends on the available RAM, which limits scope of the debugging. Additionally, since the program is executed in RAM and intertwined by the μ Boot, the timing of the debugged code and the actual execution on the device may be different. This limits the applicability of the tool to non-restrict time related codes.

3.4.1 Debug communication

The μ Boot is designed in a layered fashion. Physical layer handles the physical links such as UART, USB, Bluetooth, and the radio. The communication layer handles the packet-based communication between the debugging PC and the device MCU. The communication layer is a physical independent layer. The application layer handles the debugging.

In order to make the debugging process reliable and robust, we designed a special packet-based communication protocol. Compared to the byte-based communication, packet-based communication is more more efficient.

The packet format of our packet-based communication is shown in Figure 3.9. The packet delimiter is used to identify the beginning and the end of the packet. The packet length indicates the length of the packet, packet type defines different packets used for debugging or programming and the payload is the data of the packet, followed by the Cyclic Redundancy Check (CRC) which is used to make sure that the packet remains valid during the transmission.

Figure 3.9: eGMCP packet format

3.5 Graphical user interfaces

μ Boot supports graphical and script programming for two different platforms, i.e., PC and IOS devices. The graphical programming is supported by the μ Boot by providing the following two components:

- IDE: which is a drag and drop program, runs on the debugging PC, and includes source code editor, compiler interface, and libraries. Figure 3.10 shows the graphical user interface of the IDE.
- Debugging tool: which communicates with the target device and controls the debugging process. The debugging instructions made through the graphical user interface are sent via the debugging tool to the target device. Upon receipt of these instructions, the target device will execute them.

3.6 Partial (Re)-Programming

The μ Boot supports easy and fast re-programming because it allows partial replacement of the image file instead of replacing the entire image. The μ Boot, which is programmed at beginning of the flash memory, contains:

- System initialization function: which initializes the crystal or DCO to provide clock source;
- Flash driver: to erase and write new binary image to the device flash memory;
- Timer function library: which is the library for timer functionality and defines how the software system should use the hardware timer;
- Drivers for programming interface: such as UART, USB, and the radio;
- Network stack (if radio programming interface is used).

Re-programming using μ Boot is done by dividing the binary image into different parts in order to keep the size as small as possible. This division is done based on the software module concept. This means that each software module will be written in a specific address on the flash memory. When a specific software module needs to be re-programmed, only that part of the binary image will be updated, which helps reduce the re-programming time and effort. Table 3.1 shows examples of the code size of different device drivers of the implant hardware platform designed in Chapter 2.

(a) User requirement input

(b) Schematic design

(c) Makefile and linksript

Figure 3.10: Graphical user interface

Name	Code size (bytes)
Radio driver	2574
Battery driver	402
Temperature sensor driver (TMP102)	2734
Accelerometer and Gyroscope sensors driver (MPU6050)	1022

Table 3.1: Code size of software modules of the implant hardware platform

In our re-programming approach, the flash memory is divided into erasable units which is also the basic units for the code (re)placement. First, the software system is divided into different software modules, which in turn are written into different basic units on the flash memory. A jump table is created at the beginning of the first basic unit of each software module. The jump table stores the offset of the physical address of the symbols and functions of each software module relative to the start address of the unit. Therefore, the functions and symbols are easily locatable by other software modules even after the update and re-programming process. Using the jump table, if a specific part of the software system needs to be re-programmed and updated, only its relevant sectors/pages will be erased and re-written. This sector/page-wise approach makes the re-programming process easier and much faster. The local variables are allocated via the stack and as such are not affected by the update/re-programming process. The global variables, however, if used system wide will be greatly affected. To overcome this problem, in our approach we allocate a special sector on the flash memory to store the global variables and name and address of these variables are stored on the PC. Therefore, the updated software module can directly access the global variables using the address information from the PC.

3.7 Proof-of-concept

For evaluation purpose, we consider the following example. On the Birdtag hardware platform, whose hardware scheme was generated by our AHST in Chapter 2, there is a LED. Using the tools presented in this Chapter, we aim to perform and verify the following tasks:

1. using our automatic code generator tool to create codes to enable the LED to blink with a certain frequency according to the input from the serial port. The automatically generated system software code and the code related to the user input to enable the LED blinking with 5Hz frequency can be found in Appendix 5.2 and Appendix 5.2, respectively.

2. using our online debugger tool to debug the working condition of the serial port. The working condition of the serial port is important in this example since the input of the blinking frequency is sent from the PC via the serial port. Figure 3.12a shows that the message sent from the Birdtag to the PC is not correctly decoded. This is due to the fact that the serial port configuration is not correct. To correct this, we check the system clock as well as the clock source and the clock divider to generate the serial port baud rate using our online debugger tool. As it can be seen in Figure 3.11a the clock source and the clock divider to generate the serial port baud rate corresponding to 9600 are 416 and 6, respectively. According to $N = \frac{f_{BRCLK}}{\text{Baudrate}}$ formula, values of 208 and 3 do not generate the right baud rate. After re-calculation, correct values for the clock source and the clock divider are sent to the Birdtag. These values are shown in Figure 3.11b.

Figure 3.11: User configuration for generating serial port baud rate

3. using our re-programming tool to re-program the tag with a new system clock because the old system clock does not support the new blinking frequency. As shown in Appendix 5.2, the LED blinking is implemented using software delay, therefore the blinking frequency is determined by the system clock. At the beginning the system clock is set to 4MHz and the blink frequency is 5Hz. If we need to change the blink frequency, for example to, 20Hz without changing the delay settings, we change the system clock configuration code and reprogram it

```
user@UX32V0 ~/projects/wDesigner * picocom -b 9600 /dev/ttyUSB0
picocom v1.7

port is      : /dev/ttyUSB0
flowcontrol  : none
baudrate is  : 9600
parity is    : none
databits are : 8
escape is    : C-a
local echo is : no
noinit is    : no
noreset is   : no
nolock is    : no
send_cmd is  : sz -vv
receive_cmd is : rz -vv
imap is      :
omap is      :
emap is      : crclrf,delbs,

Terminal ready
yyyyyyyyyy
```

(a)

```
user@UX32V0 ~/projects/wDesigner * picocom -b 9600 /dev/ttyUSB0
picocom v1.7

port is      : /dev/ttyUSB0
flowcontrol  : none
baudrate is  : 9600
parity is    : none
databits are : 8
escape is    : C-a
local echo is : no
noinit is    : no
noreset is   : no
nolock is    : no
send_cmd is  : sz -vv
receive_cmd is : rz -vv
imap is      :
omap is      :
emap is      : crclrf,delbs,

Terminal ready
Birdtag System started
```

(b)

Figure 3.12: Screen shots of the output of the online debugging tool

to the MCU. The experimental results using the oscilloscope are illustrated in Figure 3.13.

Figure 3.13: The blinking frequency measured using the oscilloscope before (5Hz) and after (20Hz) the reprogramming process

3.8 Summary

In this chapter, we presented a tool to support the entire development life-cycle of the software system of the WSNs devices. One of the unique features of the tool is the tight hardware/software coupling between the hardware schematic design automatically generated by our AHSC tool (presented in Chapter 2) and the device driver and system initialization. In addition to automatic code generation, the tool enables online debugging and re-programming. Through the use of this tool, the entire development life-cycle of the software system is made easier, faster, and less error-prone. This is due to the fact that it provides pre-defined default functionalities such as interrupt handlers, watchdog timers, and communication libraries. This leaves no room for the users to make errors since the code is generated automatically by the tool. This also shields the application experts from difficulties and complexities of software development for WSNs platforms. Using a proof-of-concept, we have shown how our tool support the development life-cycle of the software system.

Part II: Cross-Layer network protocols

CHAPTER 4

Efficient network protocol design

Not only design of hardware and system software for wireless sensor networks is challenging but also design of efficient network protocols. Challenges faced by network protocol design include shared wireless communication channel, limited transmission range, limited resources in terms of power, memory, and processing, network and environment dynamicity, and cross-layer influence of various protocol stack protocols. The shared wireless communication channel often leads to collision. Since each sensor node has limited channel detection range, collision due to the hidden terminal problem may take place during the data transmission. In addition, as each sensor node has limited transmission range, the exposed terminal problem tends to appear, which decreases the channel utility. Additionally, scheduling policies such as duty cycled sleep scheduling, which are often combined with routing protocols, to minimize energy consumption create serious challenges for real-time data delivery. The medium access control and the routing should ensure reliable, timely, and energy-efficient data dissemination and delivery in WSNs applications. However depending on the application, importance and priority of these requirements may change.

In this chapter¹, we present three cross-layer MAC and routing protocols designed for specific network topologies and scale, each of which satisfying a set of network and application requirements of our target applications. Environmental monitoring and ecology applications have often large-scale static backbone infrastructures. To ensure that the network runs for very long time, a low duty cycle MAC is often

¹Guo, Peng and Meratnia, N. and Havinga, P.J.M. and Jiang, Hongbo and Zhang, Kui (2015) OPS: Opportunistic pipeline scheduling in long-strip wireless sensor networks with unreliable links. *Wireless Networks*, 21 (5). pp. 1669-1682. ISSN 1022-0038

Guo, Peng and Jiang, Tao and Zhang, Qian and Zhang, Kui (2012) Sleep Scheduling in Critical Event Monitoring with Wireless Sensor Networks. *IEEE Transactions on Parallel and Distributed Systems*, 23 (2). pp. 345-352. ISSN 1045-9219

utilized. This brings with itself the risk of either missing a packet sent by another node or delaying the message delivery and dissemination. Both of these situations can often occur in our target applications, in which an event (e.g. fire in a forest area) can occur for instance or presence of an animal should disseminated through the network as a trigger to wake up the sleeping nodes. Ensuring reliability, energy-efficiency, and fast data dissemination, a cross-layer design integrating MAC and routing is needed. To this end, contributions of this chapter include:

- an opportunistic energy-efficient cross-layer MAC and routing protocols for low duty cycle static WSNs with unreliable links and a linear topology;
- an opportunistic energy-efficient cross-layer MAC and routing protocols for low duty cycle for mobile large-scale WSNs interacting with a none time-critical static backbone infrastructure;
- a reliable and fast cross-layer MAC and routing protocol for large-scale time-critical event detection applications of WSNs;

In this chapter we introduce these protocols and their performance evaluations.

4.1 A cross-layer MAC and routing protocol for low duty cycle static WSN with unreliable links and a linear topology

In many monitoring applications of WSNs, the geometry of the physical area to be monitored is narrow, yet, quite long. Example applications include monitoring dikes and gas in coal mines [14, 24], rivers [31], and pipelines [53], in which tunnels, rivers, and pipelines are only several meters wide but tens of kilometers long. These types of geometries form a special class of WSNs called linear WSNs. Being deployed along the long monitoring area, linear WSNs usually have over tens of hop counts.

In order to ensure long-term operation of most WSNs with linear topology, sensor nodes usually operate at a very low duty cycle (e.g., 1% or less), implying that the sensor nodes sleep most of the time and stay active for only a short duration in each duty cycle. Under such circumstances, when a sender node needs to deliver a packet, it always has to wait for a certain period of time until its destination node becomes active. This results in a latency, which in a linear WSNs with large hop counts usually leads to a very long end-to-end data delivery delay. For example,

a typical sleep latency of 0.5s within each hop can lead to the end-to-end delay of over tens of seconds, which cannot be tolerated for example in case of time critical packets with short time-to-live whose information will not be meaningful after their time-to-live has passed. Additionally, because of the low duty cycle, the network can contain lots of packets that all need to be stored on the WSN devices on the route. This implies that, in particular for the large scale networks, the required buffer size can become very high or buffer overflows will occur.

To reduce the packet delivery delay and to minimize the buffer requirements, a number of delay-tolerant scheduling schemes have previously been proposed [12, 25, 41, 46, 54, 59]. However, generally speaking, the average end-to-end packet delivery delay with these schemes is approximately proportional to the product of the hop counts and duty cycle length, which is still too large for large-scale linear WSNs. In order to shorten this delay, reducing the gap between adjacent nodes' wake up time is a key issue. If the receiver node wakes up immediately after the sender node has obtained a packet, the sender node can send the packet directly to the receiver without any delay. This idea has previously been published under names such as staggered wake-up [46], streamlined wake-up [41], ladder wake-up [12], and pipeline scheduling [36]. Although using pipeline scheduling the minimum packet delivery delay can be achieved in low duty cycle linear WSNs, its performance becomes poor when being applied to unreliable communication environments. An example of this is illustrated in Figure 4.1 (a). Node A, B and C work have a duty-cycle and wake up one after another. Node A needs to deliver packets to C via B. When there is no transmission failure, the packet delivery delay is minimum 2τ , where τ is size of the time slot. However, if A fails to transmit its packet to B during its duty cycle, it has to retransmit the packet in the next duty cycle, resulting in a delay of $T + 2\tau$.

To enhance the pipeline scheduling in unreliable communication environments, authors of [36] proposed the multi-pipeline scheduling algorithm, which employs multiple parent nodes in each layer on the data delivery path to establish multiple pipeline schedules. When a transmission fails, the sender node has multiple chances to retransmit the packet in the current duty cycle instead of the next duty cycle, thus reducing the packet delivery delay due to sleep latency. However, the performance of the multi-pipeline scheduling algorithm much depends on the number of parent nodes in each layer. Authors of [56] employ data aggregation in a clustered linear multi-hop network. They achieve good results in terms of delay, energy-efficiency, however their specific scenario of clustering is not always applicable to linear WSNs.

Figure 4.1: Traditional *pipeline scheduling* on unreliable links

In [59], authors proposed an asynchronous wake-up scheduling scheme to reduce the end-to-end latency, in which each sensor node has several choices to send its packet to its neighbors that has woken up at different times in the duty cycle. Therefore, the average delay among each hop can be reduced to a fraction of the duty cycle. However, the total delay still increases proportionally to the product of the hop counts and the duty cycle length. In [54], a MAC protocol is proposed, in which the receiver nodes wake up at appropriate times according to the demand of their sender nodes. However, the packet is delivered only during the short duration of nodes' active time in each duty cycle, which restricts the hop counts of transmission demands in each duty cycle. In [12,14,41,46], authors discussed the wake-up schedule according to an established directional data traffic path. All the nodes wake up when their sender nodes have just received the data packets, and go to sleep as soon as they transmit the packets to their receiver nodes. Hence, the data packets can be delivered step by step without any waiting time and the packet delivery delay can be essentially eliminated. Authors of [25,49,60] address the use of opportunistic routing for data delivery in duty cycled linear WSNs to reduce the packet delivery delay. However, these techniques are usually based on asynchronous wake-up, and as such the average data transmission delay between each hop is still a fraction of the duty cycle, and the total delivery delay in linear WSNs with large hop counts is still very long.

4.1.1 Model and assumptions

The linear WSNs we consider are composed of M sensor nodes randomly deployed in a linear geometry shaped area. There are multiple routes in the linear WSNs for data collection. This is to prevent disconnection of the network due to the unavailability of some nodes [8]. Traffic in this linear WSNs is low and each packet (containing detected events or sampled data) can be recorded with just several bytes.

The network has a low duty cycle periodically alternating between active and dormant states. Nodes in active state can send and receive packets and sense. They change their states according to their wake up schedules. Each schedule is periodic with the same period T , i.e., the duty cycle. The active time of each sensor node is fixed in the schedule. Each node maintains a local schedule table recording the times when its neighbors are in the active state. The duty cycle T can be divided into a number of time units τ , i.e., $T = L * \tau$. Usually, $L \gg 1$. τ is approximately the round-trip transmission time (including data transmission, ACK and a guard interval). Each sensor node selects one or several consecutive time units as its wake up time slot. The size of the time slot depends on traffic load of the sensor node. An example of our duty-cycle model is shown in Figure 4.2.

Figure 4.2: The duty-cycle model

Our goal is to minimize the average end-to-end packet delivery delay in a linear WSNs. In view of the delivery latency, we are mainly concerned with the wake up time of sensor nodes' schedule. Therefore, we use T, t_i to represent the schedule of node n_i , where t_i denotes the wake up time unit of n_i .

We make following assumptions for the design of our network protocol:

- Links are unreliable and a transmission may not be successful depending on the link quality.
- Each sensor node knows its wake up schedule as well as the schedules of its neighbors. This can be implemented by allowing the sensor nodes exchange

information locally during the network initialization phase.

- Each node knows the link quality between itself and its neighbors. This can be implemented by applying existing mechanisms such as the four-bit link estimation [60] and STLE [8].
- Sensor nodes are locally synchronized so that they know when to send packets to their neighbors according to their wake up schedules. Local synchronization can be achieved using a time stamping technique, such the one described in [44], which achieves an accuracy of $2.24\mu s$ with the cost of exchanging a few bytes of packets among neighboring nodes every 15 minutes.
- The network has a linear topology, yet multiple forwarding links can be available at every node.

4.1.2 An opportunistic energy-efficient scheduling and routing for linear WSNs (OPS)

The design of our opportunistic energy-efficient scheduling and routing (OPS) protocol is based on the two following concepts:

- Sensor nodes in linear WSNs can overhear data transmissions passing by them due to their specific narrow geometry. By checking whether there is an acknowledgement (ACK) sent back from the receiver, the overhearing node can know the transmission result, and will become the new receiver to forward the packet if the transmission fails. Thus, the time wasted for waiting for the sender to re-transmit the packet in the next duty cycle can be saved. It is easy to implement the overhearing, as we can schedule the overhearing node with the same active time as that of the receiver and set the node to receive all packets no matter if it is the destination of the packet.
- If enlarging the gap between adjacent nodes' wake up time in Figure 4.1 (named as fat pipeline scheduling), sender A will have multiple chances to retransmit the packet to the receiver B in its current duty cycle before the destination node C wakes up. This prevents retransmissions in the next duty cycle. Furthermore, if node B receives the packet early in the gap before node C wakes up, it can consider to send the packet to another node closer to node C. Then, this new node can help relay the packet to node C with higher success probability when node C wakes up.

In the traditional pipeline scheduling algorithms [12, 41, 46, 49], as shown in Figure 4.3 (b), usually a data collection tree is constructed and nodes on each path of this tree wake up one by one with a gap of a certain time slot. As it can be seen from Figure 4.3 (b), there are two forwarding paths $A \rightarrow B \rightarrow C \rightarrow S$ and $D \rightarrow E \rightarrow S$. The schedules of nodes are pipelined as $T, i \rightarrow T, i+1 \rightarrow T, i+2$ and $T, i+3 \rightarrow T, i+4 \rightarrow T, i+5$, respectively. These schedules can achieve minimum data collection delay when no transmission failure happens. However, if node A fails to send its packet to node B at $T, i+1$, it has to wait for T time to retransmit the packet.

Figure 4.3: A typical example for three scheduling schemes: Network topology (a), Traditional pipeline scheme (b), Multi-pipeline scheme (c), OPS scheme (d)

In the multi-pipeline scheduling algorithm, as shown in Figure 4.3 (c), the path via which packets are sent can be switched between the two parallel pipelines on the forwarding paths, thus mitigating the influence of broken pipeline. For instance, when node A fails to transmit its packet to node B at $T, i+1$, it can transmit the packet to node D immediately at $T, i+2$, and the packet will be forwarded along the pipeline $D \rightarrow E \rightarrow S$. If the pipeline $D \rightarrow E \rightarrow S$ also fails, the packet has to be retransmitted in the next duty cycle. It can be seen that each transmission failure results in a lower path chances.

As shown in Figure 4.3 (d), in OPS we consider one main forwarding path for the data collection, where nodes A, B, C and S form the main forwarding path. Nodes outside the path, e.g., nodes D and E, send their packets only to their nearby nodes on the path. Intuitively, data collection in this way should be energy-efficient, as the traffic is low and sensor nodes on the main path can aggregate their local data. We

schedule nodes on the main forwarding path with a fat pipeline scheduling scheme which has larger gap of adjacent nodes' wake up time than the traditional pipeline scheduling. Besides the fat pipeline scheduling, we also schedule nodes outside the forwarding path with the same time slots as their nearby nodes on the path. This is to provide opportunistic links to enhance the pipeline forwarding.

To illustrate the advantages of the fat pipeline scheduling and the opportunistic links, we discuss four typical cases of packet delivery in OPS. These cases are shown in Figure 4.4, and the network topology is shown in Figure 4.3.

Figure 4.4: Design of the proposed OPS scheduling

- Case 1: In the first duty cycle, node A transmits packet p_i to node B at $T, i+2$. Suppose B successfully receives the packet. Then, node B goes to sleep immediately and wakes up again at $T, i+4$ to transmit p_i to node C who just wakes up.
- Case 2: In the second duty cycle, node A transmits packet p_{i+1} to node B still at $T, i+2$. However, this transmission fails. Then, node B will keep listening and node A can retransmit p_{i+1} to B at $T, i+3$. Suppose node B successfully receives p_{i+1} at $T, i+3$. Then, B can forward p_{i+1} to node C at $T, i+4$.
- Case 3: In the third duty cycle, node A fails to transmit packet p_{i+2} to node B at $T, i+2$. Also, node A fails to retransmit the packet at $T, i+3$. However, another node (node E), which is close to node B and is scheduled to overhear node B, successfully receives p_{i+2} at $T, i+3$. Hence, node E substitutes node B to forward p_{i+2} to node C at $T, i+4$.
- Case 4: In the fourth duty cycle, node A successfully transmits packet p_{i+3} to node B at $T, i+2$. Since node C has not yet woken up, instead of going to sleep

at $T, i + 3$, node B sends the packet to node F which is located between node B and C. Node F then will help to send p_{i+3} to node C at $T, i + 4$ with higher success probability.

According to the example, though larger gap is set in OPS, the proposed OPS has the same delay ($T, i + 4 - T, i$) as multi-pipeline scheduling in ideal transmission cases (i.e., all transmissions succeed). However, if the links are unreliable, OPS can inherently have more retransmission chances in a duty cycle than multi-pipeline scheduling, while require less retransmissions for a packet. This is because (i) the sender nodes in OPS have multiple retransmission chances in a duty cycle due to the fat pipeline scheduling, while the retransmission chances in multi-pipeline scheduling depends on the number of receivers for the sender, (ii) attributing to overhearing, every other receiver for a sender node in OPS can further double the reception chances in a duty cycle (as case 3), (iii) some other receiver nodes in OPS can further help to enhance the link quality (as case 4), thus decreasing the number of retransmissions.

4.1.2.1 Scheduling of nodes with OPS

Before explaining the scheduling procedure, we first provide the notations used as shown in Table 4.1. We specially define opportunistic forwarding nodes and opportunistic relaying nodes, which provide two kinds of opportunistic links for the forwarding path, as follows.

Γ	set of nodes on a forwarding path
$p_{i,i+1}$	link quality for link $n_i \rightarrow n_{i+1}$
n_i^f	opportunistic forwarding node for node n_i (if there is)
n_i^r	opportunistic relaying node for node n_i (if there is)
$C(n_i, n_{i-1}, n_{i-2})$	Route quality for route $n_i \rightarrow n_{i-1} \rightarrow n_{i-2}$, i.e., $p_{i,i-1} * p_{i-1,i-2}$
$N(n_i)$	set of neighboring nodes for n_i

Table 4.1: Definitions of notations

- Definition 1:** Consider a segment of the forwarding path $n_i \rightarrow n_{i+1} \rightarrow n_{i+2}$, node $node_{i+1}^f$ is an opportunistic forwarding node for node n_{i+1} , which satisfies: (1) $n_{i+1}^f \notin \Gamma$; (2) $n_{i+1}^f = \arg \max_{x \in N(n_i) \cap N(n_{i+1}) \cap N(n_{i+2})} \{C(n_i, x, n_{i+2})\}$
- Definition 2:** Consider a segment of the forwarding path $n_i \rightarrow n_{i+1} \rightarrow n_{i+2}$, node $node_{i+1}^r$ is an opportunistic relaying node for node n_{i+1} , which satisfies: (1) $n_{i+1}^r \notin \Gamma$; (2) $n_{i+1}^r = \arg \max_{x \in N(n_{i+1}) \cap N(n_{i+2})} \{C(n_{i+1}, x, n_{i+2})\}$

Figure 4.5: Opportunistic transmission links

Figure 4.5 illustrates all possible data forwarding links in the linear WSNs. Note that, not all sensor nodes on the forwarding path have their opportunistic forwarding nodes and opportunistic relaying nodes. The existence of these nodes depends on the deployment of the linear WSNs. It can be seen from Figure 4.5 that a sensor node on the forwarding path may get a packet from its sender node on the path or its sender node's opportunistic forwarding node or opportunistic relaying node depending on the random results of the wireless transmissions. In addition, the sensor node on the path may forward the packet to its receiver node on the path or its receiver's opportunistic forwarding node or opportunistic relaying node of itself.

We call the destination node of a sender as the sender's parent node and the destination node of parent node as the sender's grandparent node. During the initialization of OPS, each node on the forwarding path, e.g., n_i , will have its active time slot ($\{T, t_i\}$), its parent node n_{i-1} 's active slot ($\{T, t_{i-1}\}$), and its grandparent node n_{i-2} 's active slot ($\{T, t_{i-2}\}$). The nodes n_i^f and n_i^r , if there are, will also have $\{T, t_i\}$, $\{T, t_{i-1}\}$ and $\{T, t_{i-2}\}$.

Algorithm 2 describes the scheduling of a node n_i on the forwarding path. The node n_i on the path will periodically wakes up at its active time slot to receive the coming packet and transmits packet (if it has) to n_{i-1} at n_{i-1} 's active slot. Particularly, in n_i^f 's reception duration (i.e., n_i^f 's active slots), if n_i does not detect packet transmission, it sends a short N-ACK to *opportunistic relaying node* n_i^r (if there is) to notice the latter to go to sleep, and it also then goes to sleep. If n_i detects that there is a packet transmission in its active time but it fails to receive the packet, it will not return ACK and will keep listening to the channel for packet retransmission till n_{i-1} is going to be active. If node n_i successfully receives a packet in its active time but n_{i-1} has not been active, then n_i will send the packet to n_i^r (if there is). In n_i^r 's transmission duration (i.e., n_{i-1} 's active slots), if n_i has already successfully sent the packet to n_i^r in its current duty cycle, n_i will not transmit the packet any more (to avoid collision with n_i^r). Otherwise, n_i will keep transmitting the packet in its duty cycle until it receives an ACK from n_{i-1} .

```

1: STATE=FREE
2: while  $time \in [\{T, t_i\}, \{T, t_{i-1}\})$  do
3:   Listen to the channel
4:   if Detect a packet then
5:     STATE=BUSY
6:     if Receive the packet then
7:       Send back ACK
8:       STATE=READY
9:     end if
10:  end if
11:  if STATE=READY &  $\exists n_i^r$  then
12:    Send the packet to  $n_i^r$ 
13:    if Receive ACK from  $n_i^r$  then
14:      STATE=FREE
15:    end if
16:  end if
17:  if STATE=FREE then
18:    Send N-ACK to  $n_i^r$ 
19:    Go on sleep and BREAK
20:  end if
21:  time++
22: end while
23: while  $time \in [\{T, t_{i-1}\}, \{T, t_{i-2}\})$  do
24:   if STATE=READY then
25:     Send packet to  $n_{i-1}$ 
26:     if Receive ACK then
27:       STATE=FREE
28:     end if
29:   end if
30:   time++
31: end while

```

Algorithm 2: Scheduling for node n_i on forwarding path

Algorithm 2 describes scheduling of *opportunistic forwarding* for node n_i^f to node n_i . When node n_i^f is assigned with active time slots of n_i and n_{i-1} , it will periodically wake up at n_i 's active time slot to overhear the air-in packet. If it receives the packet but does not receive the ACK sent by n_i , it will send ACK to n_i in the guard interval (indicated as ϵ in *Algorithm 2*) in $\{T, t_{i-1}\}$ (to avoid collision with n_i later), and then will send the packet to nodes n_{i-1} at n_{i-1} during their active slot.

```

1: STATE=FREE
2: while time  $\in [\{T, t_i\}, \{T, t_{i-1}\} - \epsilon]$  do
3:   Listen to the channel
4:   if Detect a packet then
5:     STATE=BUSY
6:     if Receive the packet then
7:       STATE=READY
8:     end if
9:   end if
10:  if Receive ACK then
11:    STATE=FREE
12:    Go to sleep and BREAK
13:  end if
14:  if STATE=FREE then
15:    Go on sleep and BREAK
16:  end if
17:  time++
18: end while
19: while time  $\in [\{T, t_{i-1}\} - \epsilon, \{T, t_{i-2}\}]$  do
20:  if STATE=READY then
21:    Send ACK to  $n_i$ 
22:    Send packet to  $n_{i-1}$ 
23:    if Receive ACK then
24:      STATE=FREE
25:    end if
26:  end if
27:  time++
28: end while

```

Algorithm 3: Scheduling for node n_i^f

Algorithm 3 describes scheduling of *opportunistic relaying* for node n_i^r to node n_i . When node n_i^r is assigned with active time slots of n_i and n_{i-1} , it will periodically wake up at n_i 's active time slot to receive the possible packet. If it receives N-ACK from node n_i , which indicates there is no packet transmission in the current duty cycle, n_i^r will go to sleep immediately to save the idle listening time. If node n_i^r receives packet from n_i , it sends back ACK to node n_i immediately and transmits the packet

to node n_{i-1} at its active slot.

```

1: STATE=FREE
2: while  $time \in [\{T, t_i\}, \{T, t_{i-1}\})$  do
3:   Listen to the channel
4:   if Receive N-ACK from  $n_i$  then
5:     Go to sleep and BREAK
6:   end if
7:   if Receive a packet from  $n_i$  then
8:     STATE=BUSY
9:     Send ACK to  $n_i$ 
10:    Go to sleep and BREAK
11:   end if
12:    $time++$ 
13: end while
14: while  $time \in [\{T, t_{i-1}\}, \{T, t_{i-2}\})$  do
15:   if STATE=BUSY then
16:     Send packet to  $n_{i-1}$ 
17:     if Receive ACK then
18:       STATE=FREE
19:     end if
20:   end if
21:    $time++$ 
22: end while

```

Algorithm 4: Scheduling for node n_i^r

4.1.2.2 Optimization of OPS

To minimize the end-to-end packet delivery delay of OPS, we need to find the optimal forwarding path and set the optimal gap for the scheduling. We formulate this problem as follows:

Suppose $\Gamma = \{D, n_1, n_2, \dots, n_m\}$ is a forwarding path in $G = (V, E)$, where $n_i \in V$ ($1 \leq i \leq m$) and $(n_i, n_{i+1}) \in E$. D is the destination node (or the sink node), n_m is a neighbor of the node in G furthest to D (or the node itself), $k_i\tau$ is the gap between wake up time of nodes n_i and n_{i-1} and k_i is an integer. The expected transmission delay between nodes n_{i+1} and n_i is:

$$\begin{aligned}
 D_{k_i}(n_{i+1}, n_i) &= \sum_{j=0}^{\infty} [1 - (1 - p_{i+1,i})^{k_i}] (1 - p_{i+1,i})^{j*k_i} (jT + k_i\tau) \\
 &= [1 - (1 - p_{i+1,i})^{k_i}] T \sum_{j=0}^{\infty} j (1 - p_{i+1,i})^{j*k_i} + k_i\tau
 \end{aligned} \tag{4.1}$$

Hence, the expected end-to-end delivery delay on the Γ path is:

$$D(\Gamma) = \sum_{i=1}^m D_{k_i}(n_{i+1}, n_i) \quad (4.2)$$

The optimal forwarding path is defined as $\Gamma^* = \arg \min_{\Gamma} D(\Gamma)$. Since k_i takes value independently, Γ^* can be computed as follows:

$$\begin{aligned} \Gamma^* &= \arg \min_{\Gamma} D(\Gamma) = \arg \min_{\Gamma=\{n_i\}} \sum_{i=1}^m D_{k_i}(n_{i+1}, n_i) \\ &= \arg \min_{\Gamma=\{n_i\}} \sum_{i=1}^m \min_{k_i \in Z} D_{k_i}(n_{i+1}, n_i) \end{aligned} \quad (4.3)$$

Hence, finding Γ^* in G is equivalent to finding the shortest path between node n_m and node D , when taking the weight of each link $n_{j+1} \rightarrow n_j$ as $\min_{k_j \in Z} D_{k_j}(n_{j+1}, n_j)$.

$\min_{k_j \in Z} D_{k_j}(n_{j+1}, n_j)$ can be computed as follows. One should note that $(1 - p_{i+1,i})^{k_i} = p_i$. According to Equation 4.1, we have:

$$\begin{aligned} D_{k_i}(n_{i+1}, n_i) &= (1 - p_i)T \sum_{j=0}^{\infty} j p_i^j + k_i \tau \\ &= (1 - p_i)T \left\{ \lim_{m \rightarrow \infty} \left[\lim_{n \rightarrow \infty} (p_i + p_i^2 + \dots + p_i^n) \right. \right. \\ &\quad \left. \left. + \lim_{n \rightarrow \infty} (p_i^2 + p_i^3 + \dots + p_i^n) + \dots + \lim_{n \rightarrow \infty} (p_i^m + p_i^{m+1} + \dots + p_i^n) \right] \right\} + k_i \tau \\ &= (1 - p_i)T \lim_{m \rightarrow \infty} \left(\frac{p_i}{1 - p_i} + \frac{p_i^2}{1 - p_i} + \dots + \frac{p_i^m}{1 - p_i} \right) + k_i \tau \\ &= \frac{p_i T}{1 - p_i} + k_i \tau = \frac{(1 - p_{i+1,i})^{k_i} T}{1 - (1 - p_{i+1,i})^{k_i}} + k_i \tau \end{aligned} \quad (4.4)$$

It is easy to prove that $D_{k_i}(n_{i+1}, n_i)''|_{k_i} > 0$. Therefore, $D_{k_i}(n_{i+1}, n_i)$ has the minimum value. By calculating

$$D_{k_i}(n_{i+1}, n_i)' = \frac{T \ln(1 - p_{i+1,i})}{(1 - (1 - p_{i+1,i})^{k_i^*})^2} (1 - p_{i+1,i})^{k_i^*} + \tau = 0 \quad (4.5)$$

we will obtain

Figure 4.6: Comparison on the expected delay

$$k_i^* = \frac{\ln(1 - \frac{T \ln(1-p_{i+1,i})}{2\tau}) - \sqrt{(1 - \frac{T \ln(1-p_{i+1,i})}{2\tau})^2 - 1}}{\ln(1 - p_{i+1,i})} \quad (4.6)$$

Hence, the optimal gap between the wake up time of node n_i and node n_{i-1} is $\lfloor k_i^* \rfloor \tau$, and:

$$\min_{k_j \in \mathbb{Z}} D_{k_j}(n_{j+1}, n_j) = \frac{(1 - p_{i+1,i})^{\lfloor k_i^* \rfloor}}{1 - (1 - p_{i+1,i})^{\lfloor k_i^* \rfloor}} T + \lfloor k_i^* \rfloor \tau \quad (4.7)$$

After obtaining $\min_{k_j \in \mathbb{Z}} D_{k_j}(n_{j+1}, n_j)$, i.e., the weight of link $n_{j+1} \rightarrow n_j$, it is easy to

find the optimal forwarding path Γ^* in G , by leveraging a shortest path searching algorithm. Then the opportunistic forwarding node and the opportunistic relaying node for each node on the Γ^* path will be found according to **Definition 1** and **2**.

In order to theoretically show the performance improvement brought by *opportunistic forwarding* in OPS, we take the one-hop delivery instance, and compare the expectation delivery delay of OPS with that of the *multi-pipeline scheduling* and the *single pipeline scheduling*. To simplify the evaluation, we suppose the link quality of all links are the same with value p .

Referring to Equation 4.4, the expectation delay of *single pipeline scheduling* and *multi-pipeline scheduling* are $\frac{1-p}{p} * T + \tau$ and $\frac{(1-p)^m}{1-(1-p)^m} * T + m\tau$, respectively. However, for OPS, with the *opportunistic forwarding* node doubling retransmission chances, the expectation delay is $\frac{(1-p)^{2k}}{1-(1-p)^{2k}} * T + k\tau$, where $k\tau = \lfloor \frac{\ln(1-\frac{T}{\tau}*\ln(1-p))-\sqrt{(1-\frac{T}{\tau}*\ln(1-p))^2-1}}{2\ln(1-p)} \rfloor \tau$. Set $T = 1s$, $\tau = 5ms$, the expectation delay of the three schemes can be illustrated with Figure 4.6.

From Figure 4.6, the expectation delay of the *single pipeline scheduling* (denoted as *single pipeline* in the figure) and the *multi-pipeline scheduling* increases when link quality goes down. However, the expectation delay of OPS always keeps very small even when the link quality decreases to 0.6. In addition, when increasing the number of parent nodes m in the *multi-pipeline scheduling*, the expectation delay will decrease as expected. However, when $m = 6$, the delay is even larger than that when $m = 4$ in the condition of $p > 0.68$, which means more parent nodes are not always a good thing.

4.1.3 Performance evaluation using simulation

4.1.3.1 Simulation setup

To evaluate the OPS method, we use a randomly-generated linear network in an area with a width of no more than $15m$. In the simulations, the network size varies between 100 nodes and 1000 nodes. The links between these nodes are considered to be unreliable following a wireless path loss channels combined with shadowing effects in [67]. We simulated various transmission ranges ranging from a minimum transmission range of $11m$ until a maximum transmission range of $32m$, respectively.

For the data collection, we assume that the size of packets initiated by nodes is small and that the local data can be aggregated into one packet. Hence, we measure the data collection delay as the time elapsed from a packet being sent out by the sensor node furthest to the sink until the packet reaches the sink, i.e., the end-to-end packet

delivery delay. Due to imperfection of the links, the packet delivery delay exhibits some inherent randomness and can be considered as a random variable. Here we propose to use the average delivery delay as a measurement of network performance. Each simulation result is based on 10 network topologies. For each topology, each node sends 100 packets to the sink with a low rate of one packet per 15s.

In addition, for comparison, we measure the energy consumption by the total number of transmissions for a single packet sent by the node furthest to the sink.

We compare the performance of OPS with *single pipeline scheduling* [12, 36, 41] and *multi-pipeline scheduling* [13]. For a fair comparison with the *single pipeline scheduling*, we improve the method with assumptions that a shortest path with the highest quality has been established in the linear WSNs and all nodes outside the path are scheduled at other time slots to send packets to the nodes on the path without collisions. The *multi-pipeline scheduling* makes use of multiple parent nodes in the network scheduled with multiple pipeline schedules.

4.1.3.2 Simulation results

- Effect of different duty cycles

We first evaluate the performance in networks with different duty cycle ratio (τ/T). In this simulation, 100 nodes are randomly deployed in a $15m * 300m$ field.

Figure 4.7 (a) and (b) show the average delivery delay and energy consumption of OPS method, *single pipeline* and *multi-pipeline* method (one time units is 6ms). Figure 4.7 (a) shows that the average delivery delay of the *multi-pipeline* method is only 50% of that of *single pipeline* when the duty cycle is low. It can be seen that multiple transmission chances can effectively enhance the performance of the pipeline forwarding. Moreover, the average delivery delay of OPS is far less than that of the *multi-pipeline* method. This is because OPS provides more transmission links than the *multi-pipeline* method. Meanwhile, OPS sets larger gaps between adjacent nodes with multiple slots in which multiple transmission chances are allowed. While number of transmission chances provided by the *multi-pipeline* method is restricted by the number of parents in each layer. On the other hand, OPS also utilizes the potential parents in each layer in the network to opportunistically help retransmit packet. This will effectively reduce the failure cases in the duty cycle.

As shown in Figure 4.7 (b), the number of packet transmissions with the *single pipeline* method and that with the *multi-pipeline* method are almost the same. This is because the number of transmissions with these methods is mainly determined by the quality of the independent links. In the OPS method, the action of opportunistic links is equivalent to enhance the link quality on the forwarding path. Therefore, some of the re-transmissions can be avoided. Hence, the number of transmission in OPS method is a bit lower than that in the *single pipeline* method and the *multi-pipeline* method.

- Effect of different network sizes and densities

We also evaluate performance of OPS using different network settings as shown in Figure 4.7 and Figure 4.8. For different network size from 60 to 140, the length of the area changes from 180m to 420m to keep a similar density. For different network densities, the network size is fixed at 100 while the length of the area changes from 200m to 450m.

Figure 4.7: Comparison of three schemes with different network sizes

As illustrated in Figure 4.7, the average delivery delay and number of packet transmissions increase with the network size. OPS outperforms the *single pipeline* method and the *multi-pipeline* method. Most of the time, the average delivery delay of OPS is less than 20% of that of the other two methods. For different network densities as shown in Figure 4.8, the average delivery delay increases with the length of area. This is because larger length of area leads to longer distance between neighboring

Figure 4.8: Comparison of three schemes with different network densities

nodes, thus resulting in worse link quality. It can be noted that when setting the length of the area to 200m, the average delivery delay of the *multi-pipeline* method becomes close to that of OPS. This is because the density in this case is high enough so that the *multi-pipeline* method will have enough parents in each layer to ensure more successful transmissions in the duty cycle.

4.1.4 Performance evaluation using experiments

To show the performance of OPS in a real communication environment, we deployed 20 TelosB nodes in the corridor of our office building, as shown in Figure 4.9 and Figure 4.11. We set the minimum transmission power of each node to -25dBm . Hence, the maximum transmission range of sensor nodes is about 1m, when putting the nodes on the ground. In the deployment, most sensor nodes have two parents in each layer, except for three nodes. We utilize this deployment to compare the performance of OPS with the *multi-pipeline*. The dashed arrows of OPS method in Figure 4.11 are the opportunistic links provided by the *opportunistic forwarding nodes*.

We set the wake up schedules for the nodes according to OPS and the *multi-pipeline* method. Figure 4.11 shows the time slot at which the sensor nodes periodically wake up in each duty cycle. It can be seen that within a segment of the network where there is only one node to forward packet, the *multi-pipeline* method reduces to be *single pipeline*. For the sake of distinct comparison, we set the destination node D in OPS with the same wake up slot as that of the node D in *multi-pipeline* method, and also reduce OPS to be *single pipeline* within the narrow segments so as to heighten the

Figure 4.9: Deployment of 20 TelosB nodes

Figure 4.10: Average delay

Figure 4.11: Topology of the deployment

contrast effect. The size of the time slot is $\tau = 6ms$. The length of the duty cycle is set to 1s, 2s, 3s, 4s, 5s, respectively. The node S delivers 50 packets to node D with an interval 10s. Node D records the delivery delay of each packet received.

Figure 4.10 shows the results of the average delivery delay with the two methods, i.e., OPS and the *multi-pipeline* method. It can be seen that OPS usually outperforms the *multi-pipeline* method, with the average delivery delay no more than 60% of that with the *multi-pipeline* method. When the duty cycle increases, the predominance becomes more distinct. It can be seen that each node in Figure 4.11 with the *multi-pipeline* method has no more than two chances in each duty cycle to deliver the packet, while a node on the forwarding path in OPS may have two chances to deliver to packet and one additional chance to opportunistically forward the packet.

4.2 A cross-layer MAC and routing protocol for interacting mobile and static rendezvous

Some WSNs applications have a hybrid architecture. This means that a ground static network, which forms the backbone, needs to interact with a network of mobile nodes. Animal monitoring application is an example of this type of networks. The mobile nodes may or may not form a network among themselves. They, however, need to communicate with the backbone to report their data.

This can be for instance because the mobile nodes need to have a rendezvous with the backbone. With static networks (like the backbone) a schedule can be made, however with mobile nodes that is not really feasible. Keeping the nodes active all the time is also not feasible due to lack of energy. Moreover, the range of the mobile node (which can be an implant in an animal) is often smaller than the backbone. This implies that the rendezvous between the mobile node and the static node is crucial, since otherwise the data may be lost (or not transmitted). Therefore using the backbone, data of the mobile nodes will be transferred to a central location. In addition to the data transfer, the backbone infrastructure is often also responsible for detection and tracking of animal movements. The data traffic is therefore random, as it depends on the presence and movement of the mobile nodes, which by itself is quite random. The backbone is duty-cycled and periodically listens to data being sent by the mobile nodes. When mobile nodes pass through the backbone infrastructure, the backbone nodes detect their presence. The presence information can further be broadcasted among the backbone nodes to enable tracking of the mobile nodes. Sink node is a backbone node, often equipped with an Internet connection for example

Figure 4.12: Mobile node interaction with a static backbone infrastructure

using GPRS. This scenario is shown in Figure 4.12.

Generally speaking, duty cycle-based data gathering is straight forward. Each node only needs to set its active time in the duty cycle according to the active time of its children nodes. Use of pipeline sleep scheduling methods described in the previous section to set this active time can result in minimizing packet delivering delay. Pipeline sleep scheduling for gathering random data in WSNs, however, is not straight forward. This is because, in addition to relay data of their children nodes to their own parent at their active time, fixed nodes need to specially set their active time to access random data coming from the nearby mobile nodes. Additionally, fixed nodes may also need to set active time to forward some command messages from the sink to configure parameters in the sleep scheduling. Hence, in one duty cycle, each fixed node may have three kinds of active times: (i) uplink time to relay data to the sink, (ii) access time to access data from the nearby mobile nodes, and (iii) downlink time to forward data received from the sink. The data traffic during these three active times are either uplink (relay data to the sink and access data from the nearby mobile nodes) or downlink (forward data received from the sink).

4.2.1 Applicability of pipeline scheduling for random data gathering

To design a pipeline based scheduling for random data gathering, the following issues need to be considered:

- unreliable links while setting the optimal gap between adjacent nodes on the data delivery path to achieve the minimum expected delay;

Figure 4.13: Active time of nodes

- wireless interference to appropriately set for each node the time when its active time starts in each duty cycle so that the collision problem can be avoided.
- parents' active time to receive multiple children' data during active time of the fixed nodes. This means that the parent active time should be large enough to receive multiple children data, as shown in Figure 4.13.
- double pipeline based scheduling for the fixed nodes to relay data to the sink node and to forward data from the sink node in a minimum end-to-end delivery delay;
- dynamicity of mobile nodes to ensure successful data transmission between mobile and fixed nodes during active time of the fixed nodes.

We expect that the fixed nodes' uplink time and downlink time follows a double pipeline scheduling scheme to achieve a minimum end-to-end packet delivering delay. Each fixed node's uplink time and downlink time will be set according to the node's hop count to the sink. To avoid any possible collision in case one frequency channel is used, fixed nodes within the same layer will be assigned different time slots. Additionally, to be able to receive data from the mobile nodes, active time of fixed nodes should be the same as active time of the mobile nodes. Setting this active time should be done in a fast and energy-efficient manner, since mobile nodes have limited power source and may move fast.

4.2.2 An energy-efficient and fast network protocol (MobiBone)

We assume that all fixed nodes are using duty cycle scheduling and the length of the duty cycle is denoted by L , which is the same for all the fixed nodes. A shortest path is established in the network. Both uplink traffic and downlink traffic go along the shortest path to or from the sink. Each fixed node is synchronized with its parent node on the shortest path. Furthermore, each fixed node has a time slot ID. According to this ID, the fixed node determines its downlink time and uplink time. All fixed

nodes set their access time to $mL + \Delta$, where $m = 1, 2, \dots$ and $0 < \Delta < L$. The length of the downlink time is short and constant, while the length of the uplink and access times are dynamic and will be set according to the size of uplink traffic and local access traffic, respectively. This allows the size of the access time and the uplink time to be adaptive. Figure 4.14 illustrates more details on the three kinds of active time, by taking an example of several fixed nodes on the same data delivery path.

Figure 4.14: Sleep scheduling scheme for gathering random data

Start of the access time is set to $mL + \Delta$. Start of the downlink time is set to $mL + ID * S_d$, where S_d is size of the downlink time. For simplicity, we set S_d as the value of the optimal pipeline gap of the worst downlink in the network. Start of the uplink time is set to $(m + 1)L - ID * S_u(m)$, where $S_u(m)$ is size of the uplink time in the m th duty cycle and can be adjusted by the command sent from the sink node.

Each mobile node wakes up during the downlink time of all fixed nodes. Thus, the mobile node can receive the configuration command and the nearby fixed nodes' ID that are broadcasted in the downlink time by the sink. In this way, mobile nodes can get the current size of access time of their nearby fixed nodes. When the access time of a nearby fixed node starts, the mobile node becomes active and will randomly access the channel during the access time.

Our MobiBone protocol consists of the following three procedures:

- collision-free time slot ID assignment: to avoid collision during the downlink and uplink time, we propose a greedy algorithm (Algorithm 5) to assign collision-free time slots to all fixed nodes. Given $G = (V, E)$ and sink node n_0 , we will have:

According to Algorithm 5, we first assign slot ID 0 to the sink node n_0 . After that, the algorithm checks each node n_j in each layer (i.e., same hop counts to the sink node) to judge whether it has unassigned neighbors in common with any other node n_t in current layer that has the same slot ID. If there is no such a node, then node n_j can be assigned with the same slot ID of n_t . Otherwise, a

```

1: Input:  $G=(V,E)$  and  $n_0$ 
2: Construct BFS tree, and assign slot ID  $k \leftarrow 0$  to  $n_0$ 
3:  $k++$ 
4:  $C \leftarrow \emptyset$ 
5: Divide  $V - n_0$  into a group of sets  $L_1, L_2, \dots, L_H$ , where  $L_i$  is the set of nodes with  $i$  hop distance to  $n_0$ 
6: for  $i \leftarrow 0$  to  $H$  do
7:    $C_i \leftarrow \emptyset$ 
8:   for Each node  $n_j \in L_i$  do
9:      $C_i \leftarrow C_i \cup n_j$ 
10:    if  $\exists n_t \in C_i$  and  $N(n_t) \cap N(n_j) \cap (V - C) = \Phi$ , where  $N(n_t)$  is the set of node  $n_t$ 's neighbors then
11:      Assign slot ID  $k$  to  $n_j$ 
12:    else
13:      Assign slot ID  $k++$  to  $n_j$ 
14:    end if
15:  end for
16:   $i++$ 
17:   $C \leftarrow C \cup C_i$ 
18: end for

```

Algorithm 5: Collision-free time slot assignment algorithm

new slot ID will be assigned to node n_j . Nodes in layer L_H do not need to be assigned IDs, as they just take the uplink time of their parents as their time to send the data and downlink time of their parents as their time to receive data.

- flexible access time setting: due to the dynamic density of mobile nodes near fixed nodes, the access time may highly vary. To adjust the access time of the fixed nodes based on this density, we propose a frame-based aloha Algorithm 6 for mobile objects to randomly access the fixed nodes' channel.

According to Algorithm 6, the fixed nodes will extend their access time when they do not complete accessing the mobile objects' data within their accessing time. However, if the accumulated accessing time exceeds the upper bound $L - ID_{max} * S_u(m) - \Delta$, the access time will not be extended anymore as the fixed node with the maximum slot ID starts its uplink time now. In the next duty cycle, the updated access time will be shortened with β parameter to be adaptive to be dynamic accessing data.

- flexible uplink time setting: as the uplink traffic on the shortest paths may vary due to the movement of mobile objects, it is needed to adjust the size of the uplink time according to the uplink traffic. To this end, we propose a flexible

- 1: Fixed node broadcasts $S_a = S_{a0}$ in its downlink time, where S_a is the initial size of accessing time in current duty cycle.
- 2: $counter_A = 0$.
- 3: Fixed node checks the channel activities within its accessing time S_a , and records the size of time when the channel is busy as S_{aB} , records the size of time when succeeding to receive accessing data is S_{aC} , till accessing time ends.
- 4: **if** $S_{aB} - S_{aC} = 0$ **then**
- 5: Go to step 14.
- 6: **else if** $\Delta + counter_A + S_a - S_{aC} + \alpha(S_{aB} - S_{aC}) > L - ID_{max} * S_u(m), (\alpha > 1)$ **then**
- 7: Broadcasts indication that the access time ends.
- 8: Go to step 14.
- 9: **else**
- 10: Broadcast the extension of accessing time for $S_a = S_a - S_{aC} + \alpha(S_{aB} - S_{aC})$
- 11: $counter_A = counter_A + S_a$
- 12: Go to step 3.
- 13: **end if**
- 14: $S_a = S_{a0} + \beta * counter_A, (0 < \beta < 1)$
- 15: Break in current duty cycle

Algorithm 6: Flexible accessing time setting algorithm

pipeline scheduling scheme as follows:

Figure 4.15: Flexible pipeline scheduling scheme

As it can be seen from Figure 4.15, our flexible pipeline scheduling scheme consists of three components, i.e., (i) traffic learning, (ii) flexible time slot setting, and (iii) flexible duty cycle setting. Since mobile nodes move around, the traffic on a data delivery path to the sink may highly vary. High traffic requires more uplink time. However, the rate of the change may also highly vary. This introduces a challenge for accurate adaptation as shown in Figure 4.16. From Figure 4.16, a constant adaptation of uplink time cannot accurately follow change of data traffic. To deal with this situation, we employ traffic learning on the fixed nodes. Although being random in terms of when they occur, mobile nodes (in this case animals) have repetitive movement patterns when they move.

Therefore, it is possible to predict the traffic on the data delivery path at different times. For example, the animals being monitored may regularly walk across some routes to look for water/food or go back home, which may lead to high but predictable change of traffic on some paths to the sink. This predictable change helps set appropriate adaptations on the size of the uplink time. For example, when there is going to be high change of uplink traffic, the sink will select larger step of adaptation of size of the uplink time for the fixed nodes. In our proposed scheme, the fixed nodes count the uplink traffic continuously and calculate the statistics of uplink traffic at different times. After some days, the system can have reliable network traffic statistics.

Figure 4.16: Highly dynamic uplink traffic and adaptation of uplink time

This traffic learning reduces the gap between the optimal uplink time required by real traffic and the actual uplink time set by the fixed nodes. However, as the gap cannot be reduced to zero, it is still needed to further adjust the uplink time according to the real traffic. If a parent fixed node finds that the ratio of channel busy time within its uplink time is larger than a given threshold Γ_u , it will request the sink to extend its uplink time to $S_u(m+1) = S_u(m) + \partial$. If the parent fixed node finds that the ratio of channel busy time within its uplink time is smaller than threshold Γ_l , it will request the sink to shorten its uplink time to $S_u(m+1) = S_u(m) - \partial$. Figure 4.17 illustrates this adjustment. The fixed parent node adjusts its uplink time only after the sink broadcasts the command in the downlink time in the next duty cycle.

When information about $S_u(m, i), 1 \leq i \leq H$ is broadcast in the downlink time, each fixed node can find the start of its uplink time and the uplink time of its

(a) Shrink the uplink time if traffic is lighter than expected

(b) Extend the uplink time if traffic is heavier than expected

Figure 4.17: Adjustment of the uplink time

parent using $f[ID, S_u(m, i)]$. These start times can be computed as follows:

$$f[ID, S_u(m, i)] = \sum_{j=1}^i K_j * S_u(m, j) \quad (4.8)$$

where K_j is the number of IDs with which the fixed nodes are in layer j .

It should be noted that when a parent node receives data from all its children early in its uplink time, it can go to sleep in advance. This results in saving energy. This is the advantage of our adaptive scheduling.

In addition, if based on traffic learning algorithm, a fixed node finds out that there is no traffic in the coming duty cycle, it does not need to wake up. This also results in further energy saving. However, wrong prediction of the traffic learning algorithm will suspend the uplink traffic and bring unexpected traffic load in the next duty cycle of the fixed nodes. This makes the uplink time in the next duty cycle insufficient. Whenever the uplink time is insufficient for the uplink traffic, part of the packets will have to be discarded, which influences the data quality.

4.2.3 Performance evaluation using simulation

4.2.3.1 Simulation setup

For the simulation, 169 fixed nodes were deployed in a grid over an area of $12000m \times 12000m$, as shown in Figure 4.18. This means that there are 13 layers (similar hop counts from all fixed nodes to the sink) in the network. The size of each grid cell is $1000m \times 1000m$. Transmission range of fixed nodes is $1000m$ and their radio works at $433MHz$ frequency. 200 mobile nodes were also simulated. Mobile nodes move randomly $2m$ per second. Communication range between mobile and fixed nodes is $100m$. The minimum access time of each fixed nodes is set to $L \times 1\%$. If the number of mobile nodes which communicate with a fixed node is less than C_{min} (in the simulation we set to 15), then the periodic length of the fixed node's duty cycle is doubled. If the number of mobile nodes which communicate with an fixed node is more than C_{max} (in the simulation we set to 25), then the periodic length of the fixed node's duty cycle is halved. Simulation is repeated for 6000 runs.

Figure 4.18: Simulation setup

Following the three algorithms presented above Algorithm 4, Algorithm 5 and Algorithm 6, we construct a shortest path tree from each fixed node to the sink.

Quality of each link on this tree is set to $p = 0.9$, i.e., the probability that a packet is successfully transmitted on a link. Each packet can contain data from no more than 10 mobile nodes. This data includes location and motion data of the mobile node.

Mobile nodes moves around randomly, i.e., move with constant speed v but randomly change the direction every one second. Fixed nodes and mobile nodes employ MobiBone sleep scheduling. The duty cycle is set to L seconds. As mentioned before, to guarantee sufficient space for the downlink time and the access time, the earliest start of the uplink time should not be earlier than $mL + (M + 1)\Delta$. We set $(M + 1)\Delta$ to $200ms$ and ∂_i to $5ms$. We assume that one packet transmission takes $5ms$. As there exist 13 layers, the earliest start of the uplink time is expressed as:

$$\begin{aligned} (m + 1)L - f[ID, S_u(m, 13)] &= (m + 1)L - \sum_{j=1}^{13} K_j * S_u(m, j) \\ &= (m + 1)L - 2 \sum_{j=1}^{12} S_u(m, j) + S_u(m, 13) \end{aligned} \quad (4.9)$$

Therefore, when executing the flexible uplink time setting algorithm, $(m + 1)L - 2 \sum_{j=1}^{12} S_u(m, j) + S_u(m, 13)$ should be larger than $mL + 200ms$. In the initialization, we set $m = 0$ and $S_u(0, j)$ to 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10ms.

We use number of detected mobile nodes and energy consumption of the fixed nodes as performance metric. For the energy consumption, we evaluate average consumption for all the fixed nodes as well as the minimum and the maximum of the whole network.

4.2.3.2 Simulation results

Figure 4.19 shows the number of mobile nodes detected by the fixed nodes using the adaptive scheduling. As it can be seen, number of mobile nodes detected by the fixed sleep scheduling methods is low and remains almost the same. However, although number of mobile nodes being detected by the adaptive scheduling is relatively low at the beginning, it increases over time. This is due to the adaptability feature of the protocol. It can also be observed that in fixed duty cycle protocols, the higher the duty cycle the lower the undetected mobile nodes. This higher detection, however, comes at the cost of energy consumption, as shown in Figure 4.20.

The trade off between number of detected mobile nodes and the energy consumption is presented in Table 4.2. These results clearly show that an adaptive protocol

Figure 4.19: Number of mobile nodes detected by fixed nodes

Figure 4.20: Average, minimum, and maximum power consumption of the fixed nodes

and approach is beneficial for achieving a good trade off.

Table 4.2: Energy consumption per mobile detection

Scheduling Type	Time used for each detection(s)	Energy used for each detection(mJ)
1%	0.23	15.94
2%	0.5	34.65
4%	0.27	18.71
8%	0.84	58.21
16%	1.17	81.08
32%	0.98	67.91
Adaptive	0.8	55.44

4.3 A reliable and fast cross-layer MAC and routing protocol for large-scale time-critical event detection applications of WSNs

In time-critical event detection applications of WSNs, nodes are deployed in wide range of areas, where a large number of sensor nodes detect and report information of interest/urgency to the end-users. Usually, the sensor nodes are expected to work for a long period of time without recharging their batteries. Therefore, nodes are often put to sleep to prolong their lifetime. Upon occurrence and detection of a critical event, an alarm and/or event data should be broadcasted to all other nodes as soon as possible. However, utilizing any sleep scheduling, sensor nodes always have to wait for the receiver nodes to wake up before transmitting packets to them, which results in transmission delay between each pair of adjacent nodes in the network. Specially, in large-scale networks, the transmission delay may become very serious or even cause the critical event invalid. Hence, the sleep scheduling mechanism for time-critical event detection application should guarantee a low broadcasting delay.

Currently, there exist many sleep schedules for event or environment monitoring [9, 10, 20, 58], most of which aim to save energy consumption according to the characteristics of data traffic in their applications. In time-critical event detection applications, usually small number of packets need to be transmitted. An exception is the alarm, which should be broadcasted to the entire network as soon as possible. That is to say that the low delay of alarm broadcasting is of most importance in time-critical event detection application. To this end, one should find the best solution

for when sensor nodes should wake up in the duty cycle so that the waiting time for the destination nodes to wake up is minimized.

Generally, most sleep scheduling schemes have the following two common features, i.e., (i) the sensor nodes sleep and wake up alternately in a duty cycle way, and (ii) sensor nodes have to wake up at least once in each duty cycle to check whether there is any packet sent to them. Obviously, the minimal transmission delay between two adjacent nodes can be achieved only if a source node wakes and receives the alarm, the destination node wakes up right away and the source node can send the alarm immediately without any waiting. This idea was first described in [42], where a periodic receive-transmit-sleep cycle with *pipeline schedule* was designed. Figure 4.21 shows principles of this technique. It can be seen from the figure that *pipeline schedule* can achieve the minimum transmission delay even in a multi-hop WSN, where the delay in each hop is just the length of time slot τ . Therefore, it could be a promising way to design the sleep scheduling for large-scale WSNs for time-critical event detection applications.

Figure 4.21: *pipeline schedule* [42]

However, to apply *pipeline schedule* idea into design of sleep scheduling for broadcasting alarms in time-critical event detection applications, there are some challenges. Firstly, the order of the nodes' wake up times should conform to the traffic direction. If there is a traffic flow in the reverse direction, the delay in each hop will be as large as the length of the whole duty cycle, as shown in Figure 4.21. Secondly, when applying it to broadcast a packet in a network with arbitrary topology, it will cause serious collisions, and the collision can not be avoided even if packets are retransmitted, as shown in Figure 4.22. Finally, the position where critical event occur is not known a priori. Moreover, transmission failure in current time slot due to unreliable wireless links may cause retransmission in the next duty cycle. This in turn results in a large delay in the entire duty cycle.

Some works have applied the *pipeline schedule* in arbitrary WSNs [36, 43, 59].

Figure 4.22: Collision due to pipeline schedule in broadcasting

However, they just consider unicast data dissemination and as such cannot be applied to the traffic of broadcasting due to potential collision. Authors of [59] studied delay-efficient sleep scheduling methods not based on *pipeline schedule*. These techniques just make use of the redundant nodes in the network to offer multiple time slots for a node to forward packet so that the transmission delay in each hop can be decreased. However, the delay is still proportional with the length of duty cycle and the end-to-end delay in a multi-hop large-scale network is large. Moreover, most of these works do not consider the issue of transmission failure caused by unreliable wireless communication in practice.

Unlike the existing techniques, we propose a special sleeping scheme based on *pipeline schedule* to guarantee a low broadcasting delay in a large-scale WSN. As the alarm may be broadcasted from any possible node, we define two steps for the alarm broadcasting. First, when a node has detected a critical event, it originates an alarm and quickly transmits it to a predefined center node using the *pipeline schedule*. Then, the center node broadcasts the alarm to all other nodes also using the *pipeline schedule*. With designing a special wake-up pattern, the two *pipeline schedules* can be both carried out by the node simultaneously without causing interference and each node just needs to wake up for no more than τ time, no matter when the alarm comes and where it comes from. Hence, the delay of alarm broadcasting is greatly decreased.

To eliminate the collision in the broadcasting phase, we establish a Colored Connected Dominant Set (CCDS) using the *iterative minimal covering* (IMC) algorithm described in [28]. Each node wakes up to receive or transmits according to its colors defined as frequency channels instead of time slots. That is to say, we partition

Table 4.3: Number of channels in typical radio chips

Type of the chips	Number of channels	Frequency band
<i>nRF903</i>	169	433/868/915MHZ
<i>nRF905</i>	169	433/868/915MHZ
<i>nRF2401</i>	125	2.4GHz
CC1100	16	315/433/915MHz
CC2420	16	2.4GHZ
CC2500	125	2.4GHz

the channel in frequency domain to avoid collision while keeping the transmission continuous in time domain. It can be proved that no more than 12 frequency channels are needed in any WSN, which can be easily obtained in most existing sensor nodes' chips. Table 4.3 shows number of channels in typical radio communication chips widely applied in the current WSNs.

4.3.1 Problem formulation

To describe the transmission delay among the sleeping nodes, we formulate the problem with the model described in [43]. Let us assume $G = (V, E)$ is an arbitrary graph. If the time slot $s \in \{0, \dots, L-1\}$ is assigned to node i , it will only wake up at slot s and will receive packets at that time slot. It can, however, transmit packets at any other time slot. Let us assume $f : V \rightarrow \{0, \dots, L-1\}$ is a time slot assignment function that assigns a time slot to every node in the graph. It is obvious that f determines the delay incurred in transmitting data from one node to the other. For a given f , let $d_f(i, j)$ be the delay in transmitting data from i to j where $(i, j) \in E$:

$$d_f(i, j) = \begin{cases} L & \text{if } f(i) = f(j), \\ (f(j) - f(i)) \bmod L & \text{otherwise.} \end{cases} \quad (4.10)$$

Then, delay on a path P under a slot assignment f is defined as:

$$d_f(P) = \sum_{(i,j) \in P} d_f(i, j) \quad (4.11)$$

We also define $F : V \rightarrow \{ch_1, \dots, ch_{12}\}$ be a channel assignment function that assigns a channel to every node to receive packet in the graph, where ch_i ($0 \leq i \leq 12$) stands for the frequency channel.

4.3.2 A reliable and fast scheduling (RealFast)

Generally, if we ignore collision, broadcasting is the most efficient way to transmit a packet to all neighboring nodes and the *pipeline schedule* is the most delay-efficient way to transmit a packet among sleeping nodes hop-by-hop. Hence, broadcasting with *pipeline schedule* can achieve the minimal delay for broadcasting alarm messages. However, *pipeline schedule* performs well only in one traffic direction. Since the location of an event is not known a priori and any nodes may originate an alarm, the direction of broadcasting traffic is surely arbitrary. Therefore, to utilize the *pipeline schedule* for broadcasting alarm messages from any node in an arbitrary WSN topology, we define two types of data traffics, i.e., (i) any node having detected the event transmits an alarm to a pre-determined node (called the center node) in uplink direction and (ii) the center node broadcasts the alarm to all other nodes in downlink direction.

Our proposed protocol, called RealFast, two steps, i.e., (i) establishing the two traffic paths in an arbitrary topology for all the nodes and (ii) designing the wake-up patterns for the sleeping nodes to carry out both uplink and downlink traffic without any waiting time, while still waking up each node no more than one time slot in average in each duty cycle .

To achieve a minimum delay for the uplink traffic, we establish a Breadth First Search (BFS) tree [61] to find the shortest path between the center node and each node. To achieve a minimum delay and to eliminate the collision in broadcasting for the downlink traffic, we use the *frequency division multiple access* (FDMA) [63] concept to avoid contention among the nodes while maintaining the transmission continuously in time domain to demand the requirement of the *pipeline schedule*. To keep number of frequency channels irrelative to scale and/or density of WSN deployment, we establish a Colored Connected Dominant Set (CCDS) among the nodes, where the density of independent nodes can be limited. Hence, the traffic of broadcasting from the center node (downlink traffic) can be coordinated using the CCDS with limited frequency channels. Then, independent nodes just broadcast locally and other nodes receive the alarm packets without contention.

4.3.2.1 Establishment of the traffic paths

Before establishing the two traffic paths in the arbitrary graph G , we find the center node defined following definition of [28]. A center node c is a node in a graph G such that the length of the shortest path from c to the farthest node is minimized [28].

Table 4.4: Definitions of variables used in the establishing CCDS algorithm

D	maximum diameter hop in the networks
c	center node in the networks
H_i	nodes with minimum hop i to c in G
I_i	independent nodes with minimum hop i to c in G
H'_i	nodes with minimum hop i to c in CCDS
I'_i	independent nodes with minimum hop i to c in CCDS
C_i	connector nodes with minimum hop i to c in CCDS
B_i	dominated nodes dominated by I'_i

- 1: **Input:** $G = (V, E)$
- 2: **Output:** the CCDS in G
- 3: Find the center node c by applying Floyd-Warshall algorithm.
- 4: Apply Alg. 8 to G, c to obtain the independent nodes I .
- 5: Apply Alg. 9 to G, c and I to obtain the connector nodes C .
- 6: Apply Alg. 10 to G, c to obtain the colors.

Algorithm 7: Establishment of CCDS

Center nodes may not be unique. If more than one node fulfills this definition, we simply select one of them. Center nodes can be found easily by applying the Floyd-Warshall algorithm [22]. It is obvious that the maximum length of shortest path from any node to the center is $D/2$.

It is easy to construct a Breadth First Search (BFS) tree for the network. Then, all nodes can be divided into $H_1, H_2, H_3, \dots, H_{D/2}$, where H_i is the set of nodes with the minimum hop i to c in G , and H_0 is the set that only contains the center node c . Hence, the uplink traffic path for the nodes can be easily obtained.

To establish the downlink traffic path, i.e., to design the CCDS in the arbitrary graph G , we proposed *Algorithm 7*. Individual elements are presented separately in Alg. 8, 9, 10 later, and the variables therein are defined in Table 4.4.

First, we construct a Maximal Independent Set (MIS) [65]. As all nodes have been divided into $H_1, H_2, H_3, \dots, H_{D/2}$ with constructing the BFS tree, the MIS can be established hop-by-hop (Alg. 8) in BFS as follows. Starting from the 0th hop, we pick up the maximum independent set and mark these nodes with I_0 . We then move on to the 1st hop, pick up the maximum independent set and mark these nodes with I_1 . Note that the independent nodes of the 1st hop also need to be independent of those of the previous hop. We repeat this process until all hops of nodes have been visited. The pseudo code of the MIS construction is given in Alg. 8.

Then, we construct the Connected Dominant Set (CDS). To obtain it, we need to

```

1: Input: BFS
2:  $I \leftarrow \emptyset$ 
3: for  $i \leftarrow 0$  to  $D/2$  do
4:   Find an MIS  $I_i \subset H_i$  also independent of  $I$ 
5:    $I \leftarrow I \cup I_i$ 
6: end for
7: return  $I$ 

```

Algorithm 8: Construction of MIS

pick some nodes from $V \setminus I$ to be connector nodes to interconnect all the independent nodes. First, we define the subgraph of G induced by the independent set I as $G(I)$ and compute $G' = G^2(I) = (I, E')$. If any two nodes n_i and n_j in I have an edge in G' , then there must be at least one node in G both adjacent to n_i and n_j , so that it can be selected as a connector node. We use the idea of IMC algorithm in [29,53] to select the connector nodes, which partitions I_i into four disjoint subsets $U_{i,j}$, $0 \leq j \leq 3$ and partitions four disjoint subsets $W_{i-1,j}$, $0 \leq j \leq 3$ among $(H_{i-1} \cup H_{i-2}) \cap \bar{I}$ to cover all the nodes in I_i . Hence, when nodes in $W_{i-1,j}$ broadcast simultaneously, they will not cause any collision among the nodes in $U_{i,j}$. The process of connector nodes selection is shown in Alg. 9.

```

1:  $C \leftarrow \emptyset$ 
2:  $I' \leftarrow \{c\}$ 
3: Computing  $G' = G^2(I) = (I, E')$ 
4: for  $i \leftarrow 2$  to  $\frac{D}{2} - 1$  do
5:    $W \leftarrow (H_{i-1} \cup H_{i-2}) \cap \bar{I}$ 
6:    $U \leftarrow I_i$ 
7:   Input:  $W$  and  $U$ 
8:   Applying IMC algorithm
9:   Output:
      $C_{i-1} \leftarrow W = W_{i-1,1} \cup W_{i-1,2} \cup W_{i-1,3} \cup W_{i-1,4}$ 
      $I_i \leftarrow U = U_{i,1} \cup U_{i,2} \cup U_{i,3} \cup U_{i,4}$ 
10:  for  $j \leftarrow 0$  to  $3$  do
11:     $E' \leftarrow$  edges between  $W_{i-1,j}$  and  $U_{i,j}$ 
12:  end for
13:   $I' \leftarrow I' \cup I_i$ 
14: end for
15:  $B \leftarrow (V \setminus I') \setminus C$ 
16: return  $C, I', B$  and CDS

```

Algorithm 9: Connector nodes selection

Now, we color the CDS to be CCDS in Alg. 10. First, all nodes in CDS are divided into several sets according to their minimum hop count to the center node. As the

```

1: Input: CDS
2: Output: CCDS
3: Divide  $I$  into  $I'_0, I'_2, I'_4, \dots$ 
4: Divide  $B$  into  $B_0, B_2, B_4, \dots$ 
5: Divide  $C$  into  $C_1, C_3, C_5, \dots$ 
6: Coloring  $G' = (I', E')$  with  $cl_1, \dots, cl_{12}$ , and each node in  $I'$  gets its sending color
7: for  $i \leftarrow 0$  to  $D - 1$  do
8:   for Each node  $n_k \in C_{i+1} \cup B_i$  do
9:      $cl_w(n_k) \leftarrow cl_s(n_t)$ ,  $n_t$  is any parents of  $n_k$  in  $I'_i$ 
10:   end for
11: for  $m \leftarrow 0$  to 3 do
12:   Each node in  $W_{i,m}$  obtains color  $cl_m$  as its sending color
13:   Each node in  $U_{i,m}$  obtains color  $cl_m$  as its waking color
14: end for
15: end for

```

Algorithm 10: Colors assignment

establishment of CDS is based on $G^2(I)$, it is easy to see that all minimum hop counts of nodes in I to the center node are even, and all minimum hop counts of nodes in C to the center node are odd. An example of CCDS is shown in Figure 4.23a (For comparison, we show the BFS of the same network in Figure 4.23b). I and C can be divided as $I'_0, I'_2, I'_4, \dots$ and $C_1, C_3, C_5, \dots$, respectively. We define *sending color* ($cl_s(n_k)$) and *waking color* ($cl_w(n_k)$) for each node n_k corresponding to the channels in which n_k sends packets and wakes up to receive packets, respectively. Then, $G' = (I', E')$ is colored with $cl_1, \dots, cl_{12}$, and each node n_k in I'_i gets its $cl_s(n_k)$. The color $cl_w(n_k)$ of node n_k in I'_i is obtained according to the subset $U_{i,j}$, which n_k belongs

to. Each node n_t in C_i obtains its $cl_s(n_t)$ according to its subset $W_{i,j}$, and obtains its $cl_w(n_t)$ according to the *sending color* of one of its parents in I'_{i-1} . Alg. 10 shows the coloring in detailed. We use only 12 colors for the independent nodes in I and 4 colors for the connector nodes in C , because each node in I has no more than 12 neighbors in I in 2-hop distance, and each node in C has no more than 4 neighbors in I in 1-hop distance [53].

Now all nodes have obtained their uplink traffic paths, downlink traffic paths, *sending colors* and *waking colors*. Next, we introduce their *wake-up patterns* to define when they need to wake up to listen the possible alarm and when they should send an alarm if they had received one.

4.3.2.2 Design of the wake up patterns

We periodically change the sleep schedule for different nodes, so that the alternation can be transparent to the alarm traffic from the source node to center node as well as the alarm broadcasting traffic from the center node to all the other nodes. The idea of this periodic change of nodes' sleep schedule is illustrated in Figure 4.24.

It can be seen from Figure 4.24 that although each node changes its sleep scheduling periodically, e.g. node n_j wakes up at slot 2 in even duty cycle and at slot $L - 1$ in odd duty cycle, the transmission of alarm in the uplink direction and the broadcasting of the alarm in the downlink direction occur continuously. To avoid the collision in broadcasting, a limited number of channels are used corresponding to the colors of nodes in CCDS.

It can be seen from our Alg. 11 that (i) each node in H_i or H'_i takes its time slot to wake up for uplink or downlink traffic according to its minimum hop count to the center node, (ii) each node takes the same time slot as $C_{i+1} \in H'_{i+1}$, (iii) in the downlink traffic mode, all nodes in CCDS wake up to receive packets in the channels corresponding to their *waking colors* and send packets in the channels corresponding to their *sending colors* in downlink, and (iv) in the uplink traffic mode, all nodes in CCDS receive or send packets in channel cl_1 .

Figure 4.25 shows the wake-up pattern and sending pattern for the three kinds of nodes in the network. It can be seen that, although the sensor nodes of different kinds have different wake up patterns, they all wake up for no more than one time slot in each duty cycle. Consequently, energy consumption of this protocol is ultra low due to idle listening when no critical event occurs.

```

1: for  $m \leftarrow 0$  to  $D/2 \bmod 2L - 1$  do
2:   for  $i \leftarrow 0$  to  $L - 1$  do
3:     for each node  $n_j \in H_{2mL+i}$  do
4:        $f(n_j) = L - i$  and  $F(n_j) = cl_1$  in  $(2k + 1) * L$  duty cycle
5:     end for
6:     for each node  $n_j \in H_{(2m+1)L+i}$  do
7:        $f(n_j) = L - i$  and  $F(n_j) = cl_1$  in  $2k * L$  duty cycle
8:     end for
9:   end for
10: end for
11: for  $m \leftarrow 0$  to  $D \bmod 2L - 1$  do
12:   for  $i \leftarrow 0$  to  $L - 1$  do
13:     for each node  $n_j \in H'_{2mL+i}$  do
14:        $f(n_j) = i$  and  $F(n_j) = cl_w(n_j)$  in  $2k * L$  duty cycle
15:     end for
16:     for each node  $n_j \in H'_{(2m+1)L+i}$  do
17:        $f(n_j) = i$  and  $F(n_j) = cl_w(n_j)$  in  $(2k + 1) * L$  duty cycle
18:     end for
19:     for each node  $n_j \in B_i$  do
20:        $f(n_j) = f(n_t)$  and  $F(n_j) = cl_w(n_t)$ , where  $n_t$  is any node in  $C_{i+1}$ 
21:     end for
22:   end for
23: end for

```

Algorithm 11: Wake-up pattern design

4.3.3 Performance evaluation using simulation

4.3.3.1 Simulation setup

We use *ns-2* simulator to evaluate the performances of the proposed scheduling method in unsteady WSNs. In Figure 4.26, 225 sensor nodes are randomly deployed in an area of $150 * 150m^2$. The successful communication probability p to characterize the wireless link between any two nodes is employed. Considering the interference caused by non-neighboring nodes, we define the worse link quality than that in practice with assumption $p = 1 - (\frac{d}{20})^2$, where d is the distance between two nodes and $d < 20$. The links with $p \geq 50\%$ are chosen to form the topology of network for the proposed scheme, as shown in Figure 4.26. The dashed lines are the links with $p < 50\%$. The duty cycle is 1s.

For comparison, we also conduct some simulations for the *ADB* [55] and *DW-MAC* [54] schemes. In *ADB* scheme, as sensor nodes work with asynchronous duty cycles, the average transmission delay in each hop is actually about half of the duty cycle in ideal environment. For fair comparison, we suppose a determined broadcast

Figure 4.24: Periodic alternation of sleep scheduling

tree is established with a priori knowledge of the link quality so as to simplify the overhead in ADB. In *DW-MAC* scheme, as sensor nodes have to reserve data transmissions by transmitting the scheduling frame (SCH) in the short active time, the packet transmission delay is actually determined by the maximum hop counts of SCH transmission within the active time. For fair comparison, we improve *DW-MAC* scheme for critical event monitoring as follows (called as improved *DW-MAC*). As few data packets are transmitted in the network and the alarm is just a sign with small size, we regard the SCH in *DW-MAC* as the alarm packet and assume their sizes are the same. Hence, we are concerned just with the SCH (i.e., alarm) broadcasting delay in the network. We ignore the SYNC duration and SIFS duration in *DW-MAC* and assume multiple channels are already appropriately assigned among nodes to avoid collision in the SCH broadcasting. The configuration of nodes' duty cycle in three schemes are shown in Table 4.5. The sizes of active time in three schemes are set the same so as to compare their performance in the same level of energy consumption during the monitoring.

Table 4.5: Duty cycle configuration

	Active time	Duty cycle
RealFast	$T_{data} = timeslot$	1s
<i>DW-MAC</i>	$T_{sync} = 0ms, T_{data} = timeslot$	1s
ADB	$T_{beacon} = 0ms, T_{data} = timeslot$	1s

Figure 4.25: Wake-up pattern and sending pattern of the nodes

4.3.3.2 Simulation results

- Different time slot sizes:

We first set the size of the time slot to be the minimum time τ for sensor nodes to transmit an alarm packet, e.g. $2ms$. When an alarm transmission fails between two adjacent nodes with the proposed scheme, the sender node has to retransmit the alarm after 2 duty cycles. While, for the *ADB* and the improved *DW-MAC* schemes, the sender node retransmits the alarm after 1 duty cycle. Figure 4.27 (a) shows the broadcasting delay with the three schemes in the WSN shown in Figure 4.26. Obviously, the proposed scheme does not exhibit good performance in the case of minimum time slot. To improve it, we set the size of the time slot to be $10ms$. Each sensor node still listens for $2ms$ during each duty cycle. When a sensor node wakes up to listen to the channel and detects a collision or a failing reception during the $2ms$, it will keep listening and receiving till the end of this time slot. Accordingly, when the sender node finds that it fails to transmit the alarm packet during the $2ms$, it will keep retransmitting the packet till the end of the time slot. With this improvement, sensor nodes may successfully retransmit packets within a time slot and do not need to retransmit packets after 2 duty cycles. Hence, the transmission delay could be largely reduced. Figure 4.27 (b) shows the simulation results in the same network shown in Figure 4.26. It can be seen, the broadcasting delay with the proposed scheme becomes much lower when the size of time slot is $10ms$. Similarly, it is shown in Figure 4.27 (b) that,

Figure 4.26: The distribution of nodes in an unsteady WSN

Figure 4.27: Broadcasting delay with different sizes of time slot

as the links in the network are unsteady, transmissions in the experiments are randomly successful, which affects the results in the experiments. For example, in experiments 1 and 8, packets usually can not be successfully transmitted within a time slot, and have to be retransmitted after 2s in next duty cycle. Therefore, the delay becomes large. Compared with the proposed scheme, the delay with the *ADB* and the improved *DW-MAC* schemes is even larger in most experiments. As sensor nodes in *ADB* wake up asynchronously, the average transmission delay in each hop is at least about half a duty cycle even if all transmissions were successful. While, for the improved *DW-MAC*, because the SCH (i.e., alarm) is forwarded within synchronous time slots, the number of hop counts of SCH transmission in each duty cycle is restricted by the size of time slot T_{data} . In ideal case, the number is T_{data} / τ . However, due to unsteady

Table 4.6: Average delay/standard deviation in different networks ($timeslot=0.01s$)

Network	1	2	3	4	5
RealFast	5.4s/4.8	3.1/4.7	4.2/5.3	5.9/5.7	3.6/3.8
DW-MAC	15.5/4.3	16.4/6.4	13.4/5.4	15.7/6.6	12.1/5.8
ADB	14.2/2.1	13.7/2.1	11.9/2.2	14.9/2.7	10.8/2.0

links, the number is dynamic. Hence, the number of duty cycles needed for the broadcasting in the network is random, resulting in highly dynamic results in the experiments.

We further enlarge the time slot to be $20ms$ for the three schemes and corresponding results are shown in Figure 4.27 (c). It can be seen, the proposed scheme achieves a distinct predominance to the other two schemes. Moreover, the broadcasting delay with the proposed scheme and the ADB scheme becomes much steadier in 10 experiments, as almost each packet can be successfully transmitted within $20ms$. While, the delay with the improved DW-MAC is still dynamic, because the number of hop counts of alarm transmission in each duty cycle is still uncertain due to unsteady links.

It is unnecessary to further enlarge the size of time slot, because the performance of the proposed scheme could not be further promoted. On the other hand, further enlargement of time slot increases energy consumption of sensor nodes, especially for the improved DW-MAC as nodes have to keep awake during the whole of the synchronous time slot.

We conduct more experiments with the schemes in several networks. All the networks are generated randomly with 225 sensor nodes. In each network, we made 20 experiments and the average broadcasting delay with the standard deviation is shown in Table 4.6 and Table 4.7, respectively. For example, the average broadcasting delay in network 1 with the proposed scheme is 5.4s and the standard deviation of the delay is 4.8s, which is denoted as 5.4/4.8 in Table 4.6. From Table 4.6 and Table 4.7, the average broadcasting delay of the proposed scheme is always much lower than that of the other two methods. When $timeslot=0.02s$, the delay of the proposed scheme almost keeps invariable in experiments in each network.

- Multiple alarms: In some cases, the critical event may trigger several alarms in the network, and they may be sent to a parent node when it wakes up. To deal with the collision, we design a mechanism for the proposed scheduling as

Table 4.7: Average delay/standard deviation in different networks ($timeslot=0.02s$)

Network	1	2	3	4	5
RealFast	0.48/0	0.52/0	0.42/0	0.50/0	0.38/0
DW-MAC	9.2/2.5	9.6/3.3	8.5/1.8	9.9/2.9	8.3/1.9
ADB	12.4/0.3	13.4/0.3	10.9/0.2	13.0/0.3	9.5/0.2

Figure 4.28: Transmission delay for multiple alarms

follows. Suppose the time slot is denoted as $k * \tau$. When a sensor node having detected the event is going to send an alarm packet, it keeps transmitting the packet randomly with the probability $1/2$ during the time slot. However, if the node detects some others are transmitting alarm packets during the same time slot, it gives up its transmission. Through this way, the nodes sending alarms could be decreased gradually. Note that, the parent node just needs to successfully receive one alarm. The parent node can not judge whether there is an alarm packet by just detecting the channel, because some configuration packets also need to be transmitted in the network and the alarm packet needs to be exactly received to avoid misinformation.

We evaluate the performance of the mechanism with a simple and typical network model. Suppose there are M ($1 \leq M \leq 10$) nodes that need to send packets to a parent node which keeps awake for $20ms$ every two duty cycles periodically. The quality of the link between the parent node and each children is 70%. Suppose the range of the event region is smaller than that of nodes' radio detection. Figure 4.28 shows the time when the parent node successfully receives a packet. For each value of M , we conduct 20 experiments and give the maximum time. It is obviously from Figure 4.28 that, when $M < 8$, the M

children nodes can successfully send one packet to their parent within the $20ms$. When $M = 8, 9$, or 10 , it needs two duty cycles to send the packet, resulting in $2s$ extra delay. However, the total broadcasting delay is still much lower than that of the improved *DW-MAC* and *ADB* schemes, according to Figure 4.27 (c).

- **Energy consumption:** We also analyze the energy consumption of sensor nodes with the proposed scheme in WSN. Since the energy consumption is mainly due to the idle listening when there is no critical event most of the time, it is reasonable for us to approximatively calculate the energy consumption according to the length of wake-up duration in a duty cycle. For example, when a MicaZ node turns on its radio module, its current is about $20mA$. Hence, the energy consumption within $5ms$ wake-up duration is about $3.3V * 20mA * 5ms = 3.3mJ$.

As described above, most sensor nodes with the proposed scheme stay awake for only τ in a duty cycle and some even stay awake for τ every two cycles. Therefore, the energy consumption of the proposed scheme could not be higher than that of most existing schemes. In addition, when the size of time slot is enlarged, each sensor node with the proposed scheme does not need to keep awake during the whole of the time slot. If a node does not detect the busyness of channel when it wakes up, it will go to sleep immediately.

Note that before applying the proposed scheme, some initialization efforts need to be done first, such as obtaining the topology and broadcasting the assignment, which would cost sensor nodes some extra energy. However, this kind of initial cost could be amortized by the afterward low energy consumption of the proposed scheme in the long term of monitoring.

4.4 Summary

In this chapter, we proposed three network protocols satisfying different performance requirements, network topology, and scale of WSNs. OPS is an opportunistic energy-efficient cross-layer scheduling and routing protocol for low duty cycle linear WSNs with unreliable links. Being based on multi-pipeline scheduling concept, it provides more retransmission chances for sensor nodes than traditional multi-pipeline scheduling methods, while does not require more active time or transmissions of sensor nodes in view of a given amount of packets to be delivered. Moreover, it does not strictly require multiple parent nodes in each layer of the data delivery path. Different from multi-pipeline scheduling employing multiple receivers assigned with different

slots, OPS spends these slots on one receiver (by setting larger gap), while saving the multiple receivers to provide opportunistic links without extra slots. Our evaluation results show that OPS outperforms multi-pipeline scheduling in terms of delay and energy consumption. In addition, OPS does not strictly require that each node on the main forwarding path should have neighboring nodes outside of the path to provide the opportunistic links. However, the multi-pipeline scheduling performance highly depends on the number of candidate receivers in each segment of the linear WSNs.

MobiBone is an opportunistic energy-efficient cross-layer MAC and routing protocol for mobile large-scale WSNs interacting with none time-critical static backbone infrastructure. It is based on the pipeline sleep scheduling with an extra functionality to adapt the duty cycle based on the traffic. The supported traffics are uplink, downlink, and random access. The protocol utilizes three components, i.e., (i) traffic learning, (ii) flexible time slot setting, and (iii) flexible duty cycle setting to support successful data delivery between fast moving nodes and fixed backbone infrastructure. Evaluation results clearly show that an adaptive protocol and approach is beneficial for achieving a good trade off.

RealFast is a reliable and fast cross-layer MAC and routing protocol for large-scale time-critical event detection applications of WSNs. It guarantees a low packet delivery while consuming low energy. The upper bound of the broadcasting delay is $3/2D + 2L$, where $1/2D$ is the maximum hop counts between sensor nodes and the center node, while L is the length of duty cycle (whose the unit is the size of the time slot). As the delay is only a linear combination of hops and duty cycle, it is very small even for large-scale WSNs. The broadcasting delay is independent of the length of the duty cycle, but it increases linearly with number of hops. The broadcasting delay is also independent of node density. Energy consumption of our protocol is very low as nodes wake up for only one time slot in the duty cycle.

CHAPTER 5

Conclusion and future direction

In this thesis we presented design and tooling solutions as well as network protocols to support application experts in the entire development life-cycle of wireless sensor networks. The complete life-cycle of wireless sensor networks starts with the user/application requirement analysis. It then goes through the following steps: hardware schematic design, PCB design, system software design, network protocol design, application software design, testing, debugging, and re-programming. In addition to being error-prone and time consuming tasks, all these steps require expertise, experience, in-depth knowledge about electronics and computer science, which application experts often do not possess. Instead of large quantities of WSN platforms with perfect design, application experts often require limited number of custom-designed wireless sensor node platforms, often not commercially available, for fast prototyping. They are interested in platforms that can be easily used and experimented with and can easily be adapted into their/applications requirements to find the best and most appropriate setup (software and hardware wise). As such availability of accurate and easy-to-use support tools, which can be used by application experts to translate their requirements easily and efficiently into a hardware schematic and associated software code and to make the entire process of hardware and software design easier, faster, and more efficient would greatly help bridge the gap between the demands of application experts and their WSN's expertise and knowledge.

The main focus of this thesis is on providing tools and mechanisms for application experts in general, and environmental and ecological scientists in particular, to support the entire life-cycle of wireless sensor nodes, facilitating and accelerating the process. To this end, the main research question addressed in this thesis is:

Can the entire process of design and development of wireless sensor networks be made efficient,

easy, fast, less error-prone and open to application experts in the field of environmental monitoring and ecology who have no a priori knowledge about WSNs?

In Chapter 2 we presented an automatic hardware schematic design tool (AHST) to automatically generate hardware schematics of the entire system according to the user requirements. We first presented a tool for modeling the major hardware components used in WSN platforms. The model, which is filled in by WSN experts to model existing commercially available hardware components, is stored in the hardware database used by AHST tool. We presented two simple templates to model user requirements as well as hardware module requirements. Both these model templates will be filled in by the application experts to express their functional and non-function requirements. To fill in these templates, they can either use the given graphical user interface or the given script. Based on our optimization and search algorithms, AHST tool will find the most suitable set of hardware components satisfying the user requirements, The objective function for the optimization is based on various parameters including the lowest cost, the lowest energy consumption, and the smallest size. After finding the most suitable components, the tool takes care of the automatic pin allocation. We evaluated AHST tool using a proof-of-concept on two different designs, based on which prototypes were manufactured, tested, and currently are used in different projects. Having environmental and ecological scientists as our target users, these two designs were about an implant and a Birdtag. The two application experts, taking part in the evaluation, were unable to finish the design, which results in $\infty\%$ time saving for both designs using the AHST tool. Compared with the two WSN experts, the AHST tool led to in average 68% and 74% time saving for the design of implant and the Birdtag, respectively. The same results hold for application experts in terms of number of errors made, while compared with the two WSN experts, the AHST tool led to in average 60% and 50% less errors in the design of implant and the Birdtag, respectively.

In Chapter 3 we presented tools to facilitate generation, debugging, testing, and re-programming of device drivers and their initialization and to make them easier, faster, more flexible, resource-aware, and less error prone. The tight coupling between the generated hardware schematics of Chapter 2 and code generation, debugging, testing, and re-programming of Chapter 3 is a unique feature, which significantly contributes to the reduction of complexity of software design, reduction of making software errors, and speeding up the development life-cycle of the software design. By parsing the generated schematics of Chapter 2, the tool chain, link script, and

system initialization codes are fetched from the hardware database based on the type of processor used in the schematic design. Device drivers are also fetched from the database based on the type of devices used in the schematic. The code generation tool offers pre-defined functionalities for four software modules including system initialization, event-triggered task, device driver, and service loop. By providing these functionalities, the risk that the application experts make software error is minimized. Our tool also enables partial programming and debugging and by doing so speeds up the development life-cycle on the one hand and reduces the resource consumption of the WSN platform on the other hand. Because of not needing an extra hardware for debugging and re-programming, the entire process is made more efficient and faster. Evaluation of the tools has been done using a proof-of-concept to correct the wrong frequency of a LED blinking task.

In Chapter 4 we addressed the hypothesis that an integrated and cross-layer MAC and routing protocol design can fulfill conflicting requirements of static and mobile WSNs used in environmental monitoring and ecology applications. While reliability and energy efficiency can be ensured by the MAC, a cross-layer design integrating MAC and routing is needed to ensure the short data dissemination delay. We presented three cross-layer MAC and routing protocols to achieve a good tradeoff between energy-efficiency, reliability, and delay for (i) static chain-based WSNs with unreliable links, (ii) mobile WSNs interacting with none time-critical static backbone infrastructure, and (iii) large-scale time-critical event detection applications of WSNs.

Through our work presented in Chapter 2-4, we have shown that our hypothesis are correct and that indeed:

- *It is possible to support application experts in fast prototyping of WSNs platforms if hardware and software design are tightly coupled and hardware schematics, pin allocations and their connectivity, as well as code generation are done in an automatic and all-inclusive manner requiring little interaction from application experts.*

and:

- *An integrated and cross-layer MAC and routing protocol design can fulfill conflicting requirements of static and mobile WSNs used in environmental monitoring and ecology applications.*

5.1 Conclusions and lessons learned

Some of the valuable lessons learned during this research are:

- Requirements of application experts are not fixed, they are fascinated with the idea of wireless sensor networks and what they all can do and would like to try all the features. This makes the design process very challenging and very dynamic. As such tools to enable fast prototyping is of utmost importance for them.
- Tight coupling between hardware and software designs is very efficient and can reduce design time, complexity, and errors.
- Even WSN experts experience difficulty in hardware/software co-design and make errors during the design phase.
- Not all applications of WSNs require full-fledged operating systems running on the node. Fast prototyping requirement of application experts implies keeping things simple and as such no operating system is needed.
- It is possible to design tools to support the entire development life-cycle of various applications of WSNs, including wearable, underwater sensor networks, underground soil moisture monitoring, environmental monitoring, and animal monitoring.

5.2 Future work

Some of the issues still to be addressed are:

- Currently the hardware database is filled in by WSN experts manually using the information read from the data-sheets. This task can be eliminated by directly parsing the datasheets to make the entire design process less dependent on the WSN experts.
- By adding a simulation functionality, the generated hardware/software designs and codes can be verified before actual prototyping and manufacturing.

List of publications

- Seraj, F. and Kui Zhang and Turkes, O. and Meratnia, N. and Havinga, P.J.M. A Smartphone Based Method to Enhance Road Pavement Anomaly Detection by Analyzing the Driver Behavior (The 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing)
- Guo, Peng and Cao, Jiannong and Zhang, Kui (2015) Distributed Topological Convex Hull Estimation of Event Region in Wireless Sensor Networks without Location Information. IEEE Transactions on Parallel and Distributed Systems (TPDS), 26 (1). pp. 85-94. ISSN 1045-9219
- Guo, Peng and Meratnia, N. and Havinga, P.J.M. and Jiang, Hongbo and Zhang, Kui (2015) OPS: Opportunistic pipeline scheduling in long-strip wireless sensor networks with unreliable links. Wireless Networks, 21 (5). pp. 1669-1682. ISSN 1022-0038
- Karuppiiah Ramachandran, V.R. and Kui Zhang and Meratnia, N. and Havinga, P.J.M. Towards Implantable Body Sensor Networks Performance of MICS Band Radio Communication in Animal Tissue (The 10th International Conference on Body Area Networks)
- Guo, Peng and Jiang, Tao and Zhang, Qian and Zhang, Kui (2012) Sleep Scheduling in Critical Event Monitoring with Wireless Sensor Networks. IEEE Transactions on Parallel and Distributed Systems, 23 (2). pp. 345-352. ISSN 1045-9219
- Guo, Peng and Jiang, Tao and Yi, Youwen and Zhang, Qian and Zhang, Kui (2011) Novel Navigation Algorithm for Wireless Sensor Networks without

- Information of Locations. In: IEEE GLOBECOM 2011, 5-9 Dec 2011, Houston, Texas, USA. pp. 1-6. IEEE Communications Society. ISBN 978-1-4244-9268-8
- Zhang, Kui and Climent, S. and Meratnia, N. and Havinga, P.J.M. (2011) Practical problems of experimenting with an underwater wireless sensor node platform. In: Proceedings of the 7th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2011), 6-9 Dec 2011, Adelaide, Australia. pp. 365-370. IEEE Computer Society. ISBN 978-1-4577-0673-8
 - Zhang, Kui and van Kleunen, W.A.P. and Meratnia, N. and Havinga, P.J.M. and Tijs, E. (2011) Underwater sensing and communication platform. In: OCEANS 2011, 6-9 Jun 2011, Santander, Spain. pp. 1-6. IEEE Oceanic Engineering Society. ISBN 978-1-4577-0087-3
 - Zhang, Kui and Guo, P. and Meratnia, N. and Havinga, P.J.M. (2010) A practical localization solution for wireless sensor networks deployed in linear topography. In: Proceedings of the 2010 Sixth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 7-10 December, Brisbane, Australia. pp. 115-120. IEEE Computer Society. ISBN 978-1-4244-7176-8
 - Zhang, Kui and Meratnia, N. and Havinga, P.J.M. and Guo, P. (2010) Fast alarm broadcasting in critical event monitoring using wireless sensor networks. In: SenSys '10 - Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems, SenSys '10, 03 - 05 November 2010, Zurich, Switzerland. pp. 379-380. ACM. ISBN 978-1-4503-0344-6
 - Guo, Peng and Tao, Jiang and Zhang, Kui and Chen, Hsiao-Hwa (2009) Clustering algorithm in initialization of multi-hop wireless sensor networks. IEEE Transactions on wireless communications, 8 (12). pp. 5713-5717. ISSN 1536-1276
 - Zhang, Kui and Zhang, Qian and Tao, Jiang and Peng, Guo (2009) A Navigation System based on Sensor Network without Exit and Locations. In: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09), 04 Nov - 06 Nov 2009, Berkeley, California, USA. pp. 311-312. ACM. ISBN 978-1-60558-519-2

Appendices

Appendix A

Listing 1: Script for MSP430F5308

```
1 {
2   "Name": "MSP430F5308",
3   "Type": "CPU",
4   "Voltage": "1.8V- 3.6V",
5   "Current": "0.1uA - 23uA",
6   "Pins": [
7     {"Index": "1", "Functions": [{"Type": "GPIO",
8       "Description": "P6.0"}, {"Type": "COMPARATOR",
9       "Description": "CB0"}, {"Type": "ANALOG", "Description":
10      "A0"}]},
11    {"Index": "2", "Functions": [{"Type": "GPIO",
12      "Description": "P6.1"}, {"Type": "COMPARATOR",
13      "Description": "CB1"}, {"Type": "ANALOG", "Description":
14      "A1"}]},
15    {"Index": "3", "Functions": [{"Type": "GPIO",
16      "Description": "P6.2"}, {"Type": "COMPARATOR",
17      "Description": "CB2"}, {"Type": "ANALOG", "Description":
18      "A2"}]},
19    {"Index": "4", "Functions": [{"Type": "GPIO",
20      "Description": "P6.3"}, {"Type": "COMPARATOR",
21      "Description": "CB3"}, {"Type": "ANALOG", "Description":
22      "A3"}]}],
23 }
```


Appendix A - Hardware scripts

```
11  {"Index": "5", "Functions": [{"Type": "GPIO",
    "Description": "P5.0"}, {"Type": "ANALOG",
    "Description": "A8"}, {"Type": "VREF", "Description":
    "VeREF+"}]},
12  {"Index": "6", "Functions": [{"Type": "GPIO",
    "Description": "P5.1"}, {"Type": "ANALOG",
    "Description": "A9"}, {"Type": "VREF", "Description":
    "VeREF-"}]},
13  {"Index": "7", "Functions": [{"Type": "AVCC",
    "Description": "AVCC1"}]},
14  {"Index": "8", "Functions": [{"Type": "GPIO",
    "Description": "P5.4"}, {"Type": "CRYSTAL",
    "Description": "XIN"}]},
15  {"Index": "9", "Functions": [{"Type": "GPIO",
    "Description": "P5.5"}, {"Type": "CRYSTAL",
    "Description": "XOUT"}]},
16  {"Index": "10", "Functions": [{"Type": "AVSS",
    "Description": "AVSS1"}]},
17  {"Index": "11", "Functions": [{"Type": "DVCC",
    "Description": "DVCC1"}]},
18  {"Index": "12", "Functions": [{"Type": "DVSS",
    "Description": "DVSS1"}]},
19  {"Index": "13", "Functions": [{"Type": "VCORE",
    "Description": "VCORE"}]},
20  {"Index": "14", "Functions": [{"Type": "GPIO",
    "Description": "P1.0"}, {"Type": "CLOCK", "Description":
    "TA0CLK"}, {"Type": "CLOCK", "Description": "ACLK"}]},
21  {"Index": "15", "Functions": [{"Type": "GPIO",
    "Description": "P1.1"}, {"Type": "CLOCK", "Description":
    "TA0.0"}]},
22  {"Index": "16", "Functions": [{"Type": "GPIO",
    "Description": "P1.2"}, {"Type": "CLOCK", "Description":
    "TA0.1"}]},
23  {"Index": "17", "Functions": [{"Type": "GPIO",
    "Description": "P1.3"}, {"Type": "CLOCK", "Description":
    "TA0.2"}]},
```

```
24  {"Index": "18", "Functions": [{"Type": "GPIO",
    "Description": "P1.4"}, {"Type": "CLOCK", "Description":
    "TA0.3"}]},
25  {"Index": "19", "Functions": [{"Type": "GPIO",
    "Description": "P1.5"}, {"Type": "CLOCK", "Description":
    "TA0.4"}]},
26  {"Index": "20", "Functions": [{"Type": "GPIO",
    "Description": "P1.6"}, {"Type": "CLOCK", "Description":
    "TA1CLK"}, {"Type": "COMPARATOR", "Description":
    "CBOUT"}]},
27  {"Index": "21", "Functions": [{"Type": "GPIO",
    "Description": "P1.7"}, {"Type": "CLOCK", "Description":
    "TA1.0"}]},
28  {"Index": "22", "Functions": [{"Type": "GPIO",
    "Description": "P2.0"}, {"Type": "CLOCK", "Description":
    "TA1.1"}]},
29  {"Index": "23", "Functions": [{"Type": "GPIO",
    "Description": "PJ.0"}, {"Type": "JTAGTDO",
    "Description": "TDO"}]},
30  {"Index": "24", "Functions": [{"Type": "GPIO",
    "Description": "PJ.1"}, {"Type": "JTAGTDI",
    "Description": "TDI"}, {"Type": "CLOCK", "Description":
    "TCLK"}]},
31  {"Index": "25", "Functions": [{"Type": "GPIO",
    "Description": "PJ.2"}, {"Type": "JTAGTMS",
    "Description": "TMS"}]},
32  {"Index": "26", "Functions": [{"Type": "GPIO",
    "Description": "PJ.3"}, {"Type": "JTAGTCK",
    "Description": "TCK"}]},
33  {"Index": "27", "Functions": [{"Type": "DVSS",
    "Description": "DVSS2"}]},
34  {"Index": "28", "Functions": [{"Type": "DVCC",
    "Description": "DVCC2"}]},
```

Appendix A - Hardware scripts

```
35 {"Index": "29", "Functions": [{"Type": "GPIO",
    "Description": "P4.0"}, {"Type": "STE", "Description":
    "PM_UCB1STE"}, {"Type": "XX", "Description":
    "PM_UCA1CLK"}]},
36 {"Index": "30", "Functions": [{"Type": "GPIO",
    "Description": "P4.1"}, {"Type": "SPISIMO",
    "Description": "PM_UCB1SIMO"}, {"Type": "IICSDA",
    "Description": "PM_UCB1SDA"}]},
37 {"Index": "31", "Functions": [{"Type": "GPIO",
    "Description": "P4.2"}, {"Type": "SPISOMI",
    "Description": "PM_UCB1SOMI"}, {"Type": "IICSCL",
    "Description": "PM_UCB1SCL"}]},
38 {"Index": "32", "Functions": [{"Type": "GPIO",
    "Description": "P4.3"}, {"Type": "SPICLK",
    "Description": "PM_UCB1CLK"}, {"Type": "STE",
    "Description": "PM_UCA1STE"}]},
39 {"Index": "33", "Functions": [{"Type": "GPIO",
    "Description": "P4.4"}, {"Type": "UARTTXD",
    "Description": "PM_UCA1TXD"}, {"Type": "SPISIMO",
    "Description": "PM_UCA1SIMO"}]},
40 {"Index": "34", "Functions": [{"Type": "GPIO",
    "Description": "P4.5"}, {"Type": "UARTRXD",
    "Description": "PM_UCA1RXD"}, {"Type": "SPISOMI",
    "Description": "PM_UCA1SOMI"}]},
41 {"Index": "35", "Functions": [{"Type": "GPIO",
    "Description": "P4.6"}]},
42 {"Index": "36", "Functions": [{"Type": "GPIO",
    "Description": "P4.7"}]},
43 {"Index": "37", "Functions": [{"Type": "VSS",
    "Description": "VSSU"}]},
44 {"Index": "38", "Functions": [{"Type": "GPIO",
    "Description": "PU.0"}]},
45 {"Index": "39", "Functions": [{"Type": "NC", "Description":
    "NC"}]},
46 {"Index": "40", "Functions": [{"Type": "GPIO",
    "Description": "PU.1"}]},
```

```

47  {"Index": "41", "Functions": [{"Type": "LDO",
    "Description": "LDO1"}]},
48  {"Index": "42", "Functions": [{"Type": "LDO",
    "Description": "LDO0"}]},
49  {"Index": "43", "Functions": [{"Type": "NC", "Description":
    "NC"}]},
50  {"Index": "44", "Functions": [{"Type": "AVSS",
    "Description": "AVSS2"}]},
51  {"Index": "45", "Functions": [{"Type": "GPIO",
    "Description": "P5.2"}, {"Type": "CRYSTAL",
    "Description": "XT2IN"}]},
52  {"Index": "46", "Functions": [{"Type": "GPIO",
    "Description": "P5.3"}, {"Type": "CRYSTAL",
    "Description": "XT2OUT"}]},
53  {"Index": "47", "Functions": [{"Type": "TEST",
    "Description": "P5.2"}, {"Type": "SBWTCK",
    "Description": "SBWTCK"}]},
54  {"Index": "48", "Functions": [{"Type": "REST",
    "Description": "Negative"}, {"Type": "NMI",
    "Description": "NMI"}, {"Type": "SBWTDIO",
    "Description": "SBWTDIO"}]},
55  }

```

Listing 2: Code for MSP430F5308

```

1  {
2    struct cpu_t msp430f5308 {
3      struct pin_t
4    };
5  }

```

(a) MSP430F5308

(b) GPS

(c) GPRS module (GU900)

(d) Power circuits

Figure 2: Birdtag Hardware Schematic

Appendix C

Generated code for hardware schematic.

```
#include "hw.h"
#define LIST_HEAD_INIT(name) { &(amp;name), &(name) }
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}

static inline void __list_add(struct list_head *new,
                             struct list_head *prev,
                             struct list_head *next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}

static inline void list_add(struct list_head *new, struct list_head
                           *head)
{
    __list_add(new, head, head->next);
}
```


Appendix C - Generated code for hardware schematic

```
static inline void list_add_tail(struct list_head *new, struct
    list_head *head)
{
    __list_add(new, head->prev, head);
}

#define list_entry(ptr, type, member) \
    ((type *) ((char *) (ptr) - (unsigned long) (&((type
        *) 0)->member)))

msp430f5308_pins[0] = {
    .index = 0,
    funcs = {struct func},
};

struct pin msp430f5308_pins = {

};

struct hwcomponent msp430f5308 = {
    .name = "MSP430F5308",
    .model = "IPT" ,
    .pin_total = 48,
    .pins = msp430f5308_pins,
    .voltage_range = {3.6, 1.8},
    .current_range = {},
    .datasheet_url =
        "http://www.ti.com/lit/ds/slas677e/slas677e.pdf",
    .rating = 0,
    .feedback = "",
};

struct hwcomponent mpu6050 = {
    .name = "MPU6050",
};

struct hwcomponent gu906 = {
    .name = "GU906",
};
```

```
struct hwcomponent cam_m8 = {
    .name = "CAM-M8",
};

void parseDesign



---


#include <stdio.h>
#include <string.h>
#include "uart0.h"
#include "watchdog.h"

int main(int argc, char **argv)
{
    //processor initialization
    msp430CpuInit();
    clockInit();

    //debug uart initialization
    uart0Init(115200, 8, 'N', 1);

    processInit();

    int int_flags;
    while(1) {
        while(processCurrent()) {
            if(processNeedRun()) {
                processRun();
            }
            processNext();
        }

        int_flags = splHigh();

        if(sysEvents()) {
            splX(int_flags);
        }
        else {
            watchdogStop();
        }
    }
}
```

Appendix C - Generated code for hardware schematic

```
        _BIS_SR(GIE | CPUOFF);  
        watchdogStart();  
    }  
}  
  
return 0;  
}
```

Appendix D

Generated code for the system clock.

```
void msp430InitDCO(u32 fclk)
{
    //Set DCO FLL reference = REFO
    UCSCTL3 |= SELREF_2;
    //Set ACLK = REFO
    UCSCTL4 |= SELA_2;

    UCSCTL0 = 0x0000;

    //loop until xt1, xt2, dco stabilized
    do {
        UCSCTL7 &= ~(XT2OFFG | XT1LFOFFG | DCOFFG);
        SFRIFG1 &= ~OFIFG;
    } while(SFRIFG1 & OFIFG);

    setVcoreUp(0x01);
    asmv("bis %0, r2" : : "i" (SCG0)); //disable fll control loop
    UCSCTL1 = DCORSEL_5; //Set DCO range 24MHz operation
    UCSCTL2 |= (fclk >> 15) - 1; //Set DCO Multipiler for 12MHz
    //(N + 1) * FLLRef = Fdco
    //(374 + 1) * 32768 = 12MHz
    //Set FLL Div = Fdco/2

    asmv("bic %0, r2" : : "i" (SCG0)); //enable fll control loop
```

Appendix D - Generated code for system clock

```
    //wait for stablize the dco: fclk >> 5
    __delay_cycles(250000);
}
```

Appendix E

Generated code for the interface driver.

```
void uart0Init(u32 ubr)
{
    PMAPPWD = 0x02D52;
    P1MAP5 = PM_UCA0RXD;
    P1MAP6 = PM_UCA0TXD;
    PMAPPWD = 0;

    P1DIR |= BIT6;
    P1SEL |= BIT5 | BIT6;
    UCA0CTL1 |= UCSWRST;
    UCA0CTL1 |= UCSSEL_2;    /* CLK = SMCLK */

    ubr = (12000000 / ubr);
    UCA0BR0 = ubr & 0xff;
    UCA0BR1 = (ubr >> 8) & 0xff;
    UCA0MCTL = UCBRS_3;    /* Modulation UCBRSx = 3 */

    UCA0IE &= ~UCRXIFG;
    UCA0IE &= ~UCTXIFG;

    UCA0CTL1 &= ~UCSWRST;
    UCA0IE |= UCRXIE;
}
```

Appendix E - Generated code for driver

```
void __attribute__((interrupt(USCI_A0_VECTOR)))
uart0RxIntHandler(void)
{
    u8 c;
    if(UCA0IV == 2) {
        if(UCA0STAT & UCRXERR) {
            c = UCA0RXBUF;
        }
        else {
            c = UCA0RXBUF;
            UCA0TXBUF = c;
            uart0IntSecondHalf(c);
        }
    }
}
```

Appendix F

Generated code for LED blinking according to serial port input.

```
#include <stdio.h>

uint16_t blink_freq;
uint8_t blink_freq_ch[4];
uint8_t blink_freq_index;
uint8_t parseBlinkFrequency(uint8_t c)
{
    if(c == '\n') {
        sscanf(blink_freq_ch, "%d", &blink_freq);
        blink_freq_index = 0;
        return 1;
    }
    blink_freq_ch[blink_freq_index++] = c;
    return 0;
}

int main(int argc, char **argv)
{
    //processor initialization
    msp430CpuInit();
    clockInit();

    //debug uart initialization
    uart0Init(115200, 8, 'N', 1);
    uart0SetSecondHalf(configureBlinkFrequency);
}
```


Appendix F - Code for LED blink

```
processInit();

blink_freq_index = blink_freq = 0;
int int_flags;
while(1) {
    while(processCurrent()) {
        if(processNeedRun()) {
            processRun();
        }
        processNext();
    }

    int_flags = splHigh();

    if(sysEvents()) {
        splX(int_flags);
    }
    else {
        watchdogStop();
        _BIS_SR(GIE | CPUOFF);
        watchdogStart();
    }
}

return 0;
}

void processLedToggle()
{
    if(blink_freq)
        toggleLED(blink_freq);
}
```

Bibliography

- [1] Armboot. <http://armboot.sourceforge.net/>.
- [2] Btnut. <http://www.btnode.ethz.ch/>.
- [3] Msp430bsl. <http://www.ti.com/tool/MSPBSL>.
- [4] Redboot. <http://ecos.sourceware.org/redboot/>.
- [5] U-boot. https://en.wikipedia.org/wiki/Das_U-Boot.
- [6] Fadel Adib, Hongzi Mao, Zachary Kabelac, Dina Katabi, and Robert C. Miller. Smart homes that monitor breathing and heart rate. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15*, pages 837–846, New York, NY, USA, 2015. ACM.
- [7] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.
- [8] Sanjit Biswas and Robert Morris. Exor: Opportunistic multi-hop routing for wireless networks. *SIGCOMM Comput. Commun. Rev.*, 35(4):133–144, August 2005.
- [9] N. Bouabdallah, M.E. Rivero-Angeles, and B. Sericola. Continuous monitoring using event-driven reporting for cluster-based wireless sensor networks. *Vehicular Technology, IEEE Transactions on*, 58(7):3460–3479, Sept 2009.

- [10] M.I. Brownfield, K. Mehrjoo, A.S. Fayez, and IV Davis, N.J. Wireless sensor network energy-adaptive mac protocol. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 2, pages 778–782, Jan 2006.
- [11] J. Burrell, T. Brooke, and R. Beckwith. Vineyard computing: sensor networks in agricultural production. *Pervasive Computing, IEEE*, 3(1):38–45, Jan 2004.
- [12] Qing Cao, Tarek Abdelzaher, Tian He, and John Stankovic. Towards optimal sleep scheduling in sensor networks for rare-event detection. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN '05*, Piscataway, NJ, USA, 2005. IEEE Press.
- [13] Yongle Cao, Shuo Guo, and Tian He. Robust multi-pipeline scheduling in low-duty-cycle wireless sensor networks. In *INFOCOM, 2012 Proceedings IEEE*, pages 361–369, March 2012.
- [14] Chang Wen Chen and Yu Wang. Chain-type wireless sensor network for monitoring long range infrastructures: Architecture and protocols. *IJDSN*, 4(4):287–314, 2008.
- [15] P. Corke, T. Wark, R. Jurdak, Wen Hu, P. Valencia, and D. Moore. Environmental wireless sensor networks. *Proceedings of the IEEE*, 98(11):1903–1917, Nov 2010.
- [16] M.G.R. Degrauwe, O. Nys, E. Dijkstra, J. Rijmenants, S. Bitz, B.L.A.G. Goffart, E.A. Vittoz, S. Cserveny, C. Meixenberger, G. van der Stappen, and H.J. Oguey. Idac: an interactive design tool for analog cmos circuits. *Solid-State Circuits, IEEE Journal of*, 22(6):1106–1116, Dec 1987.
- [17] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, LCN '04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.
- [18] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys '06*, pages 29–42, New York, NY, USA, 2006. ACM.
- [19] Sherine M. Abd El-kader and Basma M. Mohammad El-Basioni. Precision farming solution in egypt using the wireless sensor network technology. *Egyptian Informatics Journal*, 14(3):221 – 233, 2013.

- [20] Sinem Coleri Ergen and Pravin Varaiya. Tdma scheduling algorithms for wireless sensor networks. *Wirel. Netw.*, 16(4):985–997, May 2010.
- [21] L. Evers. *Concise and flexible programming of wireless sensor networks*. PhD thesis, University of Twente, June 2010.
- [22] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.
- [23] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: A macro-programming system for wireless sensor networks. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05*, pages 1–2, New York, NY, USA, 2005. ACM.
- [24] Peng Guo, Nirvana Meratnia, Paul J. Havinga, Hongbo Jiang, and Kui Zhang. Ops: Opportunistic pipeline scheduling in long-strip wireless sensor networks with unreliable links. *Wirel. Netw.*, 21(5):1669–1682, July 2015.
- [25] Shuo Guo, Yu Gu, Bo Jiang, and Tian He. Opportunistic flooding in low-duty-cycle wireless sensor networks with unreliable links. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking, MobiCom '09*, pages 133–144, New York, NY, USA, 2009. ACM.
- [26] Rebecca N. H, Dave L. Swain, Greg J. Bishop-hurley, Kym P. Patison, Philip Valencia, Peter Corke, and Christopher J. O’neill. Article monitoring animal behaviour and environmental interactions using wireless sensor networks, gps collars and satellite remote sensing, 2009.
- [27] R. Harjani, R.A. Rutenbar, and L.R. Carley. Oasys: a framework for analog circuit synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(12):1247–1266, Dec 1989.
- [28] S.C. Huang, Peng-Jun Wan, Xiaohua Jia, Hongwei Du, and Weiping Shang. Minimum-latency broadcast scheduling in wireless ad hoc networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 733–739, May 2007.
- [29] Scott C.-H. Huang, Hongwei Du, and E-K. Park. Minimum-latency gossiping in multi-hop wireless networks. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc '08*, pages 323–330, New York, NY, USA, 2008. ACM.

- [30] IEEE Circuit Theory Group and University of Notre Dame.(IN US). *Circuit theory : proceedings*, IEEE catalog, 11, Notre Dame, IN US, 1968. University of Notre Dame.
- [31] Imad Jawhar, Nader Mohamed, and Dharma P. Agrawal. Linear wireless sensor networks: Classification and applications. *J. Netw. Comput. Appl.*, 34(5):1671–1682, September 2011.
- [32] Jaein Jeong. Incremental network programming for wireless sensors. In *In Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON*, pages 25–33, 2004.
- [33] A. Kadri, E. Yaacoub, M. Mushtaha, and A. Abu-Dayya. Wireless sensor network for real-time air pollution monitoring. In *Communications, Signal Processing, and their Applications (ICCSPA), 2013 1st International Conference on*, pages 1–5, Feb 2013.
- [34] Mehdi Kargahi and Ali Movaghar. A method for performance analysis of earliest-deadline-first scheduling policy. *The Journal of Supercomputing*, 37(2):197–222, 2006.
- [35] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [36] Abtin Keshavarzian, Huang Lee, and Lakshmi Venkatraman. Wakeup scheduling in wireless sensor networks. In *Proceedings of the 7th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc '06*, pages 322–333, New York, NY, USA, 2006. ACM.
- [37] Joel Koshy. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *In Proceedings of the second European Workshop on Wireless Sensor Networks*, pages 354–365. IEEE Press, 2005.
- [38] Wim Kruiskamp and Domine Leenaerts. Darwin: Cmos opamp synthesis by means of a genetic algorithm. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, DAC '95*, pages 433–438, New York, NY, USA, 1995. ACM.
- [39] Koen Langendoen, Aline Baggio, and Otto Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS'06*, pages 174–174, Washington, DC, USA, 2006. IEEE Computer Society.

- [40] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.
- [41] G. Lu, B. Krishnamachari, and C.S. Raghavendra. An adaptive energy-efficient and low-latency mac for data gathering in wireless sensor networks. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 224–, April 2004.
- [42] G. Lu, B. Krishnamachari, and C.S. Raghavendra. An adaptive energy-efficient and low-latency mac for data gathering in wireless sensor networks. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 224–, April 2004.
- [43] G. Lu, N. Sadagopan, B. Krishnamachari, and A. Goel. Delay efficient sleep scheduling in wireless sensor networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2470–2481 vol. 4, March 2005.
- [44] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems, SenSys '04*, pages 39–49, New York, NY, USA, 2004. ACM.
- [45] Ioannis Mathioudakis, Neil M. White, Nick R. Harris, and Geoff V. Merrett. Wireless sensor networks: a case study for energy efficient environmental monitoring. In *EuroSensors 2008*, September 2008. Event Dates: 7-11 September 2008.
- [46] N. Mohamed, J. Al-Jaroodi, I. Jawhar, and S. Lazarova-Molnar. Failure impact on coverage in linear wireless sensor networks. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2013 International Symposium on*, pages 188–195, July 2013.
- [47] W.H. Ong and W.K. Chiu. Non-contact measurement techniques for structural health monitoring. *Procedia Engineering*, 75:45 – 50, 2014. International Conference on Materials for Advanced Technologies (ICMAT2013), Symposium W - Advanced Structural and Functional Materials for Protection.
- [48] Mohd Fauzi Othman and Khairunnisa Shazali. Wireless sensor network applications: A study in environment monitoring system. *Procedia Engineering*, 41:1204 –

- 1210, 2012. International Symposium on Robotics and Intelligent Sensors 2012 (IRIS 2012).
- [49] Tae Park and M.J. Lee. Power saving algorithms for wireless sensor networks on ieee 802.15.4. *Communications Magazine, IEEE*, 46(6):148–155, June 2008.
 - [50] F.J. Pierce and T.V. Elliott. Regional and on-farm wireless sensor networks for agricultural systems in eastern washington. *Computers and Electronics in Agriculture*, 61(1):32 – 43, 2008. Emerging Technologies For Real-time and Integrated Agriculture Decisions.
 - [51] Niels Reijers Reijers. Efficient code distribution in wireless sensor networks, 2003.
 - [52] R. Rutenbar, G. Gielen, and B. Antao. *OPASYN: A Compiler for CMOS Operational Amplifiers*, pages 56–68. Wiley-IEEE Press, 2002.
 - [53] Ivan Stoianov, Lama Nachman, Sam Madden, and Timur Tokmouline. Pipeneta wireless sensor network for pipeline monitoring. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN '07*, pages 264–273, New York, NY, USA, 2007. ACM.
 - [54] Yanjun Sun, Shu Du, Omer Gurewitz, and David B. Johnson. Dw-mac: A low latency, energy efficient demand-wakeup mac protocol for wireless sensor networks. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc '08*, pages 53–62, New York, NY, USA, 2008. ACM.
 - [55] Yanjun Sun, Omer Gurewitz, Shu Du, Lei Tang, and David B. Johnson. Adb: An efficient multihop broadcast protocol based on asynchronous duty-cycling in wireless sensor networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09*, pages 43–56, New York, NY, USA, 2009. ACM.
 - [56] Z. Taghikhaki, N. Meratnia, and P. J. M. Havinga. A reliable and energy-efficient chain-cluster based routing protocol for wireless sensor networks. In *Eighth IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing, IEEE ISSNIP 2013, Melbourne, Australia*, pages 248–253, USA, April 2013. IEEE Communications Society.

- [57] C. Toumazou, C.A. Makris, and C.M. Berrah. Isaid-a methodology for automated analog ic design. In *Circuits and Systems, 1990., IEEE International Symposium on*, pages 531–535 vol.1, May 1990.
- [58] Tijs van Dam and Koen Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys '03*, pages 171–180, New York, NY, USA, 2003. ACM.
- [59] N.A. Vasanthi and S. Annadurai. Aws: asynchronous wakeup schedule to minimize latency in wireless sensor networks. In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 1, pages 7 pp.–, June 2006.
- [60] N.A. Vasanthi and S. Annadurai. Energy efficient sleep schedule for achieving minimum latency in query based sensor networks. In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 2, pages 214–219, June 2006.
- [61] Wikipedia. Bfs. "https://en.wikipedia.org/wiki/Breadth-first_search".
- [62] Wikipedia. ELF.
- [63] Wikipedia. Fdma. "https://en.wikipedia.org/wiki/Frequency-division_multiple_access".
- [64] Wikipedia. Genetic algorithm. "https://en.wikipedia.org/wiki/Genetic_algorithm".
- [65] Wikipedia. Mis. [https://en.wikipedia.org/wiki/Independent_set_\(graph_theory\)](https://en.wikipedia.org/wiki/Independent_set_(graph_theory)).
- [66] Kui Zhang, S. Climent, N. Meratnia, and P.J.M. Havinga. Practical problems of experimenting with an underwater wireless sensor node platform. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2011 Seventh International Conference on*, pages 365–370, Dec 2011.
- [67] M. Zuniga and B. Krishnamachari. Analyzing the transitional region in low power wireless links. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 517–526, Oct 2004.