UiO : Department of Informatics
University of Oslo

Communication software model of WSN device for more accurate simulation in ns-3

Espen Volnes

Master's Thesis Spring 2018

Communication software model of WSN device for more accurate simulation in ns-3

Espen Volnes

May 16, 2018

Contents

I Introduction & Background 7

1 Introduction 9
1.1 Thesis statement . . . 10
1.2 Communication software (CSW) model . . . 10
1.3 Aim of thesis . . . 12
1.4 Outline . . . 13

2 Background 15
2.1 Wireless Sensor Network (WSN) . . . 15
2.2 WSN simulators . . . 21
2.3 Communication software model methodology (CSWMM) . . . 24
2.4 Related works . . . 27
2.5 Summary . . . 29

II Contributions 31

3 Requirements 35
3.1 Tracing framework . . . 35
3.2 CSW model . . . 37
3.3 Summary . . . 39

4 Tracing framework 41
4.1 Design . . . 41
4.2 Evaluation . . . 46
4.3 Summary . . . 48

5 Analysis and Instrumentation of TinyOS 49
5.1 Forwarding application . . . 49
5.2 TinyOS analysis . . . 51
5.3 Instrumentation . . . 54
5.4 Summary . . . 56

6 Model creation 57
6.1 Analysis of trace . . . 57
6.2 CSW model . . . 60
6.3 Summary . . . 63

7 Model evaluation **65**
7.1 Accuracy of CSW model 65
7.2 Scalability of CSW model 77
7.3 Impact of CSW model 81
7.4 Analysis 83
7.5 Summary 87

III Conclusions **89**

8 Conclusion **93**
8.1 CSW model 93
8.2 Tracing framework 97
8.3 Research limitations 97
8.4 Future work 98

References **99**

List of Figures

1.1 OSI reference model and the domain of the CSW model 11
1.2 Forwarding app topology 13
1.3 Outline of thesis 14

2.1 Illustration of WSN (http://vlabs.iitkgp.ernet.in/ant/8/theory/) 16
2.2 TPR2420CA Block Diagram [Mem] 18

3.1 How the tracing framework is used 36

4.1 Relationship between trace tuples and CSW events 42
4.2 Process of tracing a real mote 44

5.1 Forwarding app network stack 50
5.2 Summary of CSW flow when packets are forwarded 51
5.3 Overview of CSW flow when packet is forwarded 52
5.4 Overview of instrumentation in CSW 55

6.1 Relationship between trace ID, CSW events, signatures, SEMs and the CSW model 58
6.1 Format of trace tuples 58
6.2 Example of trace tuples 59
6.3 Example of trace with a timestamp error 59
6.4 Some output from trace analysis 59
6.5 Example of decompressing trace to CSW events 60
6.6 CSW events to signatures conversion 61
6.2 ns-3 simulation 61

7.1 Metrics used to evaluate the CSW model and their place in the packet forwarding process. 66
7.2 Intra-OS delay comparison between real mote and ns-3 with the CSW model at 40 pps. The intra-OS delay is in binary ms. 72
7.3 Intra-OS delay based on UDP payload size 74
7.4 Variation in the IPAQ fill-level affecting the intra-OS delay. The intra-OS delay is in binary ms. 75
7.5 The forwarding rate when sending packets at various packet rates 76
7.6 Time added to simulation execution time because of an active and inactive node 80
7.7 Simulation execution time and memory consumption when increasing the number of nodes 82

8.1 124-byte packets sent as fast as possible with initial backoff in TinyOS enabled, no packets are dropped. 95
8.2 Timer issue when emulating in Cooja/MSPSim. 96

List of Tables

1.1 Comparison of intra-OS processing delays in a real mote, Cooja/MSP-Sim, ns-3, and the radio transmission times 10

3.1 Functional (F) and non-functional (NF) requirements for the tracing framework (TF) and CSW model (CSWM) 39

4.1 Difference between buffered and instant tracing 48

7.1 Comparison of execution time when comparing a real mote, an emulated Cooja/MSPSim mote (extrapolated data) and the CSW model in ns-3. 79

8.1 Comparing use cases for the CSW model in ns-3, an emulated Cooja/M-SPSim mote and a real testbed. 96

Listings

4.1 Simple compression of CSW events 45
4.2 Compression of CSW events with two threads 45
4.3 (a) Compression of CSW events with many threads. (b) Compression of CSW events with many threads and a boolean state variable. 46
7.1 Equations explaining Figure 7.1 66
7.2 Equations explaining Figure 7.4 75
7.3 Output from Experiment 6 simulation runs 83
7.4 Projected intra-OS delay if packets that are ahead in the IPAQ must wait for new packets to be placed in it. Continuation of equations in Listing 7.2 85

Abstract

Emerging infrastructure-less network architectures such as WSNs consist of devices that perform packet processing in software. General-purpose network simulators do currently not possess models to simulate the intra-node delay of such devices. Consider a real TelosB mote that runs TinyOS which spends seven and fifteen ms on processing packets of size 36 and 124 bytes. Failing to simulate that results in inaccurate simulation of packet loss, jitter, and latency. In this thesis, we create a communication software model of TelosB to include its temporal behavior for more accurate WSN simulations in the ns-3 simulator. A challenge in doing so is to create a tracing framework for TinyOS that can be used to accurately and reliably trace its behavior. The evaluation of the model shows that it is accurate, scalable, and has a significant impact when including it in a simulation.

Acknowledgement

I want to thank my supervisors Thomas Peter Plagemann and Stein Kristiansen for their support and help with writing the thesis. Without their guidance, this thesis would not have been close to what it became.

I also want to thank my girlfriend for her support and encouragement throughout my studies. I am very grateful for her comments on my writing and help with proofreading the thesis.

Finally, thanks go to my family who is always there for me. I am profoundly grateful for their continuous support.

Abbreviations

TinyOS/TelosB The TinyOS operating system executing on a TelosB mote.

6LoWPAN IPv6 over Low-power Wireless Personal Area Networks.

CCA Clear Channel Assessment.

Cooja Contiki OS Java Simulator.

Cooja/MSPSim Emulation with MSPSim when using the Cooja simulator.

CPU Central Processing Unit.

CSW Communication Software.

CSWMM Communication Software Model Methodology.

DPC Deferred Procedure Call.

FSM Functional Service Model.

IPAQ IP Packet Queue.

LEU Logical Execution Unit.

MCU Microcontroller Unit.

ms Millisecond.

MSPSim MSP430 emulator.

ns-3 Network Simulator 3.

OS Operating System.

PEU Physical Execution Unit.

PHS Packet Handling Service.

PID Process ID.

RAM Random Access Memory.

RX Receive.

SEM Service Executable Model.

TOSSIM TinyOS Simulator.

TRx Transceiver.

TX Transmit.

WLAN Wireless Local Area Network.

WPAN Wireless Personal Area Network.

WSN Wireless Sensor Network.

WSS Work Scheduling Service.

μ**s** Microsecond.

Part I

Introduction & Background

Chapter 1

Introduction

Network simulators are often used for testing different types of networks. Users expect to simulate the most critical aspects of the network application they are testing, which mainly include the protocol behavior (e.g., routing behavior) and the temporal behavior of executing the protocols. The temporal behavior can be divided into two groups: (1) transmission delay when packets are sent in between the nodes and (2) intra-node delay when intermediate nodes process packets for forwarding. This thesis explores the part of the intra-node delay that is caused by the communication software (CSW). General-purpose network simulators differ from emulators and real testbeds in that they are more scalable and suitable for extensions, but also less accurate by for instance ignoring intra-node delay. Therefore, the focus of this thesis is to add such delay in a CSW model to use with an existing extension in the ns-3 simulator.

The more significant the intra-node delay is, the less accurate are the network simulators that only simulate transmission delay. An example of a network simulation that illustrates this problem is structured as a linear chain topology consisting of three nodes A, B, and C. Node A can only see B, but it wants to send a kilobyte packet to C. Therefore, it sends it to B that forwards it to C. If the data-rate of the transceivers is a kilobyte per second, the transmission delay in the simulation is two seconds. Since the general-purpose network simulator can only simulate transmission delay, the end-to-end delay is two seconds. In a real testbed scenario, however, the end-to-end delay might be as much as ten seconds. Of those ten seconds, only two seconds is transmission delay and eight seconds is the intra-node delay. In that case, the general-purpose network simulator is inaccurate because it ignores a significant delay that is observed in a real scenario.

While intra-node delay has been insignificant in many cases because the Internet uses specialized routers for processing packets in hardware quickly, infrastructure-less networks such as WSNs contain resource-constrained devices that often perform packet processing in software. Sensor devices act as both routers and endpoints, which means they forward packets amongst neighbor nodes in the network as well as performing other tasks. The intra-node delay of such an intermediate node usually includes the time it takes for the CSW to write a packet into RAM, send it to upper layer protocols, check if a route exists to the destination, write packet into the radio chip's TX queue, and then send it. Simulating such behavior is complicated because it requires knowledge about the devices and the CSW of the Operating System (OS) they run.

1.1 Thesis statement

The task of this thesis is to evaluate the overall significance of the temporal behavior of protocol handling in a TelosB mote running TinyOS that is used in WSNs and create a CSW model it in ns-3. As the model is used in conjunction with an existing transceiver model, both transmission delay and intra-node delay are added to the simulation. More specifically, the part of the intra-node delay that is caused by CSW, intra-OS delay, is added to the simulation. From now on, $^{TinyOS}/_{TelosB}$ refers to a TelosB (MTM-CM5000MSP) mote executing TinyOS.

Using general-purpose network simulators to simulate forwarding nodes in WSNs should be accompanied by CSW models for simulating the delay caused by intermediate devices, because of the significant delay they add to the total end-to-end delay. The focus of this thesis is to create such a model of a WSN device to be used in ns-3. The data in Table 1.1 is used to back up that claim. It describes the amount of time it takes for an intermediate mote to process packets of various sizes before forwarding them when using a real mote, an emulated Cooja/MSPSim mote, and an ns-3 mote without a CSW model. The results show that the intra-node delay is substantial for two reasons. First, the delay is much higher for the real mote than the transmission delay. Second, the variation in the delay is high when varying the packet size. Since ns-3 ignores the intra-node delay, it is highly inaccurate in simulating the temporal behavior. An emulated Cooja/MSPSim mote is not as accurate as a real mote and the difference increases as the packet size increases. That indicates either that Cooja/MSPSim fails to accurately simulate processing stages in which the execution time is dependent on the packet size, or the hardware model that MSPSim emulates is different from our MTM-CM5000MSP. While a transceiver model simulates the transmission delay, a CSW model adds the intra-OS delay to a simulation, which is the vast majority of the full intra-node delay of an intermediate node that processes packets in software. Therefore, using a CSW model can provide an accurate simulation of end-to-end delay.

Packet size	Send time	Cooja/MSPSim	Real mote	ns-3
36 bytes	1334 μs ~1.3 ms	5.65 ms	7.1 ms	0 ms
57 bytes	1980 μs ~2 ms	6.9 ms	9.2 ms	0 ms
76 bytes	2569 μs ~2.5 ms	7.9 ms	10.7 ms	0 ms
120 bytes	3964 μs ~4 ms	10 ms	14.8 ms	0 ms

Table 1.1: Comparison of intra-OS processing delays in a real mote, Cooja/MSPSim, ns-3, and the radio transmission times

1.2 Communication software (CSW) model

In this thesis, the communication software model methodology (CSWMM) defined in [Ste13] by Kristiansen et al. is used to create a CSW model for a device used in WSNs called TelosB. The methodology defines a step-by-step approach to do this, all of which are executed in this thesis. In short, the device to model is traced by having an instrumented CSW forward packets in different contexts at a low packet

rate. Afterward, the trace is used to create the CSW model. The primary challenge in applying the methodology to *TinyOS*/*TelosB* is to acquire the traces from it because TelosB is a resource-constrained device with only 10kB RAM in total. The CSWMM has been used before, but not with anything similar to TelosB or TinyOS. Therefore, the first task in this thesis is to create a tracing framework that enables us to understand TinyOS and TelosB, and the second task is to use traces to generate a CSW model that can be used in ns-3.

CSW is the part of the OS that processes packets in some way. That is usually the code that represents the layers in the OSI reference model, displayed in Figure 1.1. The CSW model created in this thesis focuses only on the bottom three layers: the physical layer, data-link layer, and the forwarding part of the network layer (excluding the routing part). The reason for only modeling some of the layers is that the model is of an intermediate node that forwards packets between nodes in the network by only involving those three layers. The network layer includes routing and forwarding of packets, and only the forwarding part is included since a route is assumed to exist. In the future, the other layers can also be modeled for more accurate simulation. For instance, if nodes use a routing protocol that requires the nodes to periodically refresh routes to nodes, the temporal behavior of the protocols that calculate routes can be modeled. Nevertheless, all the CSW related to forwarding of packets by intermediate nodes is included in the current CSW model.

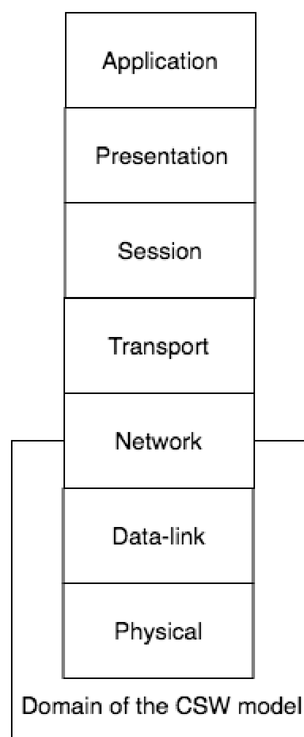


Figure 1.1: OSI reference model and the domain of the CSW model

The CSW model enables accurate simulation of packet delay, packet loss, and jitter. Packet delay is simulated because the CSW is split into several processing stages that take time to execute. Packet loss occurs when queues are full, which happens when the packet rate is too high for the CSW to process all incoming packets. Packets might be dropped at different places in the CSW, and part of the task is to find which contexts it happens. Jitter is the variable packet delay incurred by sending packets multiple times

from the source node to the destination node. For instance, if packets are sent exactly every two seconds, but are received by the destination every 1.9–2.1 seconds, the 0.2 seconds difference is due to jitter. We conduct experiments that compare the model's behavior with a real device to assess its accuracy.

The steps of the methodology to create a CSW model are:

1. instrumenting the CSW,
2. running the device in the most important contexts to generate traces,
3. human investigation of the traces and signatures that are generated from the traces, and
4. creation of a final CSW model.

Step 1 requires an understanding of the chosen OS, which is TinyOS in our case. As the CSWMM has previously been used to create three CSW models, Step 3 and 4 are relatively simple. The challenge is with Step 1 and 2 because TelosB offers little storage capacity, which makes it hard to store traces on the device. When Step 1 and 2 are accomplished, the remaining steps are straightforward since they have performed three times before. Step 3 is about verifying that the resulting trace or signatures are correct. Step 4 is about the creation of the final CSW model.

1.3 Aim of thesis

This thesis aims to assess the significance of software execution delay in devices in WSN, apply the CSWMM to create a CSW model of *TinyOS*/*TelosB* for a general-purpose network simulator (ns-3), and evaluate it. Two separate tasks need to be accomplished to do that: create a tracing framework be able to trace the CSW of TinyOS and create the CSW model in ns-3 that uses the captured trace data. This thesis succeeds if the resulting CSW model has similar packet forwarding behavior as a real *TinyOS*/*TelosB*. The scalability and impact of the model are also evaluated to see if large-scale simulations are possible and if the model is necessary in the first place. The impact is, however, demonstrated throughout this thesis in various ways, including in Table 1.1 by looking at how inaccurate a simulation is without simulating the intra-node delay.

Following is a list of the goals of this thesis:

- Demonstrate that CSW models are needed in general-purpose network simulators such as ns-3 to simulate the temporal behavior of WSNs accurately.
- Create a reliable and efficient tracing framework for *TinyOS*/*TelosB*. That is explored in Chapter 4 when the tracing framework is described and evaluated.
- Create a forwarding application in TinyOS and ns-3. The TinyOS application is executed on the intermediate mote to generate traces. The ns-3 simulation utilizes the CSW model. Traces from the TinyOS application are compared with traces from the ns-3 simulation with the CSW model to evaluate it. The more similar they are in packet delay and packet loss, the better.
- Create a CSW model that adds realistic temporal behavior of the TelosB mote. In Chapter 6, we describe the creation of the CSW model, and it is later evaluated in Chapter 7 by conducting experiments and analyzing the results.

Figure 1.2 illustrates the linear chain topology of the forwarding application used in this thesis. Mote A can see Mote B, and Mote B can see Mote C. Mote A sends a packet aimed at Mote C and sees Mote B as an intermediate mote. The CSW model simulates the behavior of the intermediate Mote B by adding extra delay to the simulation, which causes the end-to-end delay to increase and packet loss at high packet rates. Throughout this thesis, Mote A, Mote B, and Mote C refer to the motes in that topology.

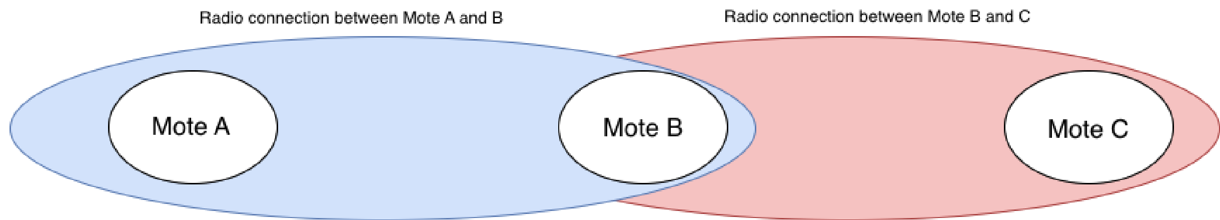


Figure 1.2: Forwarding app topology

1.4 Outline

The thesis is structured as in Figure 1.3. In this chapter (Chapter 1), an introduction is given to the problem statement, and in the next one (Chapter 2), the background knowledge required to understand the thesis is provided. In Chapter 3, the requirements we have to the tracing framework and CSW model are defined. In Chapter 4, the design, and evaluation of the tracing framework are provided. In Chapter 5, we explain the forwarding application, analyze the forwarding process of TinyOS and describe the instrumentation of it. In Chapter 6, we describe how the CSW model is created in ns-3 by using the captured traces, and in Chapter 7, the model is evaluated. In Chapter 8, we summarize everything that is learned in this thesis.

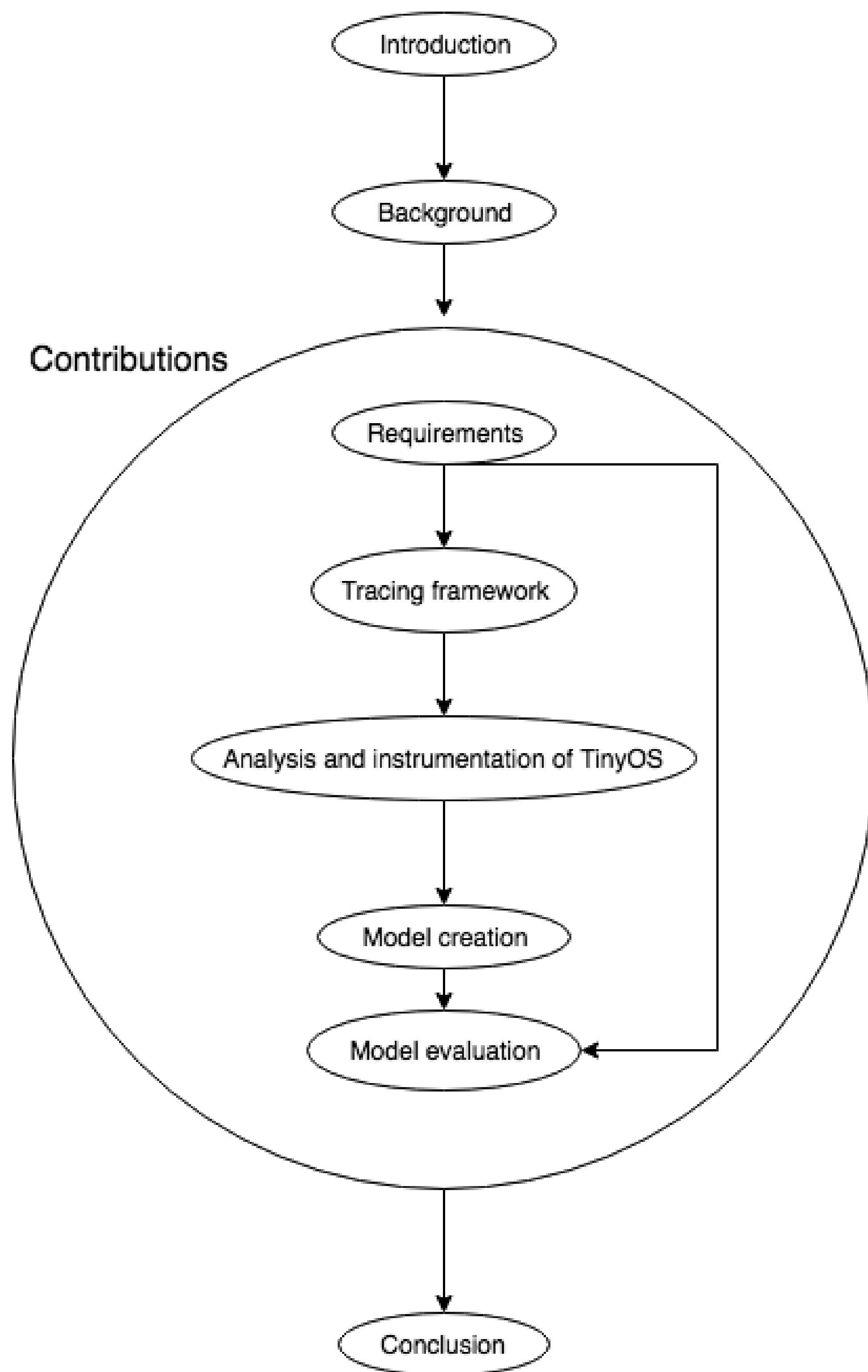


Figure 1.3: Outline of thesis

Chapter 2

Background

In this chapter, all the necessary knowledge required to understand the contributions of this thesis is explained. The main contribution is a *CSW model* of the *WSN* device *TinyOS*/*TelosB* to provide realistic temporal behavior in the *general-purpose network simulator ns-3*. Creating the CSW model requires a *tracing framework* to trace the modeled device in the proper contexts. Section 2.1 starts the chapter with an introduction to WSNs, TinyOS, and TelosB. Section 2.2 continues with explaining what simulation is, different types of network simulators and the difference between a real testbed, emulation, and general-purpose network simulation. It is useful to understand that different types of simulation require different levels of effort. Section 2.3 explains the CSWMM which is used to create the CSW model, and is imperative to understand. Section 2.4 discusses related works concerning the tracing framework needed for the CSW model, and processing delay models, of which the CSWMM is one methodology.

2.1 Wireless Sensor Network (WSN)

A WSN is a network that consists of wireless sensor nodes (motes) that collect useful information for various applications [YMG08] [F.] [HHKK04]. Figure 2.1 illustrates an example of a WSN. Motes are placed in abundance in an area to sense the environment. They periodically send sensor data to a sink device which connects to a PC that handles the data. The PC may also connect the WSN to the Internet. When motes send data to the sink, they usually have to send the data via intermediate motes in the network, unless they are close to the sink. Encapsulating the temporal behavior of the CSW of such intermediate motes in a CSW model is the focus of this thesis.

The literature describes several application areas for the WSN that can be interesting to simulate in ns-3. Examples of them include monitoring of the environment for hazardous events [HM06], health monitoring of patients [Ste17], military applications [DTDM12], automated traffic control systems [Ras13], agricultural applications [PE08], medical applications [NFR08], motion tracking of people [ZH08], and more. Many useful application areas for WSNs are yet to be deployed in real scenarios. Therefore, high-level simulation in a general-purpose network simulator is a good first step in the direction of deploying a WSN application. The temporal behavior must, however, be simulated sufficiently accurately that the results are convincing.

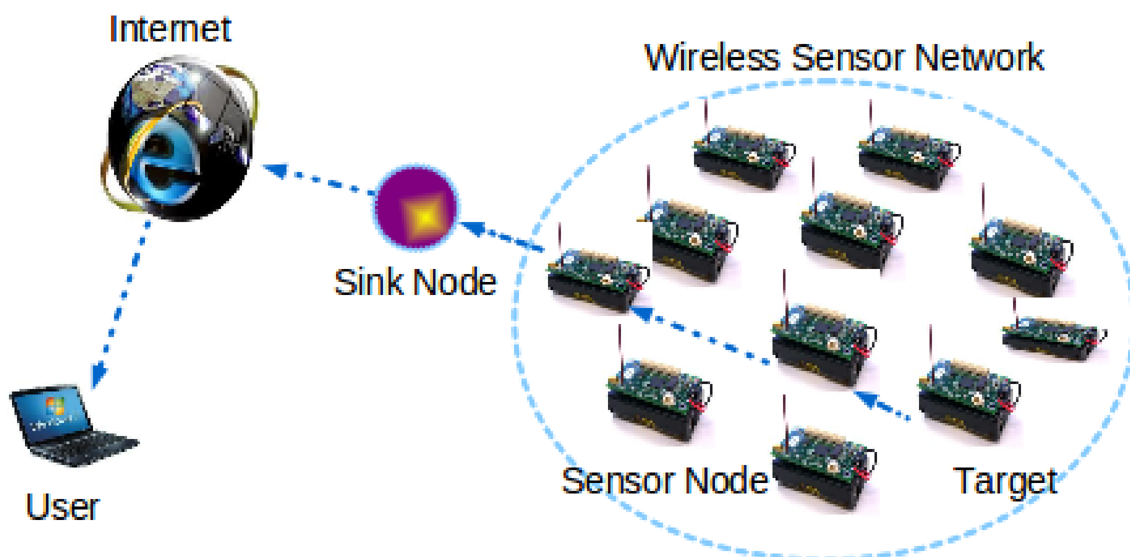


Figure 2.1: Illustration of WSN (http://vlabs.iitkgp.ernet.in/ant/8/theory/)

2.1.1 IEEE 802.15.4

IEEE 802.15.4 is a standard that defines the PHY and MAC sublayer of communication between low-cost, low-power devices in Wireless Personal Area Networks (WPANs) [iee12]. Most WSN devices, including TelosB, are based on it. In contrast to WPANs, Wireless Local Area Networks (WLANs) are usually based on IEEE 802.11 and are the types of networks that include regular computers and smartphones. WPANs can be as big as WLANs but differ in that they have little or no infrastructure, which is why the nodes in the network must forward packets for each other. Another thing that distinguishes WPANs from WLANs is that the data-rate which WPAN devices communicate with each other at ranges between 20–250kbit [iee12] and WLAN devices can communicate at a thousand times that speed. The Maximum Transmission Unit (MTU) is 127 bytes in IEEE 802.15.4, which is much less than the 1280 bytes MTU for IPv6 [six14].

As a result of the low data-rate and MTU defined by IEEE 802.15.4, devices must choose communication protocols accordingly. Despite the IPv6 MTU being much bigger than the IEEE 802.15.4 MTU, IPv6 is still used in WPANs, but with a twist. Several solutions such as 6LoWPAN and Zigbee attempt ways at accommodating the IEEE 802.15.4 requirements for WPANs to use existing communication protocols and connect them to the Internet. 6LoWPAN, or IPv6 over Low-power Wireless Personal Area Networks, introduces an adaptation layer that enables IPv6 packets to be sent over WPAN networks [six14]. It fragments packets greater than the IEEE 802.15.4 MTU and compresses IPv6 and UDP/TCP headers to consume less memory. 6LoWPAN is excluded from the current CSW model because it has little effect on the forwarding scenario: fragmentation of packets is discouraged [six14] and a variable packet header size only means a few additional or fewer bytes to a packet. Integrating the CSW model with 6LoWPAN can be considered in future work.

2.1.2 Mote

Devices in WSNs are small sensor computers, also called motes, that are characterized by their resource-constrained components. Throughout this thesis, the words device, node, sensor computer, and mote are used interchangeably to describe devices in WSNs. This thesis focuses on a mote called TelosB which runs a 4MHz CPU and has 10kB of Random Access Memory (RAM). On the other hand, a modern PC can have 4GHz CPU and 16GB of RAM, which means it runs a thousand times faster and can save 1,600,000 more data in memory than TelosB. As a consequence of its little RAM capacity, capturing the behavior of TelosB is a challenge.

A mote is usually a composition of multiple components that are placed on a circuit board. Most of them consist of the same types of components. These include a microcontroller unit (MCU) that runs the hardware instructions of the application and OS. A mote also contains a radio chip, which defines the physical layer of the network communication for the mote and performs the transmission and reception of packets. Furthermore, they usually have a small amount of RAM available for volatile memory storage, and a flash chip for non-volatile memory to be able to store data that cannot fit in RAM. Serial communication is commonly used for message sending between PC and sensor device, which is the case for TelosB. Motes often use AA batteries for power supply or get it from the USB connection. The final and most essential components of a mote are its sensors. A mote has a set of sensors and a way of converting the analog signal to digital through an ADC (Analogue to Digital Converter).

TelosB (MTM-CM5000MSP)

We choose to work with TelosB in this thesis. The mote is also called Tmote Sky, but the specific hardware number is MTM-CM5000MSP. It is an ultra-low-power wireless sensor module developed at UC Berkeley [PSC05] [Mem]. The reason why this device is chosen is that it is the only device that can be emulated with TinyOS in Cooja/MSPSim. An alternative device is MICAz, which can be simulated with TOSSIM. Since both ns-3 and TOSSIM perform discrete-event simulation, however, hardware instruction simulation with Cooja/MSPSim is more interesting.

Specification The mote consists of a set of chips on top of a board, each of which has its qualities and specifications. Figure 2.2 illustrates the components on TelosB. Following is a list of the main components and specifications that are related to the packet forwarding of the mote:

- MCU: MSP430, 4MHz effective CPU frequency, max baud rate of serial (UART - RS232) communication: 115200.

- Radio chip: CC2420, max transmission bit-rate: 250kbit per second. It has an RX and TX queue each with a capacity of 128 bytes. Has a CCA feature which avoids sending packets when the channel is not clear.

- RAM: 10kB.

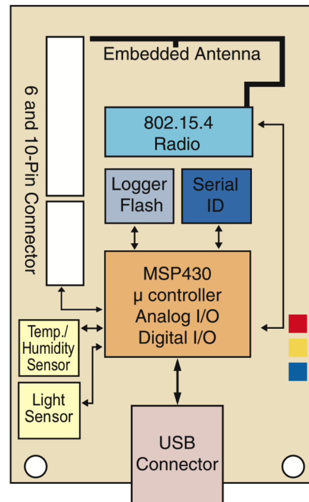


Figure 2.2: TPR2420CA Block Diagram [Mem]

CC2420 radio chip

CC2420 is the radio chip used by TelosB and its driver in TinyOS is the most important part of the CSW when working with CC2420. The chip has a 128-byte RX queue and TX queue for receiving and transmitting packets. If the radio receives more than 128 bytes, an overflow occurs. When the radio is in an overflow state, no packets can be received, and the RX queue must be flushed before new data can be received. That is done within the CC2420 driver and is among the places we instrument. Other places include when a new packet is being written into RAM, finished being written to RAM, finished being written into the TX queue, and more places. All tracepoints we place in the CC2420 driver and other places in TinyOS are discussed in detail in Chapter 5.

Clear channel assessment (CCA) TinyOS and CC2420 both offer a CCA feature for preventing collisions and starvation. TinyOS implements it as a backoff timer that causes the node to wait a random number of milliseconds before sending a packet. The backoff is there to prevent collisions and too long occupation of the channel. One backoff timer is used before the first time a packet is sent, a different one when the channel is not clear. Both the TinyOS and CC2420 CCA features are enabled when the variable ccaOn is set to true in the CC2420 driver. ccaOn is enabled by default, which means that the CSW backs off for a random amount of milliseconds before it sends each packet, even though there is no reason to believe there is network traffic. Since it adds undesired random delay, the TinyOS CCA backoff feature is turned off in all experiments when evaluating the CSW model.

On the other hand, the CCA feature in CC2420 is enabled for our experiments. CC2420 implements CCA as a backoff when the channel is not clear [Davb]. Thus, the radio only sends the packet when the channel is clear, a decision made based on a configurable setting. In [Tex14], the different settings of CC2420 deciding when the channel is clear are listed: (1) when the received signal strength is above a certain threshold, (2) when not receiving valid IEEE 802.15.4 data, and (3) is both setting 1 and 2 combined. If the feature is kept enabled when the packet rate is high, it prevents collisions from occurring. The CCA settings used for the experiments in Chapter 7 are

the default for TinyOS v2.1.2.1.

MSP430F1611 MCU

TelosB uses MSP430F1611, or just MSP430, a popular low-power MCU with a 16-bit CPU. The CPU has a maximum speed of 8MHz, but TelosB only utilizes 4MHz. Although the MSP430 driver is not instrumented, it is still a significant component in regards to the temporal behavior of packet processing since it executes the CSW. The low speed of the MCU (4MHz) makes it an appealing target for the CSWMM because the impact of the model is likely to be more significant compared to fast clock rates. Processing delays within the CSW model are defined in CPU cycles. As the CPU speed and size of queues in the model are configurable, a way to use the CSW model is to tweak those variables to see if the behavior improves, worsens or remains unchanged. That way, developers might find that they can manage with 1MHz CPUs instead of 4MHz, for instance. Changing the CPU frequency is not trivial to do when using an emulator, and especially not on a real mote. If the CPU frequency is turned to a high number, the CSW model will have little effect on the simulation, and it can be used to test the limits of the simulated nodes. To sum up, MSP430 is a key component in the CSW because it contains the CPU, but the MSP430 driver is not instrumented.

TinyOS uses the timers on the MCU to get software timers with a precision of 1MHz, 32KHz or 1KHz. Microsecond (1MHz) precision is used for the timestamps when performing tracing. An explicit CPU cycle counter would be a better alternative, but MSP430 does not offer it. At microsecond precision, the clock operates at 1/4 of the MSP430 CPU frequency, which means the microsecond clock is a good alternative for capturing the CPU cycle counter. One thing to note about the microsecond timer is that it is not completely accurate, and therefore we must be able to exclude traces that seem to have inaccurate timestamps. Trace analysis tools are used for filtering out inaccurate parts of traces.

Main memory As mentioned previously, TelosB has a total amount of just 10kB RAM, which is a tiny amount compared to a modern computer or smartphone, which often possesses more than 1GB RAM. 1GB is 100,000 times more than 10kB. That poses a problem for the tracing step of the CSWMM since RAM is usually the storage option that results in the least tracing delay, which is an essential non-functional requirement for the tracing framework. Therefore, the memory consumption of the trace events must be minimized.

2.1.3 TinyOS

TinyOS is one out of many OSs that are used in WSNs including in this thesis. Other examples of OSs include MANTIS, Contiki OS, Nano-RK and LiteOS [FK11] [RSM14]. TinyOS is minimalistic and can be executed on resource-constrained devices. Furthermore, it is event-driven, has a RAM footprint of only 400 bytes, is single-threaded, and there is no scheduler preemption [LMP+05]. Additionally, heap memory does not exist, which means all data is stored statically with no temporary memory allocation. TinyOS 2.x is the OS we choose to model, and throughout this thesis, TinyOS refers to TinyOS version 2.1.2.1 unless otherwise specified.

TinyOS is compiled for one application at a time, and that application has full access to all hardware resources. There is no user or supervisor mode and no virtual memory either. No virtual memory means that all addresses point to physical memory. As such, the OS is significantly different from Windows, Linux, and Mac OS X since they offer multi-threading and support for multi-core CPUs. TinyOS does offer a library called tosthreads that enables some high-level multi-threading features described in [KLP+09], but it is not used in this thesis. The consequence of being single-threaded is that concurrency in TinyOS is done in an event-driven way by designing the application such that functions do not last for too long, which means it is the responsibility of the application developer to make sure the OS and application work smoothly.

The event-driven model in TinyOS makes it challenging to write applications, a result of TinyOS being single-threaded. When the OS performs its routine operations such as handling packets that are received and buttons being clicked, hardware interrupt events are invoked. They often post longer lasting tasks to fulfill their primary purpose. Since tasks cannot preempt each other and especially not hardware interrupt events, the application developer must write short lasting hardware interrupt events and tasks for the OS to work correctly. Thus, applications in TinyOS often use timers to invoke events every x milliseconds, and they post finite lasting tasks. The results is a reactive and event-driven execution model.

Timers in TinyOS are all displayed in binary time. That means one second contains 1024 binary milliseconds, 32768 32kHz ticks, or 1,048,576 microseconds [Corb]. Therefore, the binary time is sometimes used in this thesis instead of decimal. The impact of this difference is small when the measured time is small, but since the intra-OS delay of a mote ranges from 7500–15000 μs as in Table 1.1, the binary time is 7680–15360, which is significantly different. It is mentioned in figures when the displayed time is in binary instead of decimal.

nesC

nesC is the programming language that TinyOS and its applications are written in [GLvB+03]. It is a subset of the C programming language with some extra features. There are three new types of functions as compared to C: events, commands, and tasks [Dava]. Events can be viewed as software/hardware interrupts or callback functions, commands as regular functions and tasks as deferred procedure calls (DPCs). Hardware interrupt events are the only functions that preempt in TinyOS. Receiving a packet, turning on the mote and a timer going off result in events being signaled. Otherwise, everything runs until completion. nesC code is compiled to C code, which is compiled to binary code.

Tasks are DPCs that other commands, events or tasks *post* (defer), and are executed sometime later by the task scheduler [Cora]. The task scheduler executes tasks from a queue within an infinite loop. When the queue is empty, the mote goes to sleep to preserve power. A task that is posted several times before addressed by the scheduler is only executed once. If a task needs to be executed multiple times, it can post itself. Since a task requires a scheduler to execute them and thus depends on TinyOS to work, it differs from the other nesC language constructs that can technically be implemented as an extension to the C programming language.

Events and commands are invoked through something called an interface that

defines what program modules provide and use. If a module provides an interface, it must implement all the commands that are defined in it. If a module uses an interface, it must implement all the events in it. A typical use case for these types of functions is that a command is called to perform some action, which posts a task to be executed later, and an event is signaled when the task is finished executing. That is called split-phase operation. In this example, the event functions as a callback function for the command. Nevertheless, commands do not have to post tasks, and tasks do not have to signal events; any of the three can signal events, call commands and post tasks.

Since the interface functions as an abstraction between components, an additional component is needed to decide which events are signaled and commands are called. Those components are called configurations, and wire other components such that the correct functions are called when a command or event is invoked. For instance, calling a command that another component provides results in the correct command to be called. During the compilation process, it is verified that all interfaces are wired to components that provide or use the right commands or events. Compilation fails if some interface of a module is not wired to another module, or that module does not implement the correct functions.

2.2 WSN simulators

There are three ways of testing a WSN network application: general-purpose network simulation, emulation, and using a real testbed. We use the general-purpose network simulator ns-3 that simulates the network-layer, similar to OMNET++, ns-2, and more. General-purpose network simulation is an abstract way of running an application that usually uses discrete-event simulation. Testbed includes real devices running the application and is the most accurate way of testing a network scenario, but also a cumbersome way due to scalability and repeatability issues. Emulation is something in between testbed and general-purpose simulation in which some aspects such as hardware instructions of a hardware device are executed, but other aspects are simulated using discrete-event simulation.

Discrete-event simulation involves splitting tasks into several events that are scheduled to execute at some point in the simulation. When an event is finished executing, the simulator picks the next event that is to be executed at the earliest time and runs it. Simulation time only advances when the scheduler executes the next event, not during the execution of them. Each event has a timestamp attached to it. The scheduler selects the event with the earliest timestamp, sets the current simulation time to the timestamp and executes the event. Since discrete-event simulation is abstract and high-level, it benefits from being efficient, scalable and repeatable.

The scalability of emulation is better than with a real testbed, but not as well as only with discrete-event simulation. Discrete-event simulation and emulation can both be used to repeat simulations. It might be more difficult, however, to repeat a simulation with an emulator because the parameters that control the simulation flow are more intricate. A significant benefit of using emulators is that it can be used for debugging of applications, which general-purpose networks simulation cannot and is hard to do with real devices because of scalability and repeatability reasons. Still, the question of how scalable and accurate is a simulator depends on more than just whether it is an emulator or discrete-event simulator.

That is because many different simulators exist that are used to simulate WSNs. They include ns-2, ns-3 [RH10], TOSSIM [LLWC03], GLoMoSim [ZBG98], UWSim [PPFS12], Avrora [TLP05], SENS, Cooja [ODE+06, Ös06], Castalia, Shawn, Prowler, EmStar, Opnet, MSPSim, OMNET++, ATEMU [PBM+04] and J-Sim [SCH+05] [SKLD+11] [KSKT09] [MZ12]. Examples of emulators are Avrora, MSPSim, and ATEMU, that execute the code of a corresponding MCU. ns-2, ns-3, and OMNET++ are general-purpose network simulators that only focus on simulating the network communication as discrete-event simulators, and disregard the OS and hardware instruction layers. TOSSIM is a discrete-event simulator that simulates the OS-layer of TinyOS by splitting the code into events and executing them. In summary, there are many different simulators because they simulate different layers, are suitable for different application areas and have different levels of accuracy and scalability.

2.2.1 ns-3

ns-3 is a discrete-event general-purpose network simulator that we extend with a CSW model. It is used mainly for educational and research purposes and is meant to be an easily accessible and extensible platform for testing out network protocols. The goal of ns-3 is to have a community of researchers that can contribute to extending the simulator and adding models of whatever technologies that need to be simulated, for instance, WiFi, WiMAX, and LTE. These are complex technologies that one does not want to code the behavior of in every new project. The models provide realistic protocol and temporal behaviors of the technologies they represent, but the existing models only focus on the transmission delay.

Since Table 1.1 shows that the intra-node delay is significant, it is an indication that intra-node or CSW models are needed. Furthermore, [KSKT09, Mic05] explain that one of the problems with ns-2, and by extension ns-3, is that the simulator does not simulate the application code execution delay. Our model simulates the most important part of that delay in a networking scenario, which is the delay caused by the CSW of TinyOS. Thus, the model created for this thesis is an example of a model that can be added to ns-3.

2.2.2 Cooja

Cooja is a simulator designed for Contiki OS, but also runs the code of other OSs such as TinyOS [ODE+06] [Ös06]. It simulates the network layer, OS layer, and uses Avrora [TLP05] or MSPSim [Joa] to simulate the hardware instruction layer to achieve cross-layer simulation [ODE+06]. Where TOSSIM simulates the OS layer, ns-3 simulates the network layer, and MSPSim simulates the hardware instructions of MSP430, Cooja can do all three. That Cooja can simulate the network and hardware instruction layer at once makes it the simulator that is closest to running on a real device and is the way we use to debug and test the forwarding application described in this thesis.

In our case, Cooja executes the MSPSim emulator when running the forwarding application in TinyOS. Therefore, when we write that data is collected from an emulated Cooja/MSPSim mote, MSPSim is executing the hardware instructions. That is relevant because results from preliminary experiments in Table 1.1 indicate that some processing stages might take too little time to execute. Then it is highly likely that MSPSim is causing that. We also discuss in Chapter 8 Section 8.3 the possibility that

MSPSim emulates slightly different hardware, but with the same components as our MTM-CM5000MSP/TelosB/Tmote Sky mote.

Plugins in the simulator enable the user to control and monitor the simulation. By using a built-in plugin, the user can place as many nodes as needed on a map and move them around. As each mote has a TX/RX range that can be adjusted, we know for sure whether a node is in range to receive packets or not. Arrows on the map indicate which mote is receiving packets from where and the Radio Messages plugin can be used to read their contents. We use the map to set up the topology for the forwarding application, in which Mote A can see B but not C, and Mote C can see B but not A. The Mote Output plugin outputs the serial communication from the motes, which we use both when debugging the forwarding application and when tracing the intermediate Mote B.

Cooja is similar to ns-3 in some ways. They both offer general-purpose simulation, but this thesis focuses on the emulation part of Cooja. Both simulators can be used to simulate WSNs, although Cooja is specially made for that purpose. They differ in that Cooja can use emulators that simulate the CPU accurately, and ns-3 focuses mainly on the network portion of the devices. As the CSW model we create adds the temporal behavior of intermediate nodes in communication scenarios, it will be fascinating to compare results from a real mote with an emulated Cooja/MSPSim mote to see if results from running ns-3 with CSW models are more or less accurate than Cooja/MSPSim. If the results are similarly accurate, it can make ns-3 a viable alternative to the more complex emulation with Cooja/MSPSim.

2.2.3 TOSSIM

TOSSIM is the TinyOS simulator, an accurate and efficient discrete-event network simulator that runs code of a TinyOS application by splitting it into events [LLWC03]. TOSSIM is often used as a debugger for TinyOS software, but can currently only be used to run software for the MICAz mote. What differs ns-3 from TOSSIM is that ns-3 is primarily concerned with simulating network communication, whereas TOSSIM focuses on the OS part. The CSW model we create for ns-3 adds the temporal behavior of the simulated CSW, whereas TOSSIM runs the code and simulates at a deeper level, but not as deep as with emulation. TOSSIM differs from emulation with Cooja/MSPSim in the sense that it does not offer cycle-accurate simulation, but rather uses discrete-event simulation to progress time for each simulation event that is executed. There are, however, extensions that offer more accurate timing to simulation in TOSSIM [Muh07] [HC09b] [LAW08]. As a result of executing TinyOS code, but with discrete-event simulation, TOSSIM provides an accurate and efficient way of simulating WSNs.

The simulator has previously had problems because it is tightly coupled with TinyOS 1.x, which causes TOSSIM not to work so well with TinyOS 2.x as with TinyOS 1.x. [ZN11] describes how Mo Zhu et al. fixed the CC2420 support for TOSSIM because of a new version of TinyOS broke the functionality. Additionally, the IP library used in this thesis (BLIP) is currently not working well with TOSSIM. The simulator currently only works with the MICAz platform, but the forwarding application used in this thesis is designed for TelosB. Therefore, the forwarding application does not work with TOSSIM. Making the forwarding application work with MICAz is not trivial because it uses a different MCU and radio chip from TelosB, and therefore the drivers

are different. Consequently, we cannot include results from running TOSSIM when assessing the accuracy of the CSW model.

2.2.4 Discussion

Simulation with TOSSIM and emulation with Cooja/MSPSim results in accurate simulation, but that accuracy makes TOSSIM more dependent on TinyOS and Cooja more dependent on emulation software when compared to ns-3. If a new type of MCU comes out, Cooja would have to use a new emulator specifically for that MCU to be able to simulate applications compiled for it. If a new TinyOS version comes out, TOSSIM is in the danger of not working correctly due to its tight coupling with TinyOS. ns-3 will most likely not change much in the next few decades and does not depend on any components to perform its simulation. Moreover, models from earlier versions of ns- can be implemented in newer versions with less effort, as when the CC2420 transceiver model is implemented in ns-3 with an existing model in ns-2 as starting point [IG]. Therefore, ns-3 being extensible and open source are points in favor of ns-3 over simulators that are less extensible or not at all.

2.3 Communication software model methodology (CSWMM)

The communication software model methodology (CSWMM) proposed in [Ste13] by Kristiansen et al. describes all the steps required to make models of the CSW of any given OS. CSW models are created by tracing the CSW of a real device to capture its temporal behavior. Using traces from real executions, they add realistic temporal behavior of intermediate nodes to a general-purpose network simulator. The methodology consists of two parts which are performed only once by programmers who know the system. The first part is to make models from the CSW of the device one wants to simulate. The second part is to make an extension for the network simulator one wants to use, which serves as the execution environment for the models. The benefit of separating the tasks of creating CSW models and creating simulator extensions is that the same CSW models work with different network simulators and vice versa, with some need for porting. This thesis focuses on the first part: making a CSW model of $^{TinyOS}/_{TelosB}$. When a network simulator is extended with such a model, it should add approximately the same delay as the service it corresponds to on a real device.

The six core concepts from [KPG13b] are briefly summarized to give the reader an overview of the methodology:

- Physical Execution Unit (PEU). Specialized hardware units which run in parallel with the CPU, such as Direct Memory Access chip (DMA) and Network Interface Card (NIC).
- Logical Execution Unit (LEU). Construct for interrupts and threads to share the CPU with their memory. These run separately with their memory space, but do not run in parallel.
- Packet Handling Services (PHS). Functions which act as entry points to protocol services. They use packet queues to decide which packet is currently in processing.

- Work Scheduling Services (WSS). Functions which deal with scheduling work for services by using service queues. A WSS schedules services within an LEU by running functions that are hard-coded in the WSS or by dequeuing functions from a function queue.

- Services. PHSs and WSSs are types of services. While an OS has many different types of services, only the CSW services that handle packets or deal with work schedule matter in this context.

- Behavior, set of instructions executed in a service given a context. Different contexts cause different behaviors, and we want to find out when and how often certain behaviors occur.

- Context, set of state values for all state variables that affect the behavior of a service. At this abstraction layer, not all sets of state values for state variables in services are important, only the ones that have a direct effect on the temporal behavior of the service. An example of a set of state values for a state variable such as a queue is non-empty and empty.

We want to map the temporal behavior of each service in the CSW, in all the relevant contexts we can find, to a signature. The first step is to instrument the CSW by marking all the events in the services that are of importance to the temporal behavior of the packet forwarding. Such events can be divided into six classes [KPG13b]:

- class 1: separating services,

- class 2: CPU processing delays,

- class 3: interactions with the task scheduler,

- class 4: work scheduling within LEU,

- class 5: parallel processing, and

- class 6: execution context.

These classes use the abstractions defined above to describe the domain for the events they contain. A class 4 event can, for instance, describe starting a service within a WSS. Class 6 events are where we capture context data for the service and thus have a direct impact on the behavior of a service. A tracing framework is required to capture these events when executing a CSW, and one of the main tasks in this thesis is to create one. The tracing events must be defined such that the events can capture the CSW behavior sufficiently.

The process of making the model is divided into four main steps. They are (1) instrumentation of the code, (2) execution of the instrumented CSW and generation of traces, (3) execution of automatic analysis on the trace to get the signatures, and (4) create the final CSW model. All the following chapters are about these steps in one way or another.

2.3.1 Instrumentation

Instrumentation of the CSW first requires analysis of it to find all the services that can be involved when doing packet forwarding. These services include context switch functions, packet receive/send functions, context switch functions, and more. Afterward, the code is instrumented by putting tracepoints in the places the events defined above occur. Doing this requires a deep understanding of the services in the CSW. After this step is finished, two things can force us to repeat it: either not enough events are included of the event types that are already defined, or there is a new type of event that needs to be included in the tracing framework.

2.3.2 Tracing

Tracing is the step where the device to model performs packet forwarding, and the result is a trace that describes the temporal behavior. Before the OS is executed to generate traces, however, the most important contexts must be found. Which contexts are chosen depends on which state values affect the temporal behavior. In previous models, the researchers used packet and queue sizes. When all the contexts have been decided, we also need to find out how to store the traces afterward. Keeping the traces in RAM during the run, and later storing them to file is the fastest. That is a complicated issue as TinyOS is a resource-constrained device with only 10kB of RAM. The tracing is performed by running the OS in all the decided contexts. The result is a trace for each context which contains a sequence of events. We filter this data for each service automatically using the analysis tool described next.

2.3.3 Automatic analysis

When all the traces have been generated, they need to be translated into signatures that can be put together in a device file. Signatures are created automatically running an analysis script. The input of the script is a trace file, and the output of it is a set of signatures. For a given service, the tool locates all n invocations of it, which are called cases. It collects all the cases with identical sequences of events and parameters into a group. Each group represents one signature. We also want to capture the run times in between each event for the signatures, to get an idea of how long the processing stages take to execute. For each of these processing stages, we include one probability function, which includes data on the run times for each case in the signature.

2.3.4 Human Investigation

When we have completed the analysis, the set of signatures has to pass two manual tests to be considered valid. The first test requires that for each signature retrieved from the analysis, there should be no more than one context of a service corresponding to it. If there is more than one signature for one context, it means that the context is lacking class 6 events because at least two different sequences of events must have occurred in the same context. The second test requires that the temporal behavior of each sequence be unimodal for each signature. If for instance, half of the cases have execution times similar to each other, and the other half also have execution times that are similar to each other but entirely different from the first half, it can indicate that two

different behaviors are executed. If either test fails, it is a sign that the instrumentation step must be done again.

2.3.5 Generating and evaluating the model

When the signatures are validated, all the signatures for one service be placed in a device file and later turned into a Service Executable Model (SEM). This SEM can then extend the network simulator one wants to use. When we run the SEM with the simulator, it introduces the delay based on the observed temporal behavior of the service during the tracing. It can still be the case that the SEM is incomplete, which is why one should compare the behavior of the SEM with a real device to see if they have similar behavior when run.

An SEM contains all the merged signatures. Conceptually, it has a tree structure where each conditional event causes the branch to branch out to multiple paths. If the service does not start with a class 6 event, which are the conditional events that cause different instructions to run, the SEM starts with one branch. Later on, it continues to branch out until the end, in which the number of branches equals the number of contexts for the service. An SEM is either invoked by another SEM or from existing protocol models, which are called Functional Service Models (FSMs). The execution environment works in a way where FSMs first invoke SEMs, for instance when a transceiver model receives a packet, and the SEM invokes another FSM through triggers defined in the SEMs.

2.4 Related works

The related works to this thesis include previous work on tracing TinyOS and previous work on simulation of software execution delay. The most important related work includes the work on the CSWMM that the contribution of this thesis bases itself on, described in [Ste13], [KPG13b] and [KPG13a]. The previous work done includes the methodology [KPG13b] [Ste13] and two proofs of concept: one of the Google Nexus One [KPG13a], and another of the multi-core Samsung Galaxy Nexus [Oys16]. In contrast, this thesis focuses on the minimalistic single-core TelosB and single-threaded TinyOS.

2.4.1 Tracing framework

Tracing the behavior of TinyOS is discussed several times in the literature. Although the tracing framework we create must incur both a low tracing delay and memory usage, none of the investigated papers have low tracing delay as a primary requirement. Additionally, almost all papers focus on tracing the control-flow of the application for debugging of applications. The control flow is the sequential order of encountered events during the execution of the application. Since control flows do not require timestamps, most frameworks leave it out due to its high memory cost. Our tracing framework requires timestamps for the trace events to capture the length of processing stages.

Several attempts are made to capture the control flow of the applications that run on resource-constrained devices, but with no timestamps. In [DWDH09], a toolkit

is presented for visualizing the behavior of TinyOS applications. A call graph can be generated for a given application so that the user gets an overview of the functions that can be called. Then, the user can select the program actions to be traced (tracepoints). When the program is finished executing, a sequence diagram can be constructed from the resulting trace. In [DDHW08], a way of visualizing the local runtime behavior of TinyOS programs is presented. The result of tracing is a network-level snapshot in the form of a sequence diagram of the invoked functions. In [SEZ09], a tracing method is introduced that captures the control flow of executing applications and encodes the trace to reduce memory consumption. In [SEZ10], efficient and accurate tracing of WSN applications is presented using a novel tracing encoding scheme and a compression technique to reduce the memory consumption of traces. Since these solutions do not describe the time it takes to execute the functions, they cannot be used for our purposes.

In [HC09b], a lightweight tracing framework that enables tracing of behavioral and timing events. That includes timing information for each traced event, which is what is needed for this thesis. The resulting trace can be used to find out which functions were called and how long they lasted. In [HC09a], the same authors present an improved tracing framework called LIMOW. The problem with LIMOW and with most of the other papers is that the evaluation of their tracing frameworks only focuses on reducing the storage capacity and not tracing delay. This thesis requires an efficient tracing framework that incurs a low tracing delay for each trace event that does not drastically change the behavior of the CSW when tracing. Regardless, LIMOW appears to be the existing tracing framework that is closest to what is required in this thesis.

In [SSFM11], a generic and efficient logging framework called TinyLTS is presented as an alternative to ad-hoc tracing frameworks. The primary evaluation criteria used in the paper is flexibility and ease-of-use, which is not what is needed in this thesis. Our tracing framework must be efficient, and for that, a specialized solution is more suitable. Part of the motivation for creating a generic framework is that different projects have different requirements; some need to trace motes that are connected to a PC via serial communication and some need to trace motes that are not connected to a PC. The problem that the authors attempt to solve is that developers often end up with creating specialized solutions when tracing their resource-constrained applications. Since our tracing framework is made only for one purpose and must incur minimal overhead, we aim to develop a specialized solution.

2.4.2 Simulation of processing delays

Accurate simulation of intra-node behavior is a hot topic because general-purpose network simulators such as ns-3 are good for repeatability and scalability, but they do not usually have models that represent the intra-node behavior. Therefore, several solutions are proposed in the literature to extend network simulators such as ns-3 to add this behavior. The CSW model created for this thesis uses the CSWMM that is previously used to create two proofs of concept, and is described in detail in Section 2.3. The following solutions attempt to solve similar problems to what the CSWMM does.

In [BBB+10], a methodology to create high-level models of network devices is presented. It differs from the CSWMM in that the models created with the CSWMM are more deterministic and, thus, more accurate. In [SK15], a simulation model in ns-3

is created that simulates the processing delay of nodes in a Vehicular Area Network (VAN). A virtualization approach is chosen that assumes no detailed knowledge of the modeled hardware, which can have a negative impact on the accuracy of the simulated temporal behavior. That is an area that the CSWMM performs well at because the creation of CSW models requires a deep understanding of the CSW, and the processing delays defined in the model are based on traces from running it in the desired contexts. In [HMM16], an ns-3 model is presented for adding realistic processing delay when MSTAs (Mesh STAtions) forward packets in a Wireless Mesh Network (WMN). Since the application area for the model is Voice over IP (VoIP), processing delay is a critical element to simulate because VoIP requires low latency and jitter [Goo02], and processing delays contribute to them. Their processing delay model is a more simplified solution than the one that is created with the CSWMM, which means less accuracy as well as lower modeling effort. In [BRE+15], the authors create a model in ns-3 to accurately simulate the temporal behavior of packet processing in the NIC driver of a Linux system. The model uses a constant offset value to estimate the total intra-node latency, whereas the CSWMM simulates accurate timing of intra-OS services based on traces from a real device. The model also does not consider concurrent threads that might interfere with the simulation, whereas a model of a multi-core device has been created by using the CSWMM and is described in [Oys16]. In [RWR+14], a resource management extension for ns-3 is presented to model multi-core software routers. It is used to run intra-node resource contention models that provide realistic processing delays and packet loss for simulation of multi-core software routers.

Since the focus of this thesis is on the CSWMM, none of the above solutions are explored. To acknowledge existing solutions and work being done in the same field is still important. Comparing the performance of some of the other solutions with the CSW model created in this thesis might be an idea for future work.

2.5 Summary

The main contribution is the creation of a CSW model to be used in the general-purpose network simulator ns-3 to provide accurate temporal behavior of the WSN device TelosB that executes TinyOS. The creation of the model requires tracing the temporal behavior of TinyOS/TelosB. This background chapter introduces all the concepts required to understand that contribution. Now the foundation is laid to implement the CSW model.

Part II

Contributions

The contribution of this thesis is a CSW model of $^{TinyOS}/_{TelosB}$ that can be run in an execution environment in the ns-3 simulator. Creating it requires tracing the mote, which means we must create an efficient tracing framework in TinyOS. Thus, two things are created in this thesis: a tracing framework and the model itself. Both of them have functional and non-functional requirements that are described in Chapter 3. The creation and evaluation of the tracing framework are described in Chapter 4. The analysis and instrumentation of TinyOS and the drivers for TelosB are provided in Chapter 5. The creation of the CSW model is described in Chapter 6 and the evaluation of it described in Chapter 7. The evaluation of both the tracing framework and the CSW model depends on the requirements described in Chapter 3.

Chapter 3

Requirements

This chapter describes the requirements for the tracing framework and CSW model that are created in this thesis. Requirements are defined for the accuracy and scalability of the model. The creation of the model is done using the CSWMM, which is explained in Section 2.3. The steps to create the model are instrumentation of the CSW, execute it to generate traces, automatic analysis to create signatures, creating and finally evaluate the model. Everything is in place to perform the automatic analysis and create the model, but not the instrumentation, tracing, and evaluation steps. To be able to do them a tracing framework is needed that enables accurate tracing the CSW of a real mote. Despite the CSW model being the main contribution, the tracing framework is equally important since the CSW model relies on the tracing framework.

3.1 Tracing framework

Since a tracing framework is required to trace the CSW of *TinyOS*/*TelosB*, we first need to define what is a good and bad tracing framework. The biggest problem with TelosB is that it only has 10kB RAM available. Since the RAM capacity is low, the memory consumption must also be low. The time it takes to perform the tracing should also be reduced to avoid affecting the normal behavior. Therefore, the non-functional requirements for the tracing framework are to minimize memory consumption and the tracing delay. The functional requirements include the ability to instrument anywhere in the code, store traces in memory, and transmit traces to a connected PC. The requirements defined in this chapter form the guidelines to follow when creating a design in Chapter 4.

Figure 3.1 illustrates how the tracing framework can be used to create and evaluate the CSW model. A model centered instrumentation generates traces that tell how long each processing stage takes. Those traces can be used to generate signatures which are used to assemble the complete model. A metric centered instrumentation generates metrics that can be compared with simulation metrics generated by the model to evaluate the accuracy of the model. This chapter defines requirements that hold regardless of how the tracing framework is used.

3.1.1 Non-functional requirements

An ideal tracing framework should enable us to gather as much accurate information as desired when the device is executing. Also, the normal behavior of the device should

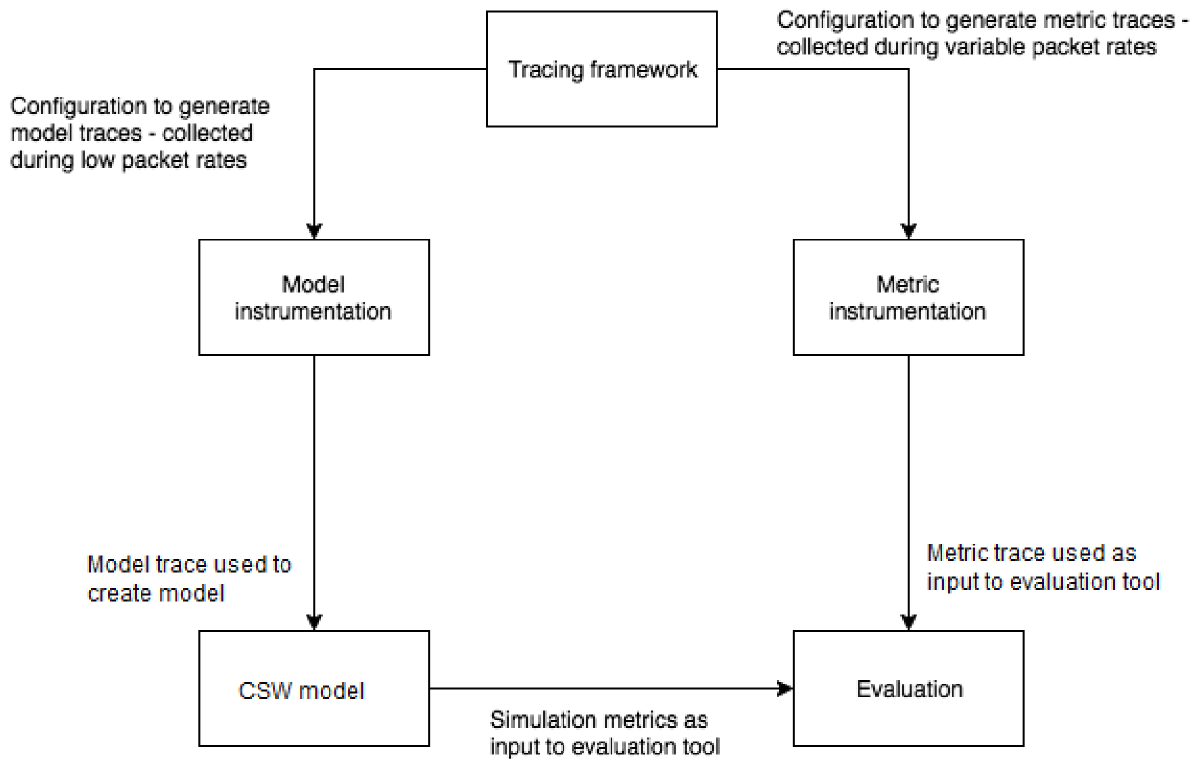


Figure 3.1: How the tracing framework is used

not change when it is traced. The problem with the ideal is that it does take time to trace and trace events must usually be saved somewhere on the mote, which means there are two limiting factors: tracing delay and memory consumption.

Memory consumption

The memory consumption of each trace event needs to be minimized. Therefore, the different types of events and verbosity of the trace events must be reduced to the extent it helps us reduce memory consumption. Contexts to trace in include various packet rates and sizes, and so we must be able to trace the behavior sufficiently and continuously while the device processes several packets at once. If only one packet can be traced before the buffer is full, we miss out on the device's behavior at high packet rates, and cannot accurately capture packet loss and packet delay.

Tracing delay

Tracing delay is the additional time that tracing adds to the intra-OS delay, and the requirement is to minimize it. The time it takes to trace is largely decided by the method of tracing and the amount of data to trace. Knowing which tracing delay is acceptable is not straightforward. The normal behavior of the device must be affected minimally. That can depend on factors such as the CPU speed of the device such as whether the CPU is single-core or multi-core and whether the OS is single-threaded or multi-threaded. Since TelosB is single-core and TinyOS is single-threaded, the tracing delay must be reduced. The reason is that when TelosB performs tracing, everything else must be interrupted, whereas a multi-core device might perform tracing in parallel with other tasks. As the CPU runs at 4MHz, however, some delay is acceptable because

hardware instructions take more time to execute than faster devices. In summary, there is no simple way of determining how much tracing delay is acceptable, but it should be minimized as much as possible.

Accuracy

Traces are required to be sufficiently accurate. The trace events must be transmitted and received in the right order with accurate timestamps. The need for an accurate timestamp depends on the use-case. A use-case where we need accurate timestamps is when tracing how long the processing stages take. Conversely, accurate timestamps are not needed when gathering metrics used to find out the percentage of successfully forwarded packets. In either case, the trace events must arrive in the same order they are traced.

3.1.2 Functional requirements

There are three functional requirements for the tracing framework: (1) the user must be able to place tracepoints anywhere in TinyOS, (2) the traces must be possible to save in RAM, and (3) the traces must be possible to transmit to a connected PC while the device is running. A tracepoint can be performed through a function call in TinyOS. Saving traces in memory is needed because we do not necessarily want to transmit traces immediately, but instead, store trace events in memory and defer the transmission. The ability to transmit traces to a connected PC while the device is running is required because the device has little RAM capacity, which can make it necessary to transmit traces several times throughout the execution to collect enough trace events. These functional requirements enable us to use the tracing framework to capture the CSW behavior of $^{TinyOS}/_{TelosB}$.

3.2 CSW model

An ideal CSW model in ns-3 is one that has the same packet forwarding related behavior as a real device in all possible ways, and the scalability of the model has no limits. The aim is to reach both of these goals, but that ideal will not be reached here either. The accuracy cannot be perfect because the model is still an abstraction of the device where no instructions of the OS are executed. Infinite scalability cannot be achieved either because it takes time to execute the model and the model consumes memory. The goal is that even without that level of detail and some overhead, the accuracy is reasonably high and the scalability of the model is good enough.

3.2.1 Non-functional requirements

We define two types of non-functional requirements: one for the accuracy of the CSW model, and another for the scalability. The first type includes requirements for accurate simulation of packet delay, jitter, and forwarding rate. The other includes requirements for scalable simulation when increasing the number of nodes to the simulation, the number of processed packets and the number of simulated seconds.

Accuracy of the CSW model

We require that the CSW model can accurately simulate the following list of behaviors:

- The end-to-end delay of sending a packet from Mote A to C via B.
- Jitter as a result of varying packet size and fill-level in OS packet queues.
- The forwarding rate when processing packets, relative to the packet size and packet rate.

The above points describe the types of behaviors network administrators observe every day when they evaluate the quality of their service. They see how long it takes for a packet to get from Node A to C (end-to-end delay) from the data collected from their networks. Furthermore, they see the variation in the end-to-end delay when sending several packets (jitter), and how many packets are successfully sent (forwarding rate). The CSW model should exhibit similar behavior as the real device for all of these behaviors.

Scalability of the CSW model

WSNs can inhabit thousands of nodes, and it is therefore important that the time added to the simulation execution time and the memory consumed be small. If it takes weeks to simulate a regular scenario, the model is useless. The simulation execution time when including the model should be low enough that users can simulate regular WSN scenarios in a reasonable time. We are interested in how much time is added to the simulation execution time and memory consumed when increasing the following parameters:

- packets an active node processing packets (simulation execution time),
- seconds an inactive node spends not processing anything (simulation execution time),
- active nodes processing packets (simulation execution time), and
- inactive nodes spend not processing anything (simulation execution time and memory consumption).

We distinguish between the four cases above because the overhead added by scaling them might differ. An active node processes x packets where an inactive node spends y simulated seconds being idle. An active node that processes packets causes more or less overhead than an inactive one being idle. Additionally, increasing the number of nodes might not result in a linear increase in overhead. If the complexity increases exponentially for any of the above scenarios, the scalability will probably be poor.

3.2.2 Functional requirements

We define two functional requirements that are not fulfilled by default when creating the CSW model. The first is integration with an existing transceiver model. An example of such a transceiver model is the CC2420 transceiver model described in

[IG]. Integrating with it means that we do not have to model the radio of the mote, which is complex in an by itself. The second requirement is the ability to collect simulation metrics such as packet delay, jitter, and forwarding rate of each device during the simulation. That enables us to debug and evaluate the model by comparing those metrics with a real mote, and it enables developers to measure the simulated performance.

3.3 Summary

In this chapter, we present the requirements to the CSW model and the tracing framework. The tracing framework is designed and evaluated in Chapter 4. The evaluation of the CSW model is provided in Chapter 7. Table 3.1 contains a summary of the requirements.

For	Type	Requirement	Action
TF	NF	Memory consumption	Minimize
TF	NF	Tracing delay	Minimize
TF	NF	Accuracy	Maximize
TF	F	Trace anywhere	Enable
TF	F	Save traces in memory	Enable
TF	F	Transmit traces	Enable
CSWM	NF	Accuracy	Maximize
CSWM	NF	Scalability	Maximize
CSWM	F	Transceiver integration	Enable
CSWM	F	Log metrics	Enable

Table 3.1: Functional (F) and non-functional (NF) requirements for the tracing framework (TF) and CSW model (CSWM)

Chapter 4

Tracing framework

In this chapter, a minimalistic tracing framework is designed and evaluated that can capture the temporal behavior of the CSW of *TinyOS*/*TelosB*, illustrated as Mote B in Figure 1.2. It is used to create and evaluate the CSW model. Traces collected at a low packet rate from a real mote are converted to signatures that describe the temporal behavior of the CSW services. Signatures are an integral part of the model since they describe the captured CSW behavior. After it is created, the tracing framework is again used to capture the behavior, but this time to assess the similarity of the model to a real mote.

Instead of using one of the existing tracing solutions, we use a specialized solution. Some of the existing solutions are presented in Subsection 2.4.1. The main reason for not using any of them is that the tracing delay is not considered in most of the papers that evaluate them. To us, however, reducing the tracing delay is an essential non-functional requirement. The studied solutions do contain some insights that we include, such as compression of traces on the mote to reduce memory consumption. Therefore, we create a specialized solution that is inspired by existing solutions instead of using them.

In Section 4.1, the design of the tracing framework is explained. In Section 4.2, it is evaluated with respect to the requirements defined in Chapter 3. In Section 4.3, the chapter is summarized.

4.1 Design

The design of the framework includes the format of trace events, the method of transmitting them from the mote to the receiving PC and how the receiving PC handles them. This section describes a design that addresses all the points, which is evaluated in Section 4.2.

Resulting traces must be possible to translate to CSW events because they contain the information required to create signatures. Previous proofs of concept [KPG13b, Oys16] store CSW events in memory of the traced device, but that is not a realistic option TelosB that only has 10kB of RAM. Instead, trace events are compressed on the mote and decompressed to CSW events on the receiving PC to reduce the memory consumption when tracing. The relationship between trace tuples and CSW events is displayed in Figure 4.1; each trace tuple corresponds to one or more CSW events.

The tracing framework is used for three different purposes: (1) creating the CSW model where accurate timestamps are needed, (2) evaluating it where accurate

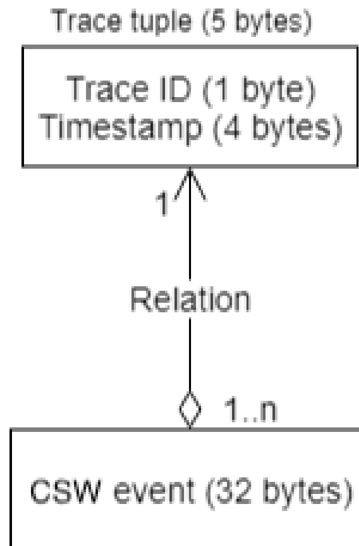


Figure 4.1: Relationship between trace tuples and CSW events

timestamps are needed and (3) evaluating it for prolonged periods where accurate timestamps are not needed. Purpose 1 requires accurate timestamps because the signatures must accurately describe the temporal behavior of the CSW. Purpose 2 also requires accurate timestamps, but this time to evaluate the accuracy of the intra-OS delay simulated by the CSW model. Purpose 3 does not require accurate timestamps and is for measuring the forwarding rate of the real mote against the CSW model to evaluate it. The tracing framework is designed for these purposes and accommodates their specific requirements in addition to the general requirements in Chapter 3.

4.1.1 Methods of tracing

As a result of the different needs in Purpose 1–3, two methods of tracing are proposed in this chapter: instant and buffered tracing. Buffered tracing is used for Purpose 1–2 and includes storing the trace tuples in memory until the buffer is full, and then transmit them all via serial communication to the connected PC. The trace events are in the format <trace ID (one byte), timestamp (four bytes)>. During the time that the mote transmits the trace tuples, the application is interrupted, and trace tuples that are traced close to the interruption must be discarded afterward. That enables the user to continue with the execution for indefinitely so long as the trace tuples affected by the interruption are discarded.

On the other hand, instant tracing can be used entirely without interruption because each trace event in the format <trace ID (one byte)> is sent immediately using serial communication. As it can be used indefinitely without interruption, it is used for Purpose 3. In contrast to buffered tracing, the timestamps are added by the receiving PC to form the trace tuple <trace ID (one byte), timestamp (four bytes)>. Therefore, the resulting trace tuple is in the same format as the top box in Figure 4.1 regardless of whether buffered or instant tracing is used.

Both methods have their advantages and disadvantages. The benefit of using instant tracing is that it is easy to trace an application without affecting the application significantly over an extended period. The downside is that the timestamps cannot

be trusted to be accurate. The benefit of using buffered tracing is that the timestamps are accurate and the tracing delay is low. The downside is that when trace tuples are transmitted, the application is interrupted. The tracing delay for each trace event is low with both methods: 20 μs (80 CPU cycles) for buffered and 40 μs (160 CPU cycles) for instant tracing.

Tracing

How tracing works depends on which method is used to trace and on which platform. Depending on whether instant or buffered tracing is used and whether a real mote or an emulated Cooja/MSPSim mote is traced, different things happen when tracing and transmitting the trace events. The real mote is physically connected to a PC and transmits the trace tuples using serial communication. The emulated Cooja/MSPSim mote transmits data to the Mote Output plugin in Cooja, which can be downloaded to a file after the execution. When instant tracing is used, trace IDs are sent immediately and can be done indefinitely without interruption. With buffered tracing, the trace event buffer fills up until it is full and the trace tuples are transmitted to the receiving PC. In all cases, however, the instrumentation of the CSW is the same. Moreover, the generated traces have the same format and are therefore handled the same way after the tracing. Consequently, no effort is required when switching from one method or platform to the other.

Real mote The process of tracing the real mote is illustrated in Figure 4.2. The traced mote is connected to a PC which executes a program that listens for any data received to the serial port. Depending on whether buffered or instant tracing is used, the received trace events are five or one bytes in size. The tool used to listen for the data received from the mote is a modified version of the *sr* program that resides in the serial communication tools in the TinyOS repository. It is modified to listen for trace events instead of packets. When the program finishes, it writes the trace to a file.

If buffered tracing is used, the trace tuples are stored in RAM of the traced mote until the buffer is full. When it is full, the buffered trace tuples are transmitted using serial communication. Each tuple contains five bytes which are sent with a waiting time of 100–500 μs in between each byte. The reason for the waiting time is to avoid characters being lost in transmission. The receiving PC stores the received bytes in a trace tuple array.

On the other hand, if instant tracing is used, the trace IDs are transmitted immediately from the mote to the PC. The receiving PC receives one byte at a time and appends a timestamp (using the rdtsc x86 assembly instruction) to each trace ID to create a trace tuple. The method is beneficial for its simplicity but suffers from poor timestamp accuracy because it takes a variable amount of time to receive the trace ID.

Cooja When tracing in Cooja, instant tracing is the fastest and most accurate method to trace. It is most accurate because the timestamp is retrieved from the internal simulation time of Cooja and is the fastest because sending a byte using serial communication takes less than 20 μs with an emulated mote in Cooja (40 μs on a real mote). Data transmitted using serial communication is outputted in the Mote Output plugin as clear text with the simulation time, which is used as the timestamp of the

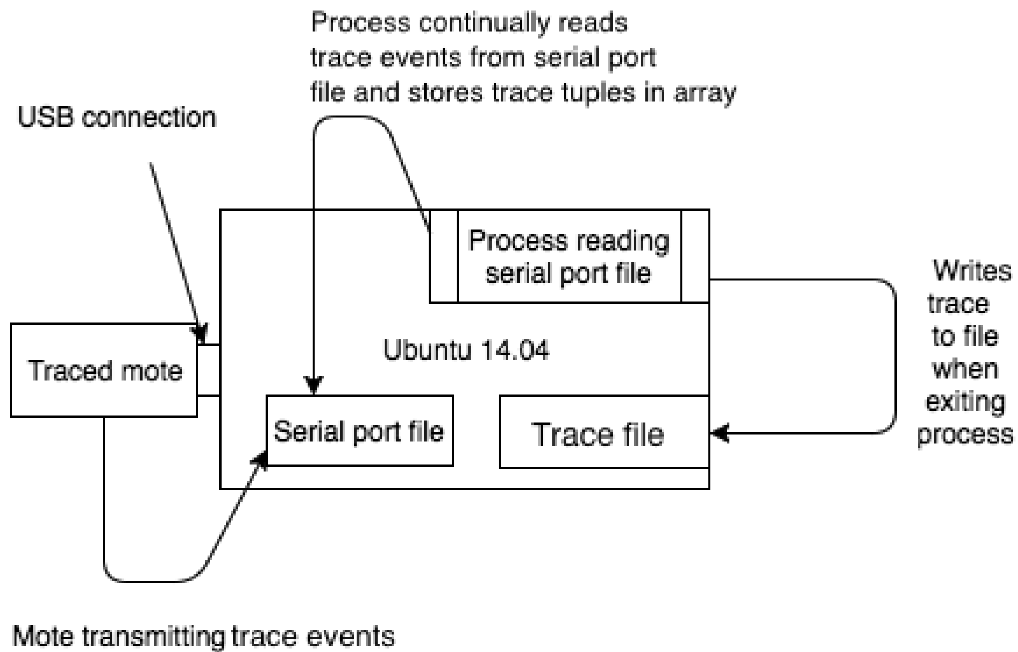


Figure 4.2: Process of tracing a real mote

trace tuple. The low overhead is beneficial when debugging and testing because it lowers the tracing delay, but also means that Cooja is less accurate than the real mote.

Minor modifications are made to the Mote Output plugin to accommodate for tracing in Cooja. The simulation time that is printed out is by default in milliseconds, which is not precise enough. Therefore, it is changed to display simulation time in microseconds, which is the highest precision that Cooja offers and the same that the real mote uses. Another modification is made to print out a new line for each character that is sent using serial communication. The reason is that each new line character sent using serial communication results in a new line of output in the Mote Output window, which includes the trace ID and Cooja simulation time in microseconds. Without the newline character, each trace event must be printed out with an explicit newline as well. That doubles the tracing delay because the trace ID and a newline character must be sent instead of just the trace ID. These are small changes, but at least the change in timestamp precision must be made for the tracing to work and the additional newline character change is optional.

4.1.2 Format

The design of trace events needs to fulfill the requirements defined in Chapter 3. Since the primary requirements have to do with minimizing tracing delay and memory consumption, the focus is on creating the most straightforward solution regardless of how inflexible and specialized it is. To minimize memory consumption, we compress the trace events on the mote and later decompress them on a PC. Even though instant tracing transmits the trace events immediately and thus does not occupy memory, it still requires compression of trace events because only one byte can be sent using serial communication at a time. Consequently, the task here is to compress the CSW events to the trace tuple containing a trace ID and timestamp.

Compressing 32-byte CSW events to five-byte trace tuples is not trivial because it requires knowledge about what data in CSW events is inferable. In TinyOS for instance, we can infer the Process ID (PID) because the OS is single-threaded. We have found that the only necessary data needed for each trace event in TinyOS is a one-byte location/trace ID (0–255) and a four-byte timestamp. Therefore, the CSW event in Listing 4.1a can be compressed and traced as the trace tuple displayed in Listing 4.1b.

```
(a) HIRQENTRY 0 1 427182894 1 0 0 <service> <location>
                          |
                          v
(b)                     0 427182894

             ____________
Trace ID = | Location ID |
             ____________

Trace tuple = <Trace ID (1 byte), timestamp (4 bytes)>
```

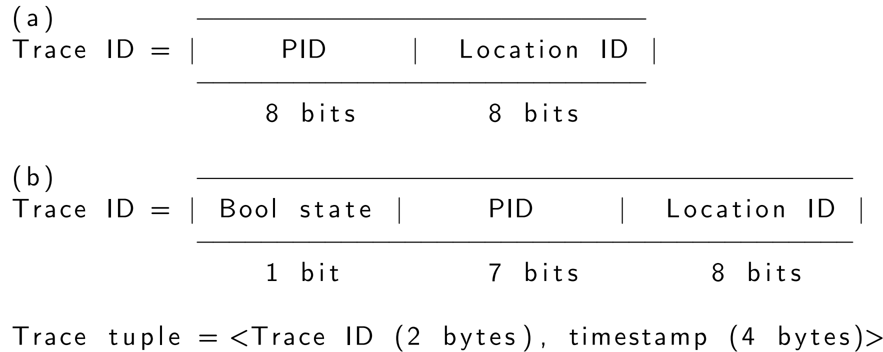
Listing 4.1: Simple compression of CSW events

The example above only works when the context is simple, for instance, that only one thread is executing. If two threads can execute and both threads run the same instrumented code at different times, the trace IDs need to include information about which thread is tracing which event, since the CSWMM also works for multi-threaded and multi-core devices [KPG13a] [Oys16]. Listing 4.2a and 4.2b format the trace ID so that it reflects whether PID 0 or 1 is tracing. Which thread is tracing is captured by allocating the most significant bit in the trace ID byte to the PID of the tracing thread. That means each thread can trace up to 128 different locations in the code, a reduction from 256 locations when only one thread is executing. Since TinyOS only runs with one thread, the example in Listing 4.1 works fine.

```
 HIRQENTRY 0 0 427182894 0 0 0 <service> <location>
                          |
(a)                       |
                          v
                        0 427182894

 HIRQENTRY 0 1 427182894 1 0 0 <service> <location>
                          |
(b)                       |
                          v
                      128 427182894

             __________________
Trace ID = | PID | Location ID |
             __________________
             1 bit    7 bits

Trace tuple = <Trace ID (1 byte), timestamp (4 bytes)>
```

Listing 4.2: Compression of CSW events with two threads

Listing 4.3a contains a trace ID format where the same tracepoint might be executed from up to 256 different threads, and therefore the one-byte trace ID is not sufficient. That is solved by increasing the size of the trace ID from one to two bytes and thus only increasing the total trace tuple size by one byte. All the bits of the trace ID should be used efficiently. If the possible PIDs in an OS are 142, 24 and 251, they should not be stored in the trace ID and occupy a byte, but instead be mapped to 0, 1 and 2.



```
(a)
              ________________________________
Trace ID = |       PID       |  Location ID  |
              ________________________________
                 8 bits          8 bits

(b)
              ______________________________________________
Trace ID = | Bool state |       PID       |  Location ID  |
              ______________________________________________
                 1 bit          7 bits          8 bits

Trace tuple = <Trace ID (2 bytes), timestamp (4 bytes)>
```

Listing 4.3: (a) Compression of CSW events with many threads. (b) Compression of CSW events with many threads and a boolean state variable.

The need to trace many different PIDs is just one example of a type of context that might need to be included in the trace ID. Listing 4.3b shows an example of a case where the developer wants to capture the value of a boolean state variable, and thus allocates one bit for the value of that variable in the trace ID. These examples demonstrate the flexibility of the compression method.

In the case of trace IDs that are larger than one-byte, the instant tracing method does not work correctly in TinyOS because two bytes cannot be transmitted using serial communication reliably without a waiting time in between each byte. Instant tracing is mostly used when timestamps are not that important, so in those cases, the trace IDs must instead be stored in RAM or flash memory without the timestamp. While instant tracing is a suboptimal method, it is used to explore the application and understand it in complicated contexts.

4.2 Evaluation

We evaluate the tracing framework against the criteria defined in Chapter 3. The metrics used to evaluate the tracing framework are tracing delay and memory consumption, and the most critical requirements regard minimizing them. The tracing framework must be efficient and enable us to capture all the necessary information to be able to use the CSWMM to create a sufficiently accurate CSW model. *TinyOS*/*TelosB* is to be traced for three different purposes as mentioned earlier in the chapter. Purpose 1 requires accurate timestamps and is for creating the model. Purpose 2 requires accurate timestamps and is for evaluating it. Purpose 3 does not require accurate timestamps and is for indefinite tracing when evaluating the model. If the tracing framework can fulfill the requirements while providing a way to use it for all three purposes, we have succeeded.

4.2.1 Memory consumption

The memory consumption of trace events depends on the method of tracing. Instant tracing transmits the one-byte trace ID immediately, resulting in practically no memory consumption. Buffered tracing stores the five-byte trace tuple in memory until the buffer is full. With both methods, it is impossible to reduce the size of the trace event further without removing necessary information. The one-byte trace event cannot be reduced any further because at least one byte must be transmitted when using serial communication. Since instant tracing enables indefinite tracing without interruption, Purpose 3 of the tracing framework is possible. Purpose 1 and 2 are possible because of the accurate timestamps added when using buffered tracing. Therefore, without reducing the precision of the timestamp, it is impossible to reduce the size of the five-byte event.

On the other hand, the size of the timestamp can be reduced from four to two bytes if the buffered tracing method uses a timestamp to record the change in microseconds since the last trace event instead of the number of microseconds that have passed since the application started. It would require the change between current time and the time of the previous trace event to be less than 2^{16} μs or 64 ms. Based on preliminary experiments, it is clear that the difference between two traced events for a packet being processed never exceeds 64 ms.

When the mote is traced for Purpose 1, the memory consumption can be reduced even further. As Purpose 3 is about creating the CSW model, the trace is captured only at a low packet rate. Therefore, the same sequence of trace IDs should be captured every time. By knowing the trace ID sequence beforehand, only the timestamp has to be transmitted, and the trace ID can be inferred afterward. Then the memory consumption of a traced event can be reduced from five to two bytes, where only the difference between the current and previous event is stored and transmitted. By reducing the size of the trace event, it enables storing 3500/2=1750 events in RAM at once. As the behavior is predictable at a low packet rate, however, it is not necessary to trace the mote for prolonged periods. Moreover, it is possible to redo the tracing several times if more trace data is needed. None of these ways of reducing the memory consumption of trace events are prioritized in this thesis for two reasons: Purpose 1–3 are fulfilled even without the optimizations, and they require extra work for handling special cases.

4.2.2 Tracing delay

Tracing delay, as mentioned in Chapter 3, is the additional time that tracing adds to the intra-OS delay. How much time depends on whether buffered or instant tracing is used. The delay caused by buffered tracing is bound by the time it takes to fetch the current time in microseconds. Reading the current time for each trace event is necessary because the event must contain an accurate timestamp. That delay is 20 μs (80 CPU cycles) when tracing a real mote, which is considered low.

The delay caused by instant tracing is bound by the time it takes to send a byte over serial communication with printf. The amount of time it takes to transmit the byte to the receiving PC is variable, but the time it takes to call printf with one byte is 40 μs (160 CPU cycles) when tracing a real mote. Although the delay is twice the amount as buffered tracing incurs, it is still low. The main problem with instant tracing is that the

timestamps are inaccurate because of the variable serial communication transmission time.

4.3 Summary

This chapter presents the tracing framework used to capture the data from the TelosB mote to create and evaluate a realistic CSW model in ns-3. The solution is implemented in TinyOS as a component that uses TinyOS' printf function to transmit trace events. Trace events are compressed on the mote and decompressed later on a PC, which results in trace events that are low in memory consumption. When using buffered tracing, the events are stored in RAM until it is full before they are transmitted using serial communication. When using instant tracing, the events are transmitted immediately using serial communication. The result is a flexible way of tracing in which the tracing delay and memory consumption are low with both methods. Therefore, the requirements that are defined in Chapter 3 are satisfied.

Tracing type	**Delay**	**# events w/o interruption**	**Timestamp accuracy**
Buffered tracing	20 μs	700	High
Instant tracing	40 μs	No limit	Low

Table 4.1: Difference between buffered and instant tracing

Table 4.1 sums up the difference between buffered and instant tracing. With buffered tracing, timestamp accuracy is high, the memory consumption is five bytes, and it takes 20 μs to trace an event. Furthermore, the number of events the mote can trace before having to transmit them is 700. With instant tracing, timestamp accuracy is low, the memory consumption is one byte, and it takes 40 μs to trace an event. Since the trace events are transmitted immediately using serial communication, there is no limit to how many events can be traced without interruption.

Chapter 5

Analysis and Instrumentation of TinyOS

In this chapter, the packet forwarding part of TinyOS for TelosB is analyzed, and the instrumentation of it is described. The instrumented CSW in TinyOS includes the CC2420 driver and Berkeley Low-power IP (BLIP) library. BLIP contains the IPv6 and 6LoWPAN implementation. Two instrumentation configurations are defined: the model centered instrumentation for creating the CSW model and the metric centered instrumentation for evaluating it. Consequently, if the evaluation indicates that the model is not accurate enough, it might be because of something missed in the analysis and instrumentation.

In Section 5.1, the forwarding application used to create and evaluate the CSW model is explained. In Section 5.2, the packet forwarding process in TinyOS is described. In Section 5.3, the instrumentation configurations are explained.

5.1 Forwarding application

Our TinyOS forwarding application is executed to capture the CSW behavior in different contexts. All three motes in the linear chain topology in Figure 1.2 run the application. Mote A sends packets to Mote B, and Mote B has a hard-coded route to Mote C. Mote B's application is instrumented and traced for the creation of the CSW model. The application works for both a real and emulated Cooja/MSPSim mote, which is beneficial because it is easier to debug the program with a simulator such as Cooja/MSPSim instead of on a real mote. Additionally, it gives the opportunity to compare the performance and differences between the real mote and Cooja/MSPSim. The difference in what each mote does is decided through if statements that check if the TOS_NODE_ID is 1, 2, or 3. For the real motes, the application is explicitly compiled for Mote A, B or C. In Cooja/MSPSim, the TOS_NODE_ID is assigned automatically. As such, all three motes run the same code where the TOS_NODE_ID is different.

5.1.1 Modifications to TinyOS

The application uses IPv6 which by default requires devices to send ICMPv6 packets and the CC2420 driver by default requires acknowledgment of packets. Both ICMPv6 packets and acknowledgment packets are disabled because only a simple packet

forwarding scenario is modeled. Clear Channel Assessment (CCA) is implemented in two ways on the mote as mentioned in Chapter 2. The first is a backoff timer in TinyOS that makes the mote wait a random amount of time before sending packets to avoid collisions. The second is a feature that the CC2420 radio chip implements to prevent packets from being sent if it senses that the medium is not clear. The radio chip's CCA functionality is kept enabled, but the backoff timer is disabled because the random backoff time causes unwanted variation in intra-OS delay. Although a backoff is usually necessary to avoid occupying the channel for too long and collision when resending packets, that is not needed here. These two modifications are made to simplify the forwarding process.

5.1.2 Network stack

Figure 5.1 contains an overview of the network stack of the application. Mote A sends a packet using UDP for transport layer, IPv6 and 6LoWPAN for its network-layer protocols, and IEEE 802.15.4 for MAC sublayer and PHY. The CSW model in this thesis does not simulate 6LoWPAN fragmentation of packets, but it can be considered in future work. Combined, the headers in the packets sent by Mote A have a size of 36 bytes, and 38 bytes when adding the two CRC bytes. Further in the thesis when results are presented, figures distinguish between UDP payload size and packet size. UDP payload size of zero bytes means a packet size of 36 bytes.

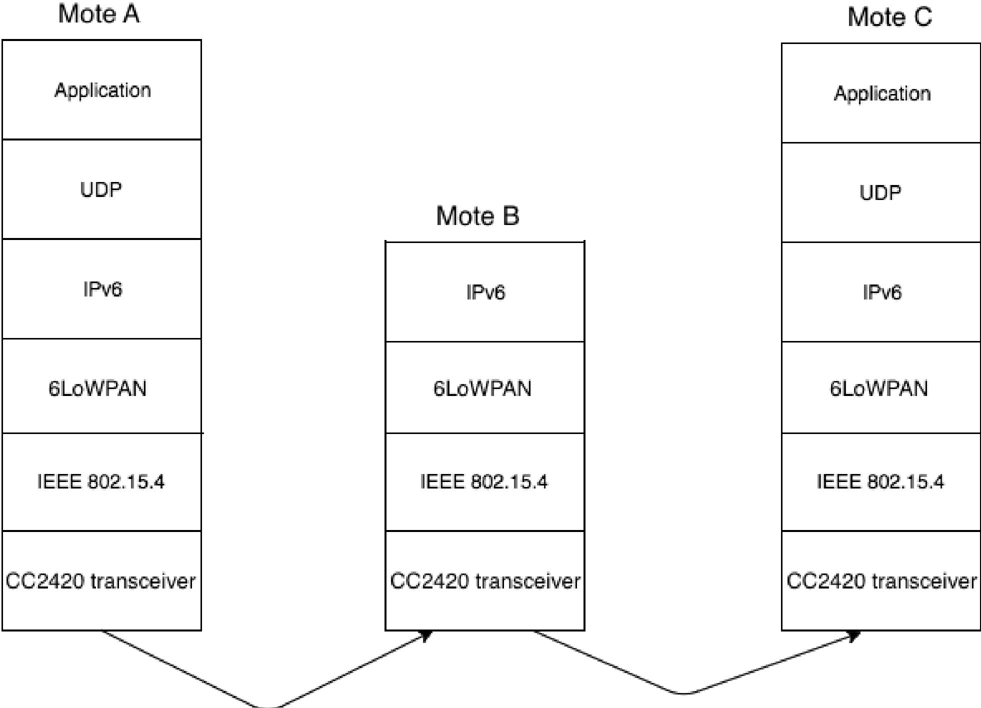


Figure 5.1: Forwarding app network stack

5.2 TinyOS analysis

In this section, the packet forwarding process in TinyOS is analyzed. The CC2420 radio chip driver and the BLIP (6LoWPAN and IPv6) library comprise the instrumented CSW and the entire packet processing flow can be divided into a receiving and sending part, as illustrated in Figure 5.2. In this case, receiving and sending mean the first and second half of the CSW packet processing. The receiving part lasts from starting to write a packet from the CC2420 RX queue into memory until enqueueing it into the IP packet queue (IPAQ). The sending part lasts from taking a packet from the IPAQ, and it is transmitted. Each part can handle only one packet at a time. Figure 5.3 illustrates an overview of the functions that are called. In TinyOS, the same process executes the drivers, the OS code, and applications. Therefore, when it is written that the driver, OS or application does something, it is always the same process.

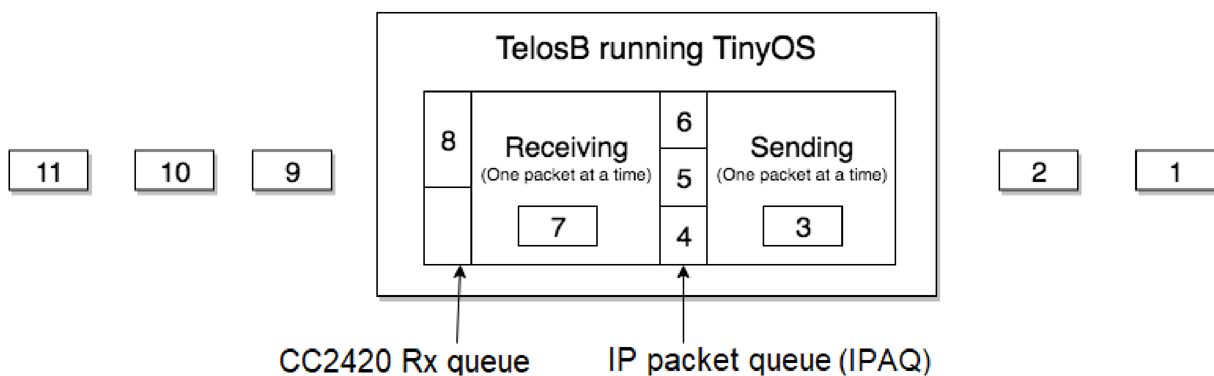


Figure 5.2: Summary of CSW flow when packets are forwarded

CC2420 driver when receiving The OS is notified that a packet is being received when the SFD pin on the CC2420 radio-chip invokes a hardware interrupt in the OS. SFD stands for Start of Frame Delimiter, which means the mote has received the first bytes of a new packet. When it is fully received, the hardware interrupt event interruptFIFOP.fired in the CC2420 driver is invoked, seen in the top box in Figure 5.3. Only one packet can be written into RAM at a time, and deferred packets get processed once the current one is finished with the receiving part.

If the driver is ready to process a new packet, it starts the three-step process of writing the packet into memory. All three steps use the readDone hardware interrupt event in box number two in Figure 5.3, where the state variable m_state has a different value in each step. The steps are to read (1) the length byte, (2) the FCF bytes, and (3) the payload of the packet. When the full packet is written into RAM, the driver checks the last byte of it to see if the CRC check succeeded (a check performed by the CC2420 radio chip). If it fails, the mote drops the packet and starts reading the next one. If it succeeds, the receiveDone_task task, box number three in Figure 5.3, is posted to run later.

receiveDone_task's job is to send the packet to the upper layer protocols and hand it over so that the next packet can be processed. First, a duplication check drops previously received packets. Next, the packet is sent to the layer handling 6LoWPAN.

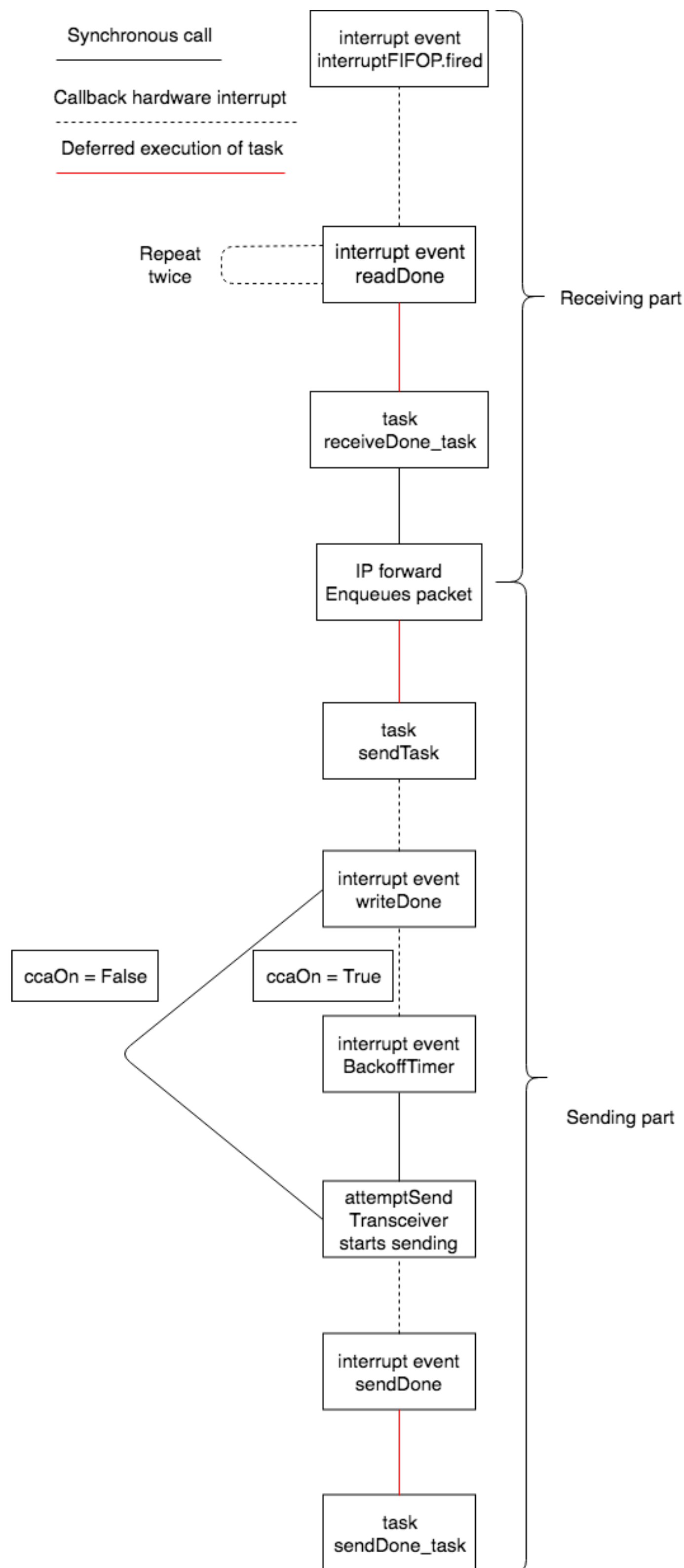


Figure 5.3: Overview of CSW flow when packet is forwarded

IP layer The 6LoWPAN adaptation layer decompresses the packet header before it sends the packet to the code handling the forwarding of packets. As the IP layer finds out that the packet is destined for another mote, it looks for a route which it finds because we add a route between Mote B and C in the forwarding application. Next, two things happen: the IP layer draws an entry from the send info pool (3 entries in total), and the packet is a placed in a queue (max 12 packets). If either the send info pool is empty or the packet queue is full, the packet is dropped. The entry from the send info pool is not put back in the pool, and the packet is not removed from the packet queue until the packet is transmitted. Since the packet queue has twelve entries, and the pool only has three entries, the bottleneck is the pool. If the pool and queue have the capacity for another packet, sendTask is posted to be executed later, which is the last thing receiveDone_task does. At this point, the receiving part ends and first now can the next packet be written to memory, unless the packet is dropped earlier due to failed CRC check.

sendTask starts the sending part, and its job is to prepare a packet enqueued into the IPAQ to be sent to its destination, which is Mote C in this case. The IP layer has a boolean state variable called radioBusy, which is used to enforce the rule that only one packet is allowed to be attempted sent from the IP layer at a time. If radioBusy is true when entering sendTask, sendTask returns and is reposted when the current packet is finished being transmitted. Otherwise, radioBusy is set to true, and the packet proceeds with being sent. The headers for the packet are set, and it is sent to the lower layers.

CC2420 driver when sending The sending part of the CC2420 driver continues the forwarding by writing the packet to the TX queue of CC2420. When the packet is written to the TX queue, a hardware interrupt event is invoked, and it is time to finally attempt to send the packet by calling the attemptSend function that is the second last box in Figure 5.3. In this application, the backoff timer is disabled. If it were enabled, the callback function to the backoff timer would invoke attemptSend, as can be seen in Figure 5.3. Instead, attemptSend is invoked right after the packet is written to the TX queue. When the radio has finished sending the packet, a hardware interrupt is called which signals a sendDone event in the IP layer. There, the send info entry is placed into the pool and the packet dequeued from the packet queue. In the CSW model, this is the place where the packet is dequeued from the IPAQ. The sending part also ends at this point, and the next packet can be sent.

Task scheduler All tasks that are posted are executed by a scheduler that runs within an infinite loop. When there is no task to be executed, the MCU goes to sleep for a while. Tasks cannot preempt each other or events, which means that they must be relatively short lasting. For instance, receiveDone_task must finish before sendTask can be executed. If receiveDone_task runs forever, sendTask will never end, and the packet will never be forwarded.

Modeling the CSW As mentioned above, the CSW can be split into two parts, receiving and sending, with a byte queue before the receiving part (CC2420 Rx queue) and a packet queue in between the receiving and sending parts (IPAQ). For the modeling to be realistic, however, it should be more finely grained than just two

processing stages. The CSWMM enables that by splitting the captured CSW process into separate services each with their processing stages.

The CSW model simulates the services as either Packet Handling Services (PHSs) or Work Scheduling Services (WHSs) in the CSWMM, which are presented in Section 2.3. In Figure 5.3, InterruptFIFOP.fired, readDone, receiveDone_task, sendTask, BackoffTimer.fired, sendDone and sendDone_task are all considered PHSs. PHSs are executed within a WSS, and the CSW model uses two of them: one for executing hardware interrupts and another for posted/deferred tasks. The tasks are executed by the task scheduler that executes within an infinite loop with some delay in between each iteration. Hardware interrupts, on the other hand, are executed as soon as they occur.

The send info entry in the IP layer can be viewed as a ticket that is used to make sure that only three packets can wait to be sent at once. Since the send info pool is a bottleneck, it is modeled in the CSW model, but as a packet queue instead of a pool. Throughout this thesis, it is referred to as the IP packet queue (IPAQ), also in the experiments with a real *TinyOS/TelosB*. The packet queue with twelve entries is ignored because it is not expected to be a bottleneck.

5.3 Instrumentation

In this section, the instrumented locations in the CSW of TinyOS are presented. A distinction is made between instrumentation used to generate model- and metric-traces. Model-traces are used to create the CSW model, and metric-traces to compare the performance of it with a real mote. Figure 5.4, an extension of Figure 5.3, contains an overview of the functions called in the CSW and the tracepoints used for the two instrumentation configurations. The following subsections explain them in more detail.

5.3.1 Model

The model centered instrumentation is a configuration of tracepoints where the focus is on gathering data to create signatures. A model-trace captures which services are called, how long they take to execute and events within the services that are important to know exactly when they occur. In our case, those events are when packets are enqueued into the IPAQ and when the radio chip is ready to attempt to transmit a packet.

Aside from two tracepoints, all model-tracepoints in Figure 5.4 are at the beginning and end of functions. The CSW model must include more than just the beginning and end of services, but all of the required CSW events can be inferred from the tracepoints in the figure. What all the tracepoints have in common is that it is necessary to know when each time they are executed to create the CSW model. All the tracepoints placed at the beginning and end of functions are needed because the signatures must describe how long time it takes to execute each function, and also the time in between the services.

The two remaining tracepoints are placed where the CSW attempts to enqueue a packet to the IPAQ and when the radio is ready to transmit. The enqueueing of a packet is necessary to capture because the CSW model simulates enqueueing and

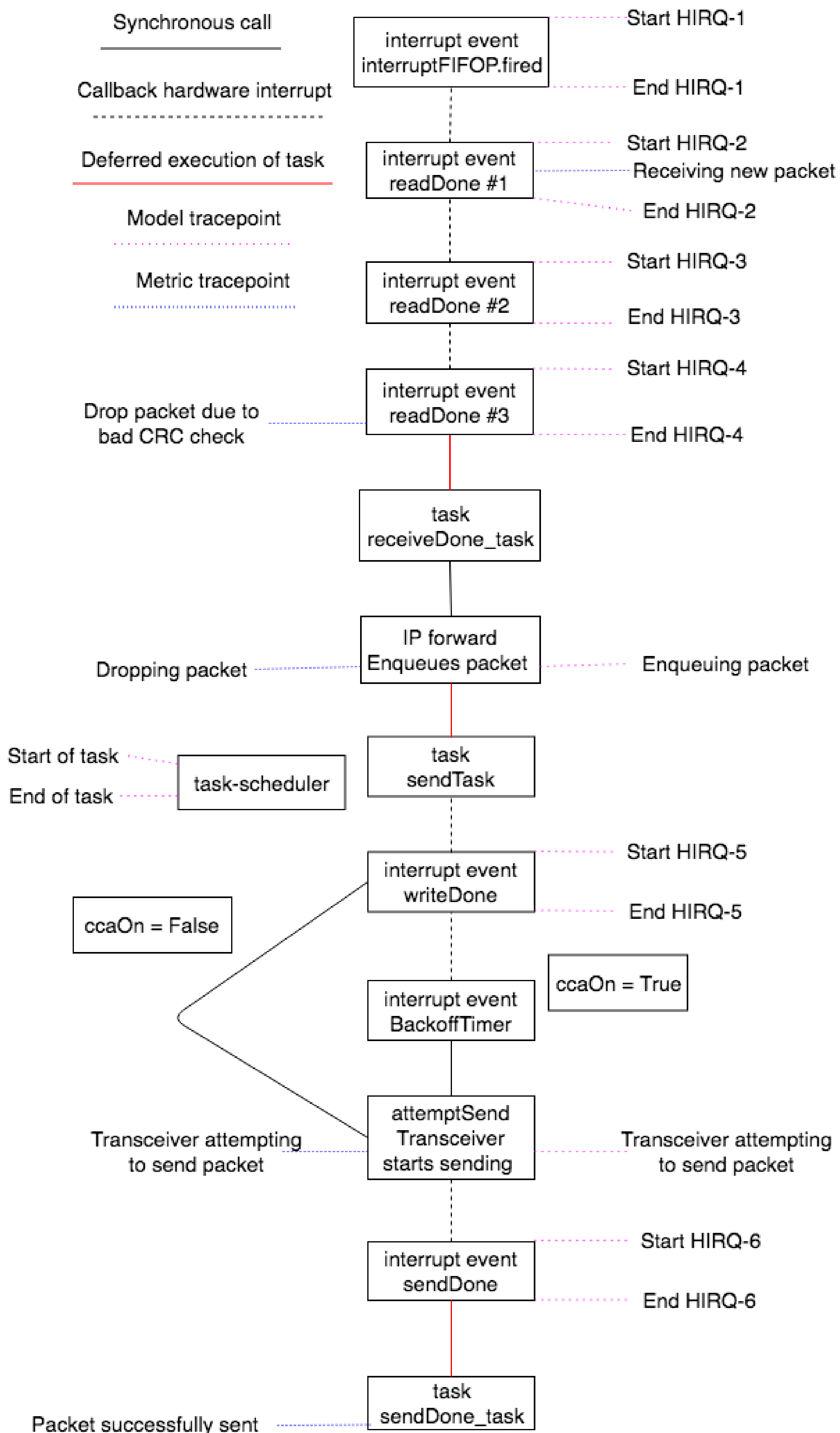


Figure 5.4: Overview of instrumentation in CSW

dequeuing of packets and needs to know the time within a service that packets are enqueued. When the radio chip is ready to attempt to transmit the packet must be captured because the radio chip can start to send it while the CSW is executing.

5.3.2 Metric

For the metric centered instrumentation, the focus is on measuring the intra-OS delay and packet forwarding rate. We care about when packets are received, sent and dropped. That is done by instrumenting five places, though the number varies depending on what data needs to be traced.

Figure 5.4 shows the locations in the CSW that belong to the metric instrumentation configuration, which make it possible to calculate the intra-OS delay of processing a packet and if it is successfully sent or dropped. The tracepoints placed where the OS has just received a packet and attempting to send a packet are used to measure the intra-OS delay. Only the trace event for the first attempt is used to calculate the intra-OS delay for a packet because multiple attempts only occur when the radio chip backs off due to the channel not being clear. In those cases, extra delay is added to the packet delay, but that delay is due to the transceiver delaying the transmission. When using the CSW model, that delay should be handled by a separate CC2420 transceiver model. Three tracepoints are used to measure the percentage of forwarded packets: when dropping packets due to failed CRC check or the IPAQ is full, and when a packet is finished transmitting. The size of the packet is necessary to trace in case the size of received packets changes from packet to packet. The experiments in Chapter 7 use these tracepoints to measure the accuracy of the CSW model.

Not all the above tracepoints are active, however, when conducting the experiments to evaluate the accuracy of the CSW model. Experiment 1 and 2 measure the intra-OS delay of packets being processed and Experiment 3 measures the packet forwarding rate. Additionally, Experiment 2 and one of the runs in Experiment 1 require tracing the packet size of received packets so that the ns-3 simulation program can replicate the packet sequence that occurs in the real mote.

5.4 Summary

In this chapter, the forwarding application used for the experiments and the modeled CSW are described and analyzed. The CC2420 driver and the BLIP library are instrumented, as well as the task scheduler. Additionally, the instrumentation of the CSW is explained.

Chapter 6

Model creation

In this chapter, the creation of the CSW model is described. Traces gathered from running Mote B at a low packet rate are used to create signatures, and finally a device file. The model is implemented in ns-3 as a C++ class that invokes SEMs in the device file to add the temporal behavior of the CSW to the simulation. After having collected traces, the remaining steps of the CSWMM to create the CSW model are:

- Verify that the trace is accurate through analysis.
- Decompress the model-trace to CSW events.
- Run the automatic analysis script with the CSW events as input to generate signatures.
- Create a CSW model in ns-3 that uses the signatures to simulate the packet forwarding.
- Create a simulation program in ns-3 that uses the model to forward packets.

Figure 6.1 shows the relationship between traces, CSW events, signatures, SEMs and the CSW model. A trace tuple refers to one or more CSW events. Several CSW events can describe several signatures. An SEM is made from one or more signatures, and one or more SEMs is used in a CSW model.

Section 6.1 presents ways to analyze a trace to mend errors and decompressing the model-trace to a list of CSW events, Section 6.2 explains in detail how the CSW model of TinyOS/TelosB works, and Section 6.3 summarizes the chapter.

6.1 Analysis of trace

A trace from Mote B might be inaccurate and contain inconsistencies that require analysis to find. Trace tuples only contain a trace ID (one byte) and a timestamp (four bytes), as seen in Listing 6.1 and 6.2. Therefore, they are easy to analyze. Each trace tuple represents one or more CSW events. This section describes how a trace can be shown to be accurate, and if not, mend it.

The trace used to create the CSW model is collected at a low packet rate, and so the same outcome is expected in the forwarding application every time a packet is processed. Even if the packet rate is high, the processing stages in TinyOS take around the same amount of time each time they are executed. In a more complex CPU, on the

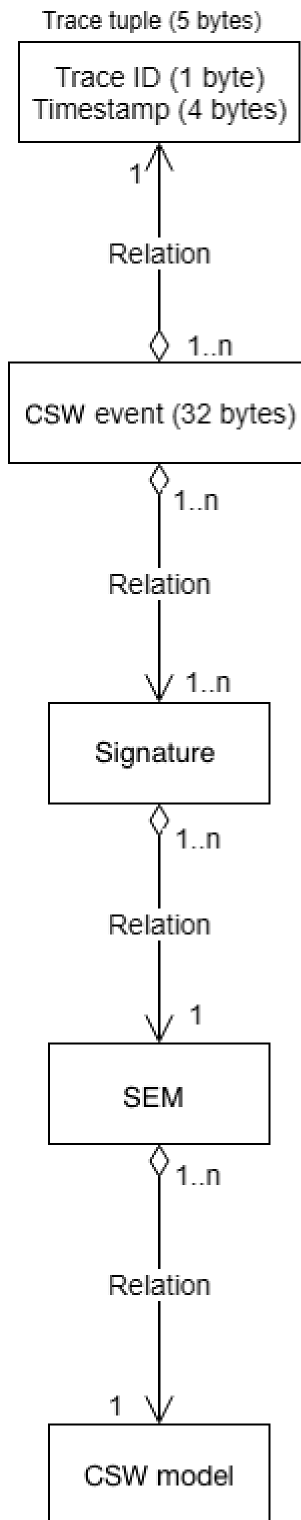


Figure 6.1: Relationship between trace ID, CSW events, signatures, SEMs and the CSW model

```
<timestamp 1> <trace id 1>
<timestamp 2> <trace id 2>
<timestamp 3> <trace id 3>
```

Listing 6.1: Format of trace tuples

```
321451 5
321455 2
321462 1
```

Listing 6.2: Example of trace tuples

```
 100 1            100 1
 104 2            104 2
 106 3            106 3
 111 1            111 1
 114 2 --fix--> 114 2
 116 3            116 3
 121 1
*118 2
 126 3

* Timestamp too early
```

Listing 6.3: Example of trace with a timestamp error

other hand, the frequency can increase when more things are happening, and a more complex OS can optimize the packet processing if multiple packets are waiting in a queue. As such, the processing times for the processing stages should be similar when tracing. If they are not, it might be because of an error with the tracing.

Analyzing the trace can reveal errors that can be fixed manually. As an example of a trace error, consider the timestamp error on the left-hand side of Listing 6.3 in the second last trace tuple. The timestamp is three time-units in the past as compared to the previous trace tuple. Since it looks like the trace tuples are cyclic with the trace IDs 1–3 occurring over and over, a solution is to remove the last cycle, as is done on the right-hand side. The tool used to identify this kind of error and related tracing errors outputs the maximum, average, median and minimum time differences between two trace IDs, and all the various time differences sorted by the number of occurrences. An example of some output from the analysis program can be seen in Listing 6.4. Consequently, using the analysis program is one way of finding outliers and errors.

The model-trace generated by Mote B is decompressed to CSW events by a script that maps trace IDs to CSW events. The script parses the trace file tuple by tuple and injects the timestamp for each CSW event. An example of this is in Listing 6.5. The resulting list of CSW events is written to an output file, which is used as input to the automatic analysis script.

```
Max: 14453 microseconds, min: 5911 microseconds
Avg: 7562.1 microseconds, median: 6726 microseconds
Number packets forwarded: 279
 First time: 15,916,307 microseconds - last time: 21,675,163 microseconds
 Forwarded on avg every 20641 microseconds
Number packets received: 413 - on average every 13943 microseconds
```

Listing 6.4: Some output from trace analysis

```
                321451 5
                321455 2
                321462 1
                   |
                   v
SRVENTRY 0 0 321451 0 0 0 service_name 0
QUEUECOND 0 0 321455 0 0 0 service_name notempty
PKTQUEUE 0 0 321455 0 0 0 service_name 0
SRVEXIT 0 0 321462 0 0 0 service_name 0
```

Listing 6.5: Example of decompressing trace to CSW events

The automatic analysis script takes the CSW events as input and generates the signatures for the CSW model as output, as displayed in Listing 6.6. The signatures themselves are similar to function definitions and can invoke each other as long as the invoked signature is defined above the caller, as in the C programming language. A file is created for each signature or signature, and each SEM consists of one or more signatures, depending on the presence of queue or state conditions. A queue condition such as the one in Listing 6.6 causes two different signatures to be defined. The execution environment parses the signature to create a single SEM with branching points in the places where the conditionals are found. Therefore, the conditionals must appear in the same place in the signatures so that the branching point is in the same place. The upper signature is called if packet_queue is not empty and the lower one if it is. When all the signatures are generated, the most significant part of the CSW model is created.

6.2 CSW model

The CSW model is currently implemented as a simple C++ class, which is instantiated with a node object as an argument to the constructor. Figure 6.2 describes a set-by-step process on how the CSW model is executed in ns-3. It starts with the ns-3 simulation (1) setting up the execution environment that executes the CSW model. The execution environment (2) then parses the device file and sets up all events in C++ that represent the CSW. Afterward, the ns-3 simulation (3) invokes FSMs that (4) invoke SEMs set up by the execution environment. As the simulation goes on, (5) triggers in the SEMs invoke callbacks to FSMs in C++. SEMs trigger FSMs and FSMs trigger SEMs until the packet forwarding is finished. Consequently, the execution of the CSW model depends on the protocol models in C++ to invoke SEMs to provide the temporal behavior.

Since the model itself only simulates the temporal behavior of the CSW, it must be integrated with existing ns-3 models for simulation of other behaviors. Currently, only integration with the CC2420 transceiver model described in [IG] is done. With the transceiver model installed under the CSW model, the reception and transmission of packets happen by using existing models, and we satisfy one of the requirements for the CSW model that are defined in Chapter 3. The transceiver model is made to simulate the behavior of the CC2420 radio chip, which is the radio chip of the TelosB mote. When the transceiver model receives a packet, a callback function is invoked that signals the CSW model to start processing the packet. When it is finished processing the packet, it forwards it by using the CC2420 transceiver model's send

```
SRVENTRY 0 0 100000 0 0 0 service_name x
QUEUECOND 0 0 100004 packet_queue packet_queue 0 service_name notempty
PKTQUEUE 0 0 100004 packet_queue packet_queue 0 service_name x
SRVEXIT 0 0 100011 0 0 0 service_name x

SRVENTRY 0 0 200000 0 0 0 service_name x
QUEUECOND 0 0 200004 packet_queue packet_queue 0 service_name empty
SRVEXIT 0 0 200011 0 0 0 service_name x
                 |
                 v
SIGSTART
NAME service_name
PEU cpu
RESOURCES cycles normal
FRACTION 100%  1940 1940

0 START
x                                 PROCESS        4 0
x QUEUECOND packet_queue packet_queue notempty
x DEQUEUE PKTQUEUE 0 packet_queue
x PROCESS 7 0
0 STOP

SIGEND

SIGSTART
NAME service_name
PEU cpu
RESOURCES cycles normal
FRACTION 100%  1940 1940

0 START
x                                 PROCESS        4 0
x QUEUECOND packet_queue packet_queue empty
x PROCESS 7 0
0 STOP

SIGEND
```

Listing 6.6: CSW events to signatures conversion

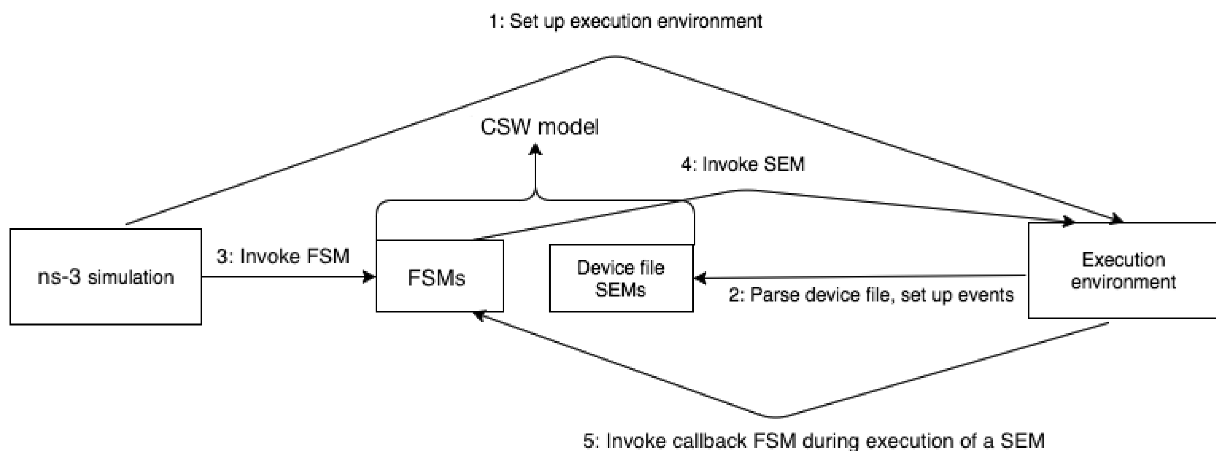


Figure 6.2: ns-3 simulation

packet functionality. Further integration with other models that provide, for instance, IPv6 and 6LoWPAN functionality will be considered in future work.

6.2.1 Device file

The device file is a big part of the CSW model. Most importantly, it contains signatures that represent the temporal behavior of the traced CSW services. It also contains settings such as the number of cores and the speed they operate at, threads, packet and service queues and the capacity of them. Lastly, callback triggers defined in the device file result in functions being called in C++ during simulation. They can be triggered by, e.g., dequeuing packets from a queue or when executing a location in an SEM. Since the device file contains many settings, it is complicated to create it. Aside from the signatures, every setting must be defined manually. In our case, we define a PEU at 4MHz to represent the CPU of TelosB, a thread to represent the task scheduler, and more. Even the signatures are not trivial to add to the device file because they must currently be placed above whichever signatures that invoke them, like functions in the C programming language. Considering all the things that must be defined in the device file, it is the second big part of the CSW model to complete after generating the signatures.

Our TinyOS/TelosB device file includes settings such as a byte queue, a packet queue, one thread, one service queue, a PEU (CPU) and several callback triggers. The byte queue represents the RX queue of the radio chip, which has a limit of 128 bytes. The packet queue is analogous to the IPAQ and has a capacity of three packets. If the queue is full and the model attempts to enqueue a packet, the packet is dropped. As mentioned above, the CPU is represented as a PEU running at 4MHz. The thread is analogous to the scheduler in TinyOS and executes services from the service queue. The TX byte queue of the radio chip does not need to be modeled because it is never overflowed. All these settings are manually configured and are based on the analysis of the CSW of TinyOS in Chapter 5.

Three boolean state variables are defined in the C++ part of the model that can currently not be implemented in the device file. One is used as RX queue overflow indicator. RX queue overflow is currently modeled by the CSW model and not the CC2420 transceiver model because overflow cannot happen unless the temporal behavior is simulated. Without the temporal behavior, the packet is immediately written into memory and thus overflow never happens. The two remaining boolean state variables indicate whether packets are being processed in the receiving and sending part of the CSW. These variables are modeled because only one packet can be processed in the receiving and sending parts at a time. While it would be best if these variables were included in the device file instead of C++, it functions just as well either way.

6.2.2 Forwarding process

The final task in creating the CSW model is to write the C++ FSMs and integrate the model with other models. In our case, the CC2420 transceiver model is wired to the CSW model such that the transceiver sends packets to the CSW model on reception and the CSW model sends packets to the transceiver model on transmission. On reception of a packet, the transceiver model calls the ReceivePacket() FSM in the CSW

model, which can be seen as the entry-point to it. At that point, the scheduler thread in the device file is woken up unless it is already awake. ReceivePacket() starts the receiving part of the CSW which can only be done with one packet at a time. That includes the hardware interrupts for writing the packet into memory (HIRQ-1, HIRQ-2, HIRQ-3, HIRQ-4) and receiveDone_task that sends it to the upper layer protocols. receiveDone_task is enqueued into the service queue by HIRQ-4 and is executed by the task scheduler thread. receiveDone_task enqueues the packet to the IPAQ, which has a capacity of three packets. If there is no room, the packet is dropped.

When the packet is enqueued into the IPAQ, the receiving part ends, and the sending part starts. As with the receiving part, only one packet can be processed at a time. The sending part includes the sendTask task, HIRQ-5 (triggers AttemptSend() FSM), HIRQ-6 and the sendDone_task task. In the AttemptSend() FSM, the transceiver model's Send function is invoked to transmit the packet to Mote C. When the packet is finished transmitting, the sendDone_task SEM triggers FinishedTransmitting() where the packet is removed from the IPAQ, the sending part ends, and the next packet can be sent. If no packet is to be sent or received, the scheduler thread is put to sleep.

Packets can be dropped in three different locations when being processed by the CSW model. First place is due to RX queue overflow which causes the radio chip to receive no data. Second is in HIRQ-4 when having written the entire packet into memory and it has a bad CRC checksum. Third is in receiveDone_task when the IPAQ is full. The first two are unlikely to occur in the real mote when the CC2420 CCA feature is enabled. Then it is usually due to the IPAQ being full. Consequently, we focus mainly on the IPAQ fill-level when evaluating the accuracy of the model.

6.3 Summary

In this chapter, the creation of the CSW model is described. Collected traces are used to create signatures that contain the temporal behavior of the CSW. They are put together in a device file along with other settings that define the core of the model. The knowledge gained from the analysis of the CSW in Chapter 5 is used to configure the device file and write the FSMs. The CC2420 transceiver model is integrated with the CSW model. When it receives packets, an FSM is triggered that invokes SEMs via the execution environment. SEMs add the temporal behavior of the model. The accuracy, scalability, and impact of the CSW model are evaluated in the next chapter.

Chapter 7

Model evaluation

In this chapter, the accuracy, scalability, and impact of the CSW model are evaluated. In Section 7.1, the accuracy is assessed by comparing the performance of it with a real mote. In Section 7.2, the scalability is measured by varying the simulation parameters: number of packets an active node processes, number of seconds a node is idle, and number of active and inactive nodes. In Section 7.3, the impact is assessed by replicating a previously conducted throughput experiment with and without the model. The higher the accuracy, scalability, and impact, the better the model is.

Figure 7.1 puts the metrics used to evaluate the accuracy and scalability of the CSW model in the context of the packet forwarding process. The metrics can be divided into two groups: metrics for the accuracy and scalability of the model, which are explained in detail in Subsection 7.1.2 and 7.2.2, respectively. We do not make specific measurements to evaluate the impact of the CSW model. Instead, the impact is evaluated in all the experiments; the accuracy can reveal how significant the delay is added and the scalability can reveal how useful the model is in large-scale simulations. In Section 7.3, however, the impact is evaluated as a previously conducted throughput experiment is replicated with and without the CSW model to put the model in the context of existing literature.

Listing 7.1 helps explain Figure 7.1 and what intra-OS delay is in the device we model. In an ns-3 simulation where transmission of packets is only performed with the CC2420 transceiver model, the full end-to-end delay only consists of the transmission delay and perhaps some processing delay done by the transceiver. Cooja/MSPSim shows that it takes approximately 98 μs after CC2420 has received a packet before the first hardware interrupt is invoked. We suspect it might be because of the CRC check. That additional delay is currently not added by the CC2420 transceiver model, but it can be added in the future. That minor delay plus the intra-OS delay constitute the full processing delay, also called intra-node delay. Therefore, most of the end-to-end delay is simulated by adding the CSW model to a simulation.

7.1 Accuracy of CSW model

Evaluation of the accuracy of the model requires three experiments: one for the variation in intra-OS delay due to packet size, variation in intra-OS delay due to IPAQ fill-level, and finally, forwarding rate relative to packet size and packet rate. Subsection 7.1.1 describes the parameters that are used in the experiments, Subsection 7.1.2 explains the metrics used to determine the accuracy of the model, Subsection 7.1.3

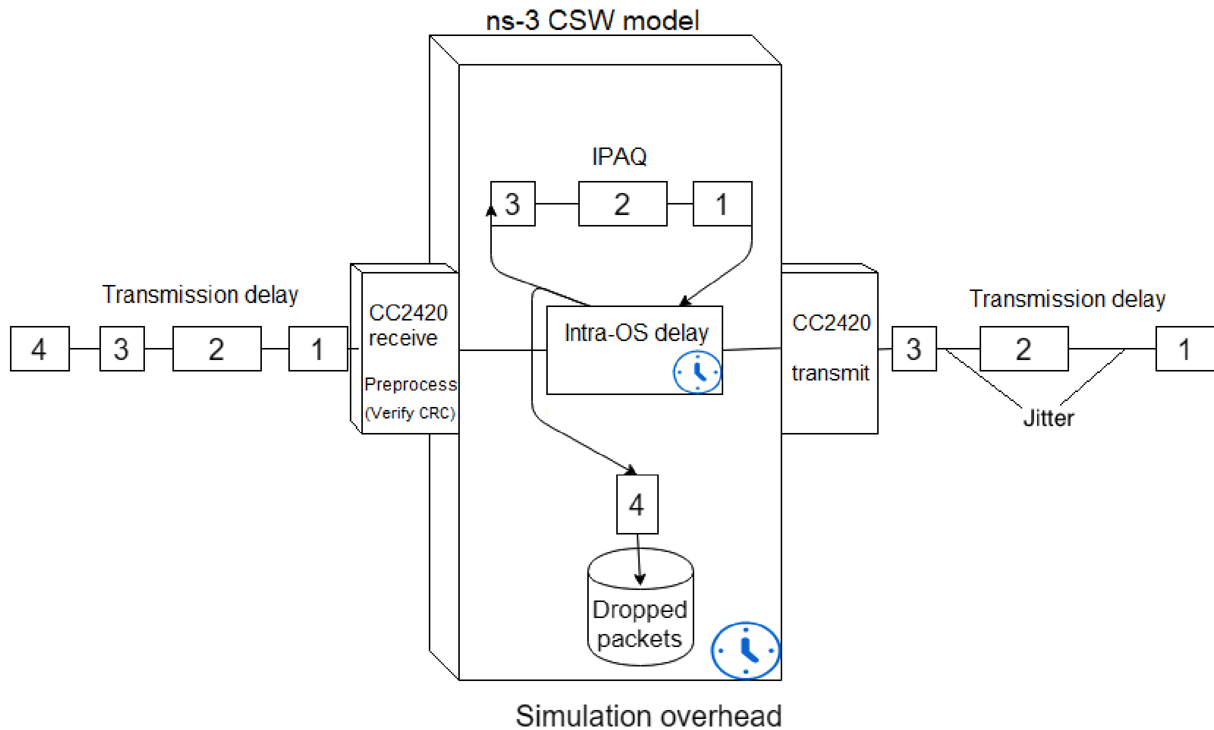


Figure 7.1: Metrics used to evaluate the CSW model and their place in the packet forwarding process.

```
d_trans = transmission delay, decided by packet size.
d_preproc = the delay before the OS is notified of received packet, which is caused
by CC2420 doing some preprocessing. According to Cooja/MSPSim, this delay is 98 μs.
d_ios = intra-OS delay

intra-node delay: d_in = d_preproc + d_ios, delay that intermediate nodes add
Packets are sent from Mote A to C via B. Therefore, transmission delay occurs twice.
ns-3 end-to-end delay with TRx model, without CSW model: d_trans + d_preproc + d_trans
ns-3 end-to-end delay with TRx model, with CSW model: d_trans + d_in + d_trans
```

Listing 7.1: Equations explaining Figure 7.1

describes the experiments conducted, and Subsection 7.1.4 presents and explains the results.

7.1.1 Parameters

The parameters that have been found to affect the performance when the real mote processes a packet are the packet size, IPAQ fill-level, and incoming packet rate. Packet size affects intra-OS delay because some processing stages include copying the packet, which means bigger packets have longer processing times. The IPAQ fill-level affects intra-OS delay because packets that have to wait in the queue have increased intra-OS delay as a result. Incoming packet rate is a parameter that somewhat overlaps with the IPAQ fill-level, but provides a different perspective. As the packet rate increases, more packets are received and must be processed, which results in a higher IPAQ fill-level and eventually packet loss.

All three parameters are used in one experiment each to provoke change in temporal behavior to assess the accuracy of the CSW model. The packet size is the primary parameter used for Experiment 1, which measures how the intra-OS delay increases with the packet size. The IPAQ fill-level is the primary parameter used for Experiment 2, which measures how the intra-OS delay changes by varying the IPAQ fill-level. Experiment 3 uses packet rate as its central parameter and measures how the forwarding rate decreases as the packet rate and packet size increase. Consequently, the most significant parameters found to affect the performance of the forwarding behavior are investigated thoroughly.

7.1.2 Metrics

The *TinyOS*/*TelosB* CSW model needs to prove itself accurate in several ways to represent the packet forwarding behavior of a real TelosB mote. We measure its accuracy in simulating intra-OS delay and relative forwarding rate. If the experiments show that the model has a similar intra-OS delay and relative forwarding rate as the real mote when changing the parameters mentioned before, the model is considered accurate.

The intra-OS delay represents the most critical metric. For instance, it determines other metrics such as packet loss and jitter. As an example, consider Figure 7.1. It shows four packets being sent from Mote A to the intermediate Mote B that forwards them to Mote C. The packets are enqueued into the IPAQ which can contain up to three packets at a time. To be removed from the queue, the packets have to be successfully forwarded. Packets 1, 2 and 3 arrive first and fill up the queue. Packet 4 arrives before any of the other packets have been sent, and there is no room in the queue. As a result, packet loss occurs. When packets 1, 2 and 3 finally are forwarded, some jitter can be seen by the variable space between them, which is due to variable packet size and waiting time in the IPAQ. The intra-OS delay is measured from when the CSW starts writing a packet into RAM to the moment the radio chip attempts to transmit the packet for the first time. If CCA is enabled (which it is here) and the channel is clear, the radio starts to transmit the packet.

Aside from being a metric, the intra-OS delay is also the main feature of the CSW model, and along with transmission delay constitute most of the per-hop packet delay for *TinyOS*/*TelosB*. The CSW model simulates intra-OS delay by scheduling simulation events in the future according to processing stages that are based on traces from a

real mote. Without it, the forwarding mote receives a packet and then immediately forwards it. Two parameters affect intra-OS delay in the CSW model and real mote: variation in the fill-level of the IPAQ and the packet size. The former is naturally simulated because packets waiting in the queue take longer to process. The latter is simulated by varying the delay for some processing stages by the size of the processed packet. As preliminary experiments have shown that intra-OS delay only varies due to packet size and IPAQ fill-level, we believe these experiments are sufficient to evaluate the CSW model's accuracy in simulating intra-OS delay.

The relative forwarding rate is the percentage of successfully forwarded packets, relative to the packet rate and the packet size. As the packet rate increases, the IPAQ is enqueued more often. When the IPAQ is full, all subsequent packets are dropped until there is free space again in the queue as shown in Figure 7.1. An example where packets will be dropped due to intra-OS delay is when the parameters are <packet rate=every 5 ms, packet size=120>. The intra-OS delay of processing a packet of 124 bytes is 14.8 ms (from Table 1.1). Receiving a packet every five ms means that the IPAQ will eventually be filled up and subsequent packets are dropped. If the intra-OS delay and concurrency are modeled correctly, packet loss should be correctly simulated. Concurrency is an issue because only one packet can be processed in both the receiving and sending part at a time. Moreover, the issue of whether the mote is busy processing something or waiting for a component to finish processing is relevant here as well. If the real mote is busy processing in a processing stage, but the CSW model is not, increasing the packet rate might cause the real mote to drop packets where the CSW model will not. Therefore, this metric affects many aspects of the forwarding behavior.

7.1.3 Experiments

Three experiments evaluate the performance of the model as compared to a real mote when varying different parameters. In Experiment 1, we study how the packet size affects the intra-OS delay. In Experiment 2, we explore how the IPAQ fill-level and packet size affect the intra-OS delay. In Experiment 3, we examine how the packet rate and packet size affect the packet drop rate.

The variable packet header size as a result of 6LoWPAN packet header compression affects the packet flow in the experiments. Since the UDP and IPv6 headers are compressed according to the 6LoWPAN protocol [six14], and as a result of the default behavior of setting the hop limit to 1, Mote B increases the packet size by one byte. Therefore, when it is specified in the experiments that Mote A sends a packet to Mote B of the size 36, 80 or 124 bytes, the packet forwarded by Mote B to Mote C has a size of 37, 81 or 125 bytes. Since the difference is only one byte, the effect is small and not expected to affect the results negatively.

All three experiments share some settings. Only two motes are included in them; Mote A sends a packet to Mote B, and Mote B processes the packet to forward it to a conceptual Mote C. Mote C is not necessary to include because the packets that are sent do not require acknowledgment packets. Furthermore, Mote B does not transmit the packets in any of the experiments, aside from Perspective 3 in Experiment 1. The most important reason for that is that the scope of these experiments is to assess the accuracy of the intra-OS behavior of the CSW model, not the transceiver model. Since Mote B cancels the transmission when the radio is ready to transmit, the accuracy of the simulation of intra-OS delay and fill-level of IPAQ are assessed. A final thing about

the experiments is that the tracing framework is used to measure the performance of the real mote in all of them. Since the tracing delay is low and few events need to be traced, the tracing delay is not expected to affect the behavior adversely. In summary, only Mote A and B are included, Mote B does not transmit packets, and the tracing framework is used to capture the behavior of the real mote.

Buffered tracing is used in Experiment 1–2 and instant tracing is used in Experiment 3. These tracing methods are described in Chapter 4. Purpose 2 of the tracing framework is to evaluate the CSW model when accurate timestamps are needed, which is the case for Experiment 1 and 2. Buffered tracing provides accurate timestamps, but the sample size is limited because the buffer can only contain 700 trace events. Purpose 3 is also to evaluate the model, but where accurate timestamps are not needed, which is the case for Experiment 3. The reason why Experiment 3 uses instant tracing is that there is no buffer or limitation to the number of trace events in a run, which is required when the packet forwarding rate is measured since many packets must be processed in many contexts.

Experiment 1

In Experiment 1, we measure how accurately the model simulates intra-OS delay with variation in the size of the processed packets. The variation in packet size is presented in three ways, called Perspective 1, 2 and 3. In Perspective 1, we compare the intra-OS delay when a real and simulated ns-3 Mote B forwards a sequence of packets when varying packet sizes. We expect the intra-OS delay to vary with the packet size, and the accuracy of the CSW model can be evaluated by assessing its similarity to the real mote on a per-packet basis. In Perspective 2, the intra-OS delay of the CSW model is compared to a real mote and an emulated Cooja/MSPSim mote with steadily increasing packet size. We study how the delay increases along with the packet size and how the delay of the model and real mote compare when contrasted with the Cooja/MSPSim mote. Finally, in Perspective 3, we look at the end-to-end delay that is added to the simulation by the CSW model, a real mote, Cooja/MSPSim and the CC2420 transceiver model in ns-3. The end-to-end delay from a real mote compared with only the transceiver model illustrates the need for CSW models; if the difference is significant, the need is accordingly.

Mote B in the experiment is traced to capture the intra-OS delay when it processes packets before forwarding it. That requires three tracepoints: (t1) when having received the packet, (t2) the packet size, and (t3) when ready to transmit it. The intra-OS delay is calculated by $t_{ios} = timestamp(t3) - timestamp(t1)$, and the packet size is captured to categorize the various delays to be able to replicate the sequence of packets in ns-3. As such, only the necessary information is traced to minimize the tracing delay.

The simulation recreates the real testbed runs by using the traces from it. It does so by scheduling the simulated Mote A to send packets to Mote B each time the real mote received one in the testbed experiment, which is possible because each trace tuple is accompanied by a timestamp that describes the number of microseconds that passed since the experiment started. Thus, the intervals between packet reception are the same in the CSW model and the real mote.

Experiment 2

In Experiment 2, we investigate how much the fill-level of the IPAQ affects intra-OS delay. The queue has a maximum capacity of three packets, and packets are enqueued into it a bit more than halfway through the forwarding process. The variation in intra-OS delay of the real mote is compared with the CSW model by artificially enqueueing the same packet three times in the IPAQ instead of once. Then it becomes apparent how much longer it takes for a packet to be processed when it has to wait until other packets are processed before it. Additionally, we observe how long the receiving and sending part of the CSW are, as discussed in Chapter 5. The same three tracepoints as in Experiment 1 are used to measure the intra-OS delay, and it is calculated in the same way. Additionally, the ns-3 simulation reproduces the simulation in the same way as in Experiment 1 by using the trace from the real mote, except that now packets are enqueued three times. The intra-OS delay of the CSW model and real mote can be compared for the different IPAQ fill-level 0–2 to assess the accuracy of the CSW model.

Experiment 3

In Experiment 3, we compare the packet forwarding rate of the CSW model with a real mote in high packet rate scenarios. Packets can drop or not received in the first place because of three reasons: (1) the CC2420 RX queue is overflowed which causes packets not to be received, (2) packets dropped because of bad CRC checksum, and (3) full IPAQ and subsequent packets are dropped. Number 3 is expected to be the reason why packets drop in a real mote when CCA is enabled, which is the case with us. When CCA is disabled, collision causes packets to be damaged and fail the CRC check. CC2420 RX queue overflow can happen in any case, but at a higher rate when CCA is disabled. If the forwarding rate of the CSW model and real mote are similar, this experiment is a success.

Tracing the forwarding rate requires three tracepoints: when (t1) starting to write a packet to RAM, (t2) bad CRC checksum, (t3) flushing the RX queue of the radio and (t4) when ready to transmit the packet. The reason why the RX queue being flushed is traced is that it happening means the RX queue has had an overflow, and there might be some packets that are not received. That is less likely to happen since CCA is enabled, but the tracepoint is there in case it happens. Thus, all tracepoints needed to calculate the forwarding rate are included. The following equation calculates the forwarding rate.

$$fwrate = \left(\frac{count(t4)}{count(t1)} \cdot 100\right)\%$$

As Mote A's application and the ns-3 simulation send packets at various sizes and rates, the forwarding rate during a wide range of parameter combinations is measured. In contrast to Experiment 1 and 2, the simulated Mote A does not use a real trace to send packets at the same time as the real experiment. Instead, Mote A sends packets at the specified packet rates, which results in a much more consistent packet rate than the real mote can achieve. The packet rates that are used are 40–130 packets per second and the application decreases the interval between packet transmissions by 250 μs for every 256 packets that are sent until 130 packets per second is reached. That process is performed for each of the UDP payload sizes 24, 40 and 80 bytes. The result is 72 different packet rates for each packet size, which is 216 parameter combinations and

55296 packets in total. The final trace contains the sub-traces for all combinations, and since there is a two-second break in between them during the experiment, we can create a trace file for each of the combinations afterward. The percentage of forwarded packets in each combination becomes a point on the resulting graph. If the actual packet rate is significantly lower than the attempted packet rate, the data is omitted from the results. To sum up this complex experiment, many different contexts are executed to assess the CSW model's accuracy in simulating packet loss.

The relative forwarding rate of a real mote and the CSW model are compared by plotting the percentage of successfully forwarded packets in different <packet size, packet rate> parameter combinations. The expectations for the accuracy of the forwarding rate are not high since the experiment is conducted at high packet rates and the behavior becomes increasingly non-deterministic. Nevertheless, the packet forwarding rate is expected to start decreasing at around the same packet rates for the CSW model and the real mote.

7.1.4 Results

In this section, the results from running the experiments that measure the accuracy of the CSW model are presented.

The traces collected from Mote B have timestamps from a binary timer, which is how TinyOS works as mentioned in Chapter 2. That means the microsecond clock used to trace the mote has 1,048,576 cycles per second instead of 1,000,000. For the most part, the difference is negligible. As the number of cycles increases, however, the difference becomes more prominent. Table 1.1 shows that it takes around 15 milliseconds to process a packet of size 124 bytes. If that number is gathered from the microsecond clock, that is not 15 milliseconds, but 14.3 milliseconds. While the difference is significant, it has little impact on the big picture. Results from Perspective 1 in Experiment 1 (Figure 7.2) and Experiment 2 (Figure 7.4) have binary time instead of decimal, and the rest have decimal time.

Experiment 1

Figure 7.2 displays Perspective 1 of how the packet size affects the intra-OS delay of packet forwarding. The CSW model and a real mote process ~250 packets the UDP payload sizes [0,8,16...80,88] bytes. As the real mote traces the packet size, the ns-3 simulation replicates that same packet sequence. These results show that the mote and CSW model have an almost identical intra-OS delay when varying the packet size. The delay varies from 7.4 to 15.7 binary ms, and one can see twelve different intervals between points along the y-axis that are directly related to the UDP payload sizes.

Figure 7.3a shows results from Perspective 2. We see how the intra-OS delay (y-axis) increases linearly with the packet size (x-axis) for a real mote, the CSW model, and an emulated Cooja/MSPSim mote. Both the real mote and CSW model are represented by the blue graph because their data is indistinguishable. Furthermore, one can see how different an emulated Cooja/MSPSim node behaves compared to a real mote and the CSW model. The Cooja/MSPSim graph starts at <x=0, y=6.1ms> and the graph combining the real mote and CSW model starts at <x=0, y=7.1ms>. The Cooja/MSPSim graph ends at <x=88, y=10.6ms> and graph combining the real mote and CSW model ends at <x=88, y=15.1>. Thus, Cooja/MSPSim starts at 14% and ends

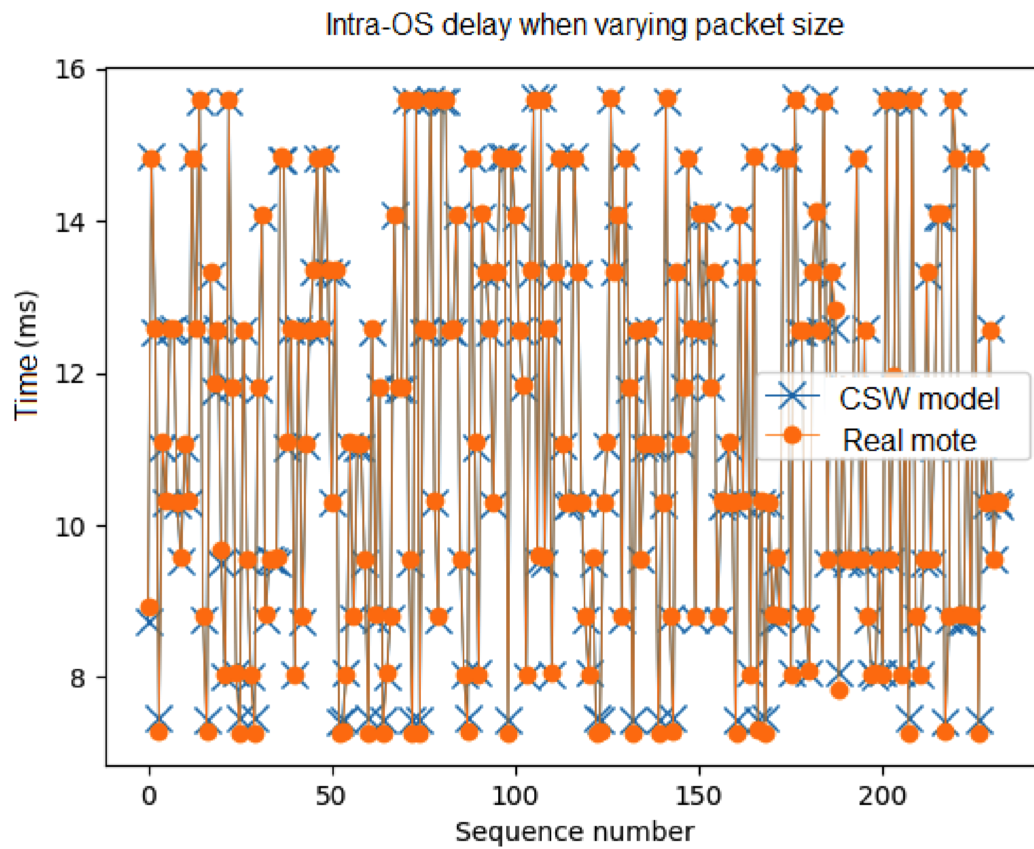


Figure 7.2: Intra-OS delay comparison between real mote and ns-3 with the CSW model at 40 pps. The intra-OS delay is in binary ms.

at 30% less intra-OS delay than the real mote and CSW model.

Figure 7.3b shows Perspective 3 of measuring how the intra-OS delay increases with the packet size. The packet size (x-axis) affects the total end-to-end delay (y-axis) for a real mote, CSW model, CC2420 transceiver model and an emulated Cooja/MSPSim node. All the graphs increase linearly with the packet size. We contrast the intra-OS delay with the CC2420 transceiver model's transmission delay to see how significant the intra-OS delay is. The transmission delay includes the time it takes for Mote A to send to Mote B, and for Mote B to send to Mote C. The CC2420 transceiver model graph starts at <x=0, y=2.3ms> and ends at <x=88, y=7.9ms>. The Cooja/MSPSim graph starts at <x=0, y=8.3ms> and ends at <x=88, y=19.5ms>. The real mote and CSW model graph starts at <x=0, y=9.46ms> and ends at <x=88, y=23ms>.

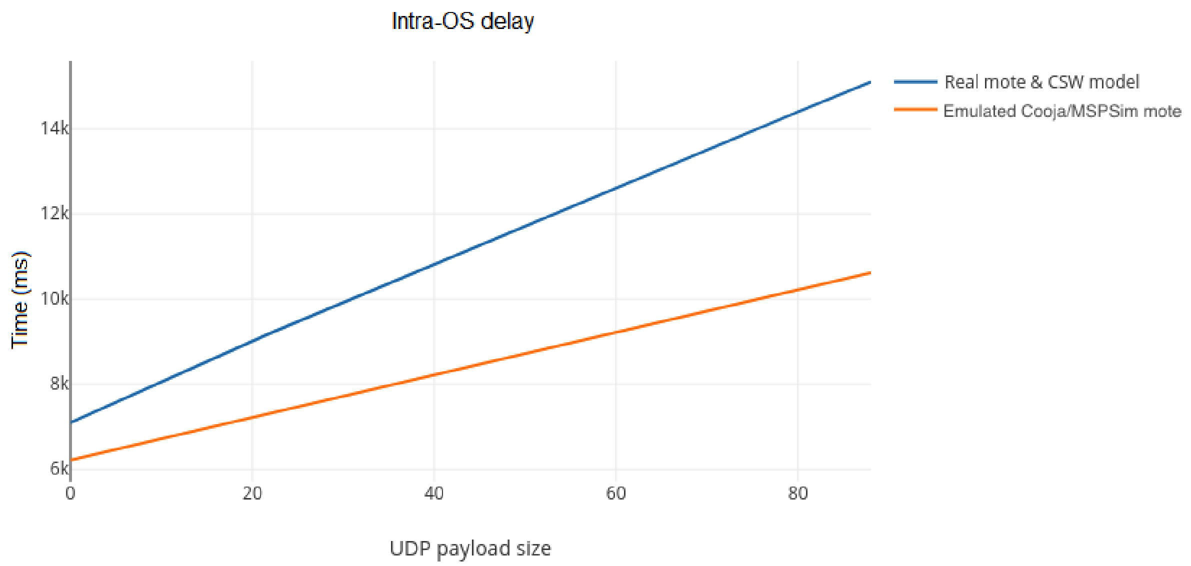
Experiment 2

Figure 7.4 compares the variation in intra-OS delay due to IPAQ fill-level for a real mote with the CSW model. The intermediate Mote B receives 25 packets of variable sizes at a low packet rate with each their sequence number (x-axis). Since the packets are enqueued three times into the IPAQ, the total intra-OS delay is measured for the packets when the IPAQ contains zero, one and two packets. One can distinguish between the delay before enqueueing the packet into the IPAQ (receiving part) and afterward (sending part), which are discussed in Chapter 5.

Listing 7.2 helps to make sense of the gaps between the graphs for understanding which delay belongs to which part. The intra-OS delay for a packet that is placed in an empty IPAQ only consists of the processing delay caused by executing the receiving and sending parts once; the same receiving and sending parts as can be seen in Figure 5.2. When a packet has to wait in the IPAQ for one packet, the intra-OS delay is the same as when the IPAQ is empty plus the processing delay of sending the enqueued packet. When the IPAQ fill-level is two, the intra-OS delay consists of the processing delay when the IPAQ is empty plus the intra-OS delay of sending two enqueued packets.

Experiment 3

Figure 7.5 compares the relative packet forwarding rate of the CSW model with a real mote when sending packets of three different sizes at different rates. The x-axis represents the packet rate and y-axis the percentage of successfully forwarded packets. In both the case of the real mote and CSW model, the percentage of successfully forwarded packets decreases as the packet rate increases beyond a certain point, depending on the packet size. It depends on the packet size because the larger the size is, the lower the maximum packet rate is. The behavior of the CSW model mirrors the most significant parts of the real mote. Although the real mote starts dropping packets a bit earlier in all three cases compared to the CSW model, the trends of both graphs are the same.



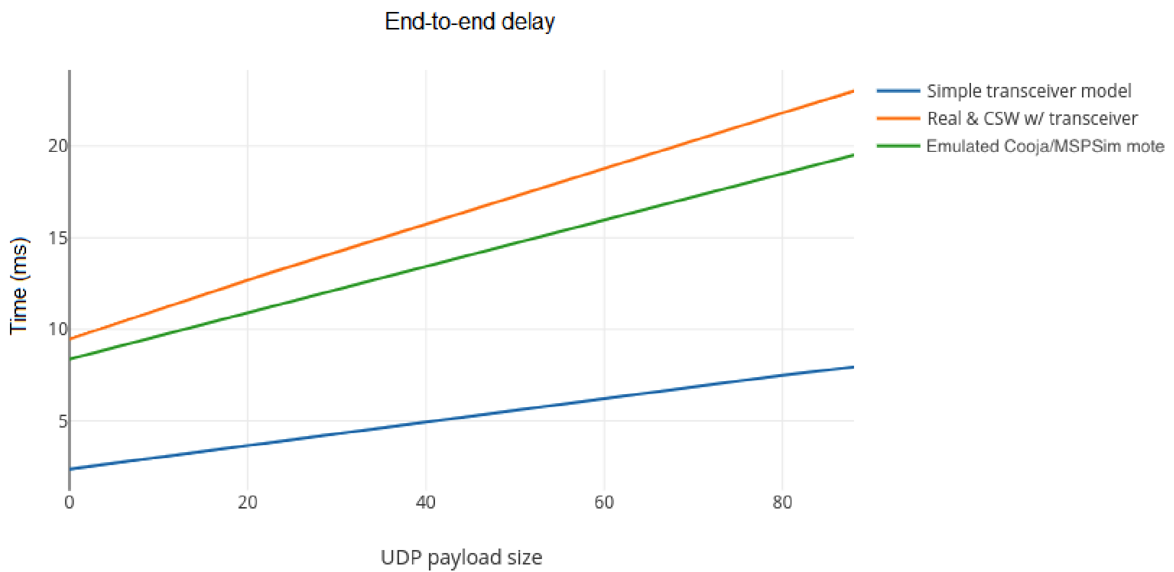


Figure 7.3: Intra-OS delay based on UDP payload size

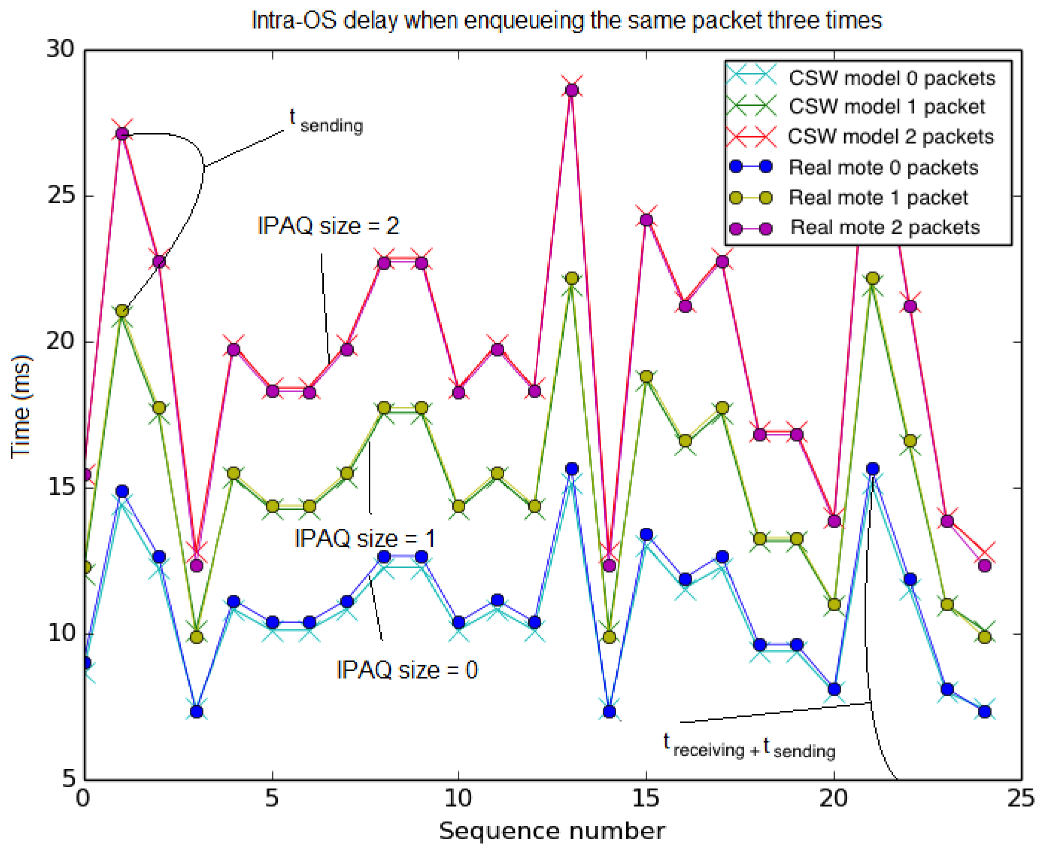


Figure 7.4: Variation in the IPAQ fill-level affecting the intra-OS delay. The intra-OS delay is in binary ms.

```
The following equations are true when packets being enqueued have the same size:

t = intra-OS delay of processing a packet when IPAQ is empty
```

$t_{ipaq0} = t = t_{receiving} + t_{sending}$

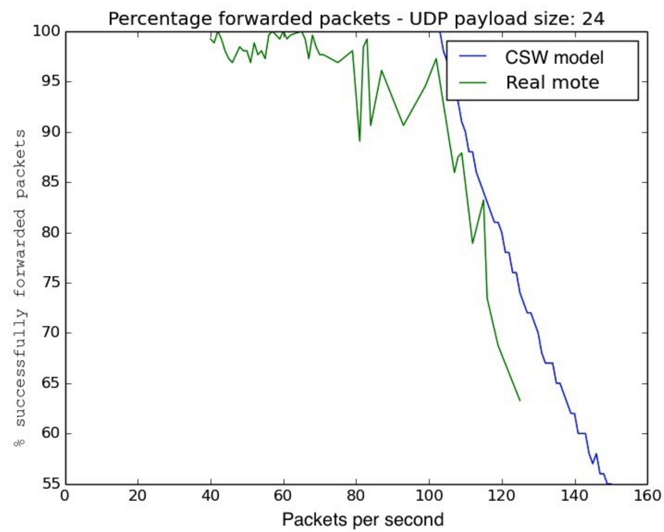
$t_{ipaq1} = t_{ipaq0} + t_{sending}$

$t_{ipaq2} = t_{ipaq1} + t_{sending}$

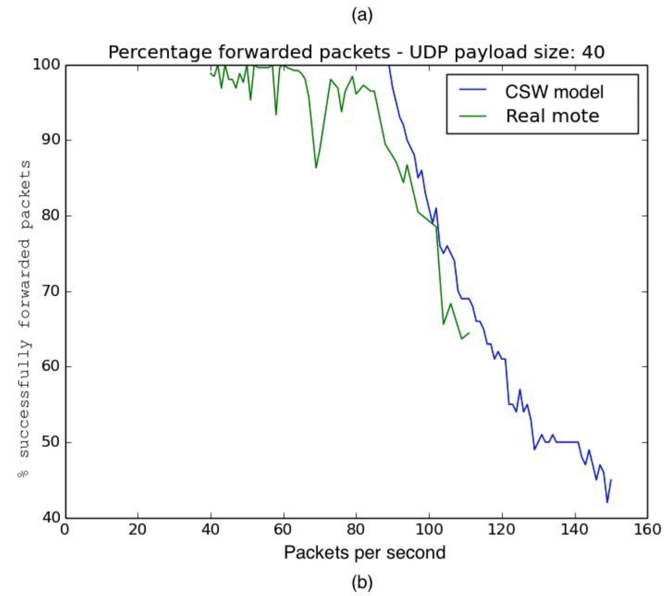
$t_{sending} = t_{ipaq2} - t_{ipaq1}$

$t_{receiving} = t_{ipaq0} - t_{sending}$

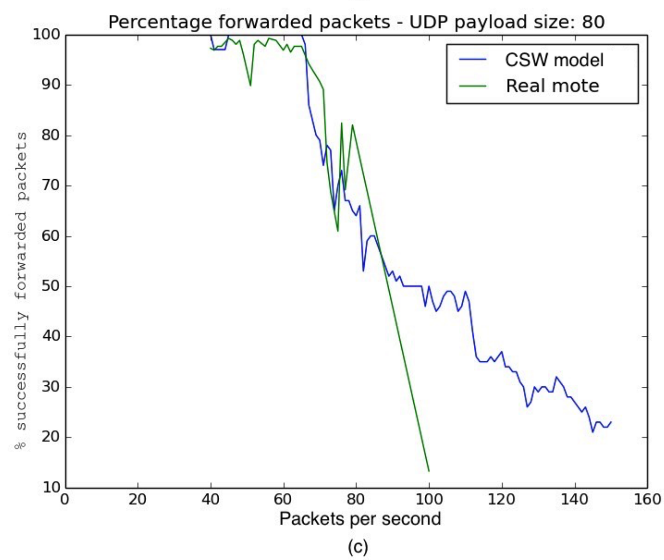
Listing 7.2: Equations explaining Figure 7.4



(a)



(b)



(c)

Figure 7.5: The forwarding rate when sending packets at various packet rates

7.2 Scalability of CSW model

Scalability of the CSW model is measured by increasing the number of nodes that execute it, the number of packets it processes and the number of seconds it simulates. Simulation overhead includes the memory consumption of nodes and the added execution time of the simulation when adjusting parameters. The model must be scalable because WSNs may contain up to several thousand nodes.

7.2.1 Parameters

Four parameters are used for Experiment 4 and 5, as discussed in Chapter 3. First is the number of packets that a node forwards. Second is the number of simulated seconds that a node is inactive. The third is the number of active nodes each processing one packet. Fourth is the number of inactive nodes each being idle for ten million seconds. By separating these groups and varying different parameters, we find out if a large-scale simulation is feasible. Experiment 4 uses the first and second group for its parameters, and Experiment 5 uses the third and fourth group. Therefore, every parameter that can affect the scalability of the CSW model is addressed in the following experiments.

7.2.2 Metrics

The metrics we use to evaluate the scalability of the model include the time added to the simulation execution time because of nodes that execute the CSW model and their memory consumption. For all four groups mentioned above, the execution time added to the simulation by varying the parameters are measured. Only for group four is the memory consumption measured.

Time added to the simulation execution time should be low. The importance of reducing the time depends on what is causing it. If the added time is because of a node that processes packets, it is less important than the time added by a node being idle. The reason is that WSNs can contain thousands of nodes and they are idle most of the time, which means even a small amount of delay added per simulated second can be devastating for scalability. An inactive node should in principle not process anything at all and only start processing when a hardware interrupt wakes it up and starts processing. When the node has finished processing the packet, the threads in its CSW model should be put to sleep. On the other hand, it must not take too long for a node to process a packet, or else the CSW model cannot be used for anything.

As the number of nodes that execute the CSW model increases, the time they add to the simulation execution time and memory consumed by them are also expected to increase. The time does not necessarily increase linearly. If the total node computation accumulates with the number of nodes, a small per-node simulation overhead can make large-scale exponential in complexity, and thus, infeasible. That is one of the things to investigate, and therefore, we make a distinction between the time added by a single node and several nodes. The memory consumption of the CSW model is quantified by using the htop Linux program to count the MBs that the simulation allocates as the number of inactive nodes increases. It must be possible to simulate up to a few thousand nodes on a non-super computer because that is an expected number of nodes in a WSN.

7.2.3 Experiments

In Experiment 4 and 5, we assess the scalability of the CSW model. Experiment 4 is designed to measure the execution time added when an active node and inactive node execute the CSW model. In Experiment 5, execution time and memory consumption are measured when varying the number of nodes. Both of the experiments are executed on a Linux 14.04 LTS PC with 16 GB RAM and 4.2GHz CPU frequency.

Experiment 4

Experiment 4 is designed to measure the execution time added by a single active and inactive node. An active node processes packets and an inactive node is idle. The execution time added by an active node is measured on a per-packet basis, and an inactive node per simulated second it is idle. The time added by an inactive node is measured by running the inactive node for many simulated seconds. The active node sends ten packets per simulated second for several simulated seconds to see how the simulation execution time increases as more packets are processed, and afterward, the time added during the inactive periods is subtracted from the time to find out how much time is added because of packet processing. In addition to plotting the results from running an inactive node in ns-3, we compare the execution time when simulating the CSW model with an emulated Cooja/MSPSim node and a real mote.

Experiment 5

In Experiment 5, we measure the execution time added when varying the number of nodes to observe how the time added to the simulation execution time and memory consumption increase with the number of nodes that execute the CSW model. The main point of the experiment is to see if the complexity concerning simulation execution time and memory consumption is linear or exponential as the number of nodes increases. As the number of active nodes increases, it is expected that the execution time increases almost the same as in Experiment 4 when increasing the number of packets that a single node processes. When increasing the number of inactive nodes, we expect the simulation execution time to increase almost the same way as in Experiment 4 when increasing the number of seconds that a single node is idle.

We also measure memory consumption of inactive nodes. That can be used to estimate how many nodes can execute with the CSW model on an average PC. The additional memory consumption of nodes when they are active and perform packet processing is expected to be relatively low, but it is currently not possible to reliably measure because of memory leak bugs that cause the memory consumption to accumulate when it should remain static.

7.2.4 Results

In this section, the results from Experiment 4 and 5 are presented and explained.

Experiment 4

Figure 7.6a illustrates a linear increase in the time (y-axis) added to the simulation execution time as the number of packets that are processed (x-axis) increases. The idle time has been subtracted from the numbers, so the additional execution time is only caused by the packet processing. Since the CSW model only uses a single thread that is asleep when no service is executing, the simulation execution time is the same regardless of the packet size. At x=10k, the execution time is approximately 4 seconds.

Figure 7.6b illustrates a linear increase in the simulation execution time (y-axis) as the number of simulated seconds (x-axis) an inactive node spends increases. The graph starts at <x=0, y=0> and ends at <x=6B, y=1 sec>, which means that 6 billion simulated seconds for an inactive node that executes the CSW model in ns-3 takes one second of real time to simulate.

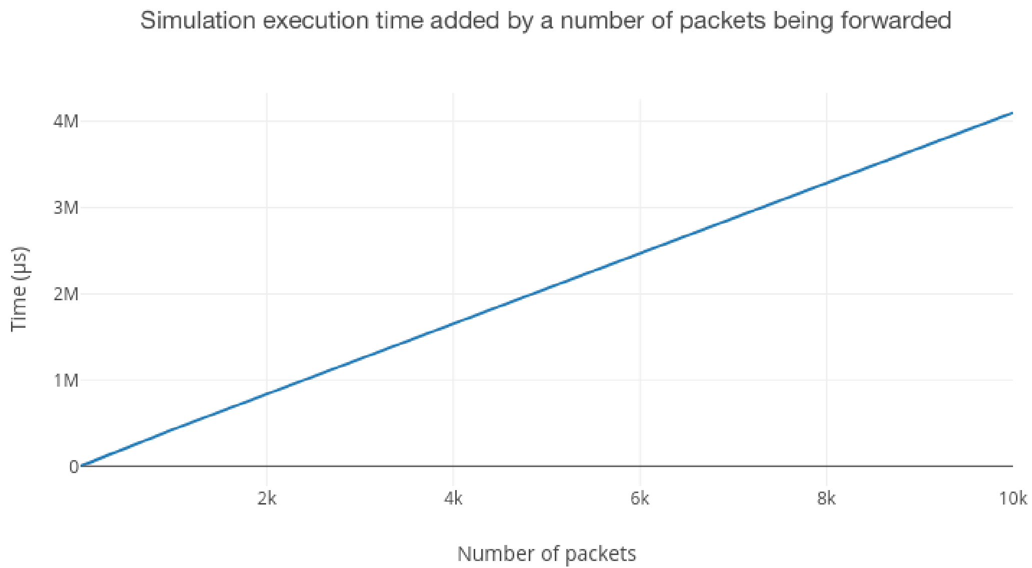
Real mote	Cooja/MSPSim	CSW model
1+e6 sec	263 sec	22.5 ms
1+e7 sec	2630 sec	24.5 ms
1+e8 sec	26300 sec	39.8 ms
1+e9 sec	263000 sec	178 ms
6+e9 sec	1578000 sec	1 sec

Table 7.1: Comparison of execution time when comparing a real mote, an emulated Cooja/MSPSim mote (extrapolated data) and the CSW model in ns-3.

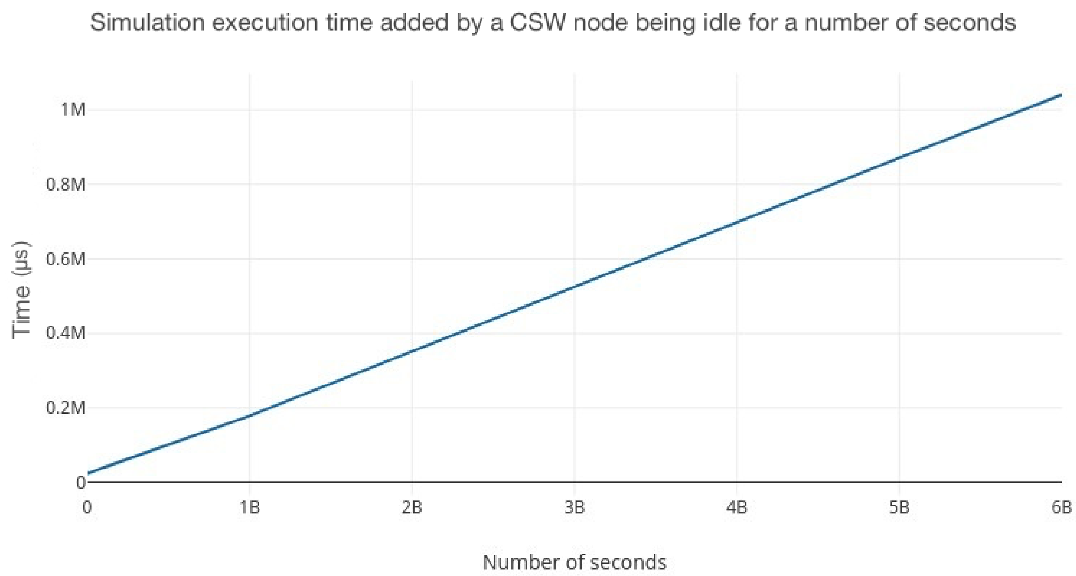
Table 7.1 illustrates the execution time of running inactive nodes for x simulated/executed seconds and comparing a real mote, an emulated Cooja/MSPSim node and the CSW model in ns-3. A real mote takes x seconds to execute x seconds being idle. The emulated Cooja/MSPSim mote takes 3–4 orders of magnitude less amount of time to simulate/execute than a real mote. The reason why it takes much less amount of time is that Cooja/MSPSim processes an application as fast as possible and speeds up time, whereas the real mote cannot do that and must process the entire time. The data from the Cooja/MSPSim mote is extrapolated from a 263-second long simulation. The ns-3 CSW model takes approximately six orders of magnitude less amount of real-time to simulate x seconds than an idle emulated Cooja/MSPSim mote when the simulation time is more than one billion seconds. ns-3 takes much less amount of time than the real mote because it processes the application as fast as possible and speeds up the time. It is also much faster than an emulated mote in Cooja/MSPSim because ns-3 is much simpler than the MSPSim emulator, which is the software that emulates the TelosB/Sky mote in Cooja.

Experiment 5

Figure 7.7a, 7.7b, and 7.7c illustrate how the simulation execution time and RAM usage (y-axis) increase linearly as the number of nodes (x-axis) increases. Figure 7.7a shows the simulation execution time when a number of CSW nodes process one packet each. The graph start at <x=0, y=0>, and ends at <x=20k, y=12M>. Figure 7.7b illustrates a linear increase in simulation execution time as the number of nodes being idle for ten million simulated seconds increases. The graph starts at <x=10k, y=25M>, and ends at



(a)



(b)

Figure 7.6: Time added to simulation execution time because of an active and inactive node

<x=100k y=310M>. Figure 7.7c shows how the memory consumption increases linearly as the number of installed nodes increases. The graph starts at <x=10k, y=1GB> and ends at <x=100k, y=11.6GB>.

7.3 Impact of CSW model

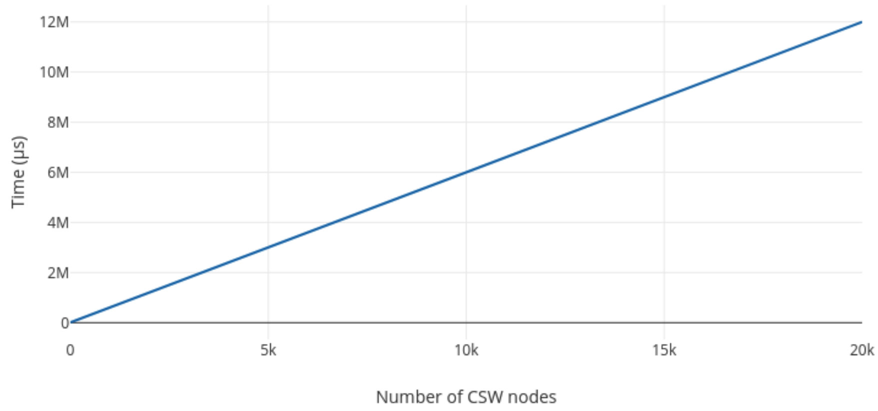
One of the thesis aims is to evaluate the impact the CSW model has on a regular simulation scenario, and we do that by conducting Experiment 6 that replicates an experiment described in the literature [IG] that evaluates a CC2420 transceiver model in ns-3. Experiment 6 measures the throughput of the model when transmitting packets at various packet rates. By introducing the CSW model, the capacity of the intermediate mote is expected to decrease. If the throughput decreases significantly, it is a confirmation that the CSW model has a significant impact.

7.3.1 Experiment 6

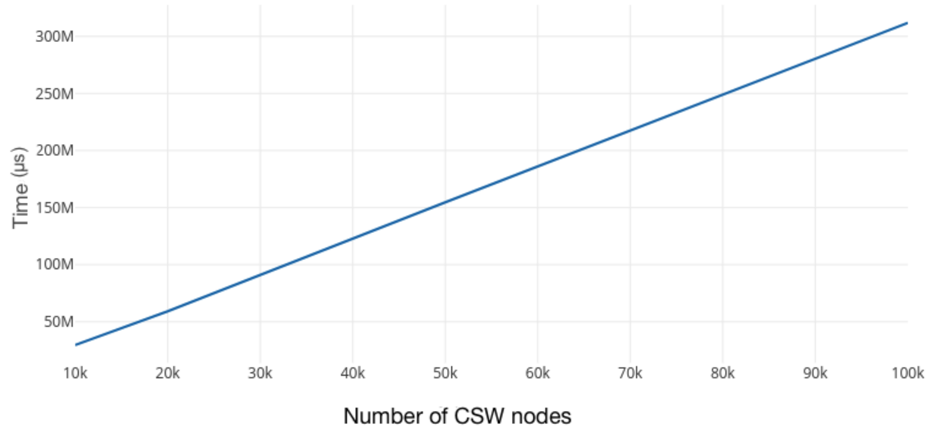
Igel et al. [IG] present results from experiments that evaluate the CC2420 transceiver module. In the original experiment, Igel et al. measure the capacity of the transceiver by adjusting the size of the packets and the rate at which they are sent. Three simulations are executed in the original experiment: (1) data-rate and packet size are OK, (2) data-rate too high but packet size OK and (3) data-rate OK but packet size too big. The results demonstrate that when data-rate is too high, not all packets are successfully transmitted, and when packet size is too big, no packets are successfully transmitted.

Experiment 6, which we conduct, uses a three-node topology instead of the two-node topology that the original experiment has. The reason is that the CSW model simulates the intra-OS behavior of the intermediate mote that performs packet forwarding. As a result of this change, the data-rate must be lowered significantly because Mote A and Mote B both send packets in turn, whereas if only Mote A is sending, it can send data all the time at 250kbps. The highest data-rate possible where both Mote A and B can send packets interchangeably is 100kbit per second, which is a bit less than the half of the max data-rate at which the transceiver can transmit (250kbit per second). Where the authors in [IG] refer to *application data-rate* as the data-rate without UDP, IP and IEEE 802.15.4 MAC headers and relative to the application payload size, we refer to data-rate as the rate at which the transceiver sends data.

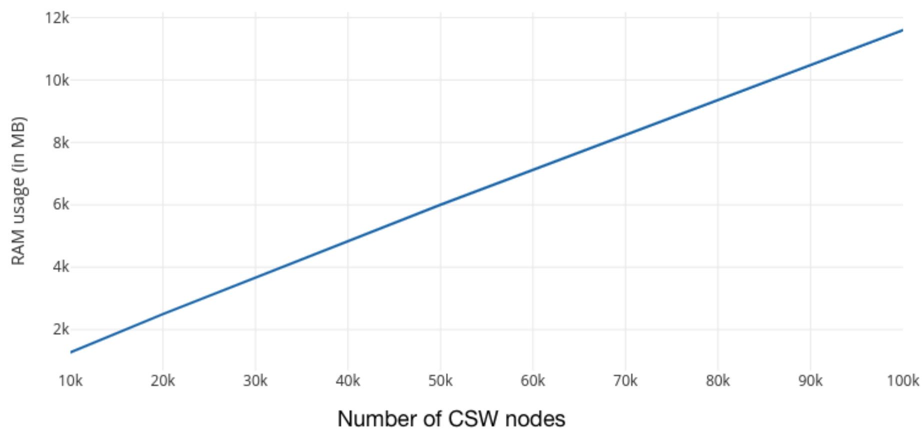
Experiment 6 runs three times for packets of 124 bytes. Run 1 is without the CSW model, Run 2 with the CSW model at the same rate as Run 1 and Run 3 is the lowest rate at which packets drop. Listing 7.3 contains the final 6–7 output lines from executing Experiment 6. In Run 1, Mote A sends packets at 100kbps, and all the packets are forwarded successfully by Mote B to Mote C. That is evident since the final total RX and total TX are equal. In Run 2, Mote A sends packets again at 100kbps, although this time with the CSW model. Last total RX in the run is 28024, and last total TX is 49848, which means the forwarding rate is only 56%. In Run 3, Mote A sends packets at only 65kbps and with the CSW model. Since the last total RX is less than total TX, it means there is still packet loss. Last total RX is 28148 bytes, and total TX is 32364 bytes, which means the forwarding rate is 87%. If the data-rate is 64kbps, all packets are successfully received by Mote C. Since 100kbps is the highest data-rate that the application without



(a)



(b)



(c)

Figure 7.7: Simulation execution time and memory consumption when increasing the number of nodes

the CSW model can run at, the throughput is reduced to 64% when the CSW model is added.

```
Run 1 (100kbps, 124 bytes packet size, without CSW model, 100% packets received by Mote C)

7.973s Mote A sent 124 bytes total TX 49600 bytes
7.97691s Mote C received 124 bytes total RX 49476 bytes
7.98292s Mote A sent 124 bytes total TX 49724 bytes
7.99183s Mote C received 124 bytes total RX 49600 bytes
7.99284s Mote A sent 124 bytes total TX 49848 bytes
7.99675s Mote C received 124 bytes total RX 49724 bytes
8.00917s Mote C received 124 bytes total RX 49848 bytes


Run 2 (100kbps, 124 bytes packet size, with CSW model, 56% packets received by Mote C)

7.978s Mote A sent 124 bytes total TX 49600 bytes
7.97873s Mote C received 124 bytes total RX 27776 bytes
7.98792s Mote A sent 124 bytes total TX 49724 bytes
7.99168s Mote C received 124 bytes total RX 27900 bytes
7.99784s Mote A sent 124 bytes total TX 49848 bytes
8.00913s Mote C received 124 bytes total RX 28024 bytes


Run 3 (65kbps, 124 bytes packet size, with CSW model, 87% packets received by Mote C)

7.95274s Mote A sent 124 bytes total TX 32116 bytes
7.96301s Mote C received 124 bytes total RX 27900 bytes
7.968s Mote A sent 124 bytes total TX 32240 bytes
7.97711s Mote C received 124 bytes total RX 28024 bytes
7.98326s Mote A sent 124 bytes total TX 32364 bytes
7.99756s Mote C received 124 bytes total RX 28148 bytes
```

Listing 7.3: Output from Experiment 6 simulation runs

7.4 Analysis

In this section, the results from the CSW model accuracy, scalability and impact experiments are analyzed. Subsection 7.4.1 analyzes and discusses the results that relate to the accuracy. Subsection 7.4.2 analyzes the results that relate to the scalability. Finally, Subsection 7.4.3 analyzes the replication experiment to assess the impact.

7.4.1 Accuracy of CSW model

The results of running the accuracy experiments prove that the CSW model accurately simulates intra-OS delay, how it varies with the packet size and IPAQ fill-level, and packet loss due to high incoming packet rates. This section provides an analysis of the results.

Packet size affecting intra-OS delay

The results of Experiment 1 demonstrate the drastic effect that the packet size has on the intra-OS delay. UDP packets can have a packet size ranging from 36 to 124 bytes, which results in intra-OS delay from 7.4 binary ms to 15.7 binary ms. Figure 7.3a also illustrates how different the intra-OS delay of an emulated Cooja/MSPSim mote is from a real mote and the CSW model. Investigation and analysis of traces from Cooja/MSPSim and a real mote confirm that the most deviating simulated delays are when writing a packet from the CC2420 RX queue to RAM and writing a packet

from RAM to the CC2420 TX queue. That explains how the gap between the emulated Cooja/MSPSim mote and real mote/CSW model increases as the packet size increases; that the per-byte delay in packet delay is lower in the emulated Cooja/MSPSim mote than in a real mote.

Figure 7.3b illustrates how significant the intra-OS delay is compared to the full end-to-end delay. CC2420 can send packets at 250kbit per second, so packets with size 36 and 124 bytes are sent in 1.2 and 4 ms. The end-to-end delay involves the packets being sent twice, once from Mote A to B and the other from Mote B to C. Therefore, the end-to-end transmission delays range from 2.5 to 8 ms. The intra-OS delay is much higher than the transmission delays and ranges from 7.1 ms to 15.1 ms. By not simulating the intra-OS delay, a significant part of the end-to-end delay is missing.

To sum up, the results from Experiment 1 are significant because of two things. First, the variation in packet size causes such a significant variation in intra-OS delay. Second, the CSW model is accurate when compared to a real mote. The high variation in intra-OS delay due to variation in packet size means that some processing stages must vary their delays by the size of the processed packet, and the CSW model enables that. As emulation in Cooja/MSPSim is unable to simulate those processing stages accurately, it is not better to emulate the temporal behavior of CSW with Cooja/MSPSim than to use the simpler CSW model in ns-3. As we discuss in Section 8.3, we do not know precisely why Cooja/MSPSim exhibits a slightly different temporal behavior than a real mote.

IPAQ fill-level affecting intra-OS delay

The results of Experiment 2 demonstrate the effect the IPAQ fill-level has on the intra-OS delay. In this experiment, the same packet is enqueued three times to the IPAQ instead of once, and thus, sending the same packet three times. To make sure that the data only shows the intra-OS delay that is added as a consequence of waiting, the packets are never transmitted across the medium to the next hop, since this would impose additional waiting time not related to packet processing in the OS. Packets that have to wait for other packets have increased intra-OS delay, which can be seen in Figure 7.4. The CSW model accurately simulates the variation in IPAQ fill-level, as is evident by the points overlapping with those for the real mote.

Listing 7.4 extrapolates the equations from Listing 7.2. The latter listing explains Figure 7.4 The incoming delay means that the packet has to wait in the IPAQ for another packet to be received and placed in the IPAQ before it is sent. When a packet must wait for two packets ahead in the IPAQ to be sent and also wait for two packets to be received and placed in the IPAQ, that is the maximum amount of time that a packet must wait in the IPAQ when packets are not dropped and applications are not actively processing. That delay is the same as performing the receiving and sending parts of the CSW three times each. In summary, the equations in Listing 7.4 describe what is almost the worst-case intra-OS delay.

To sum up, the results of Experiment 2 are significant because (1) the high variation in intra-OS delay when varying the IPAQ fill-level and (2) the CSW model accurately simulates it. As a continuation from Experiment 1, the first point further illustrates the need for a CSW model because the variation in delay is not trivial to simulate. The second point means that the CSW model accurately simulates intra-OS delay

```
The following equations are true when packets being enqueued have the same size:

t = intra-OS delay of processing a packet when IPAQ is empty
```

$$t_{ipaq2incoming0} = t_{ipaq1} + t_{sending}$$
$$t_{ipaq2incoming1} = t_{ipaq2incoming0} + t_{receiving}$$
$$t_{ipaq2incoming2} = t_{ipaq2incoming1} + t_{receiving}$$

Listing 7.4: Projected intra-OS delay if packets that are ahead in the IPAQ must wait for new packets to be placed in it. Continuation of equations in Listing 7.2

when varying both the IPAQ fill-level and packet size. The reason why the CSW model can simulate the variation correctly is that the CSW model breaks the CSW into smaller parts. If the CSW model is too simple and only provides a constant delay, as some related processing delay models do, this variation is not simulated accurately. Therefore, this experiment illustrates not just the need for processing delay models, but also models that are as detailed as our CSW model.

Relative forwarding rate

Results from conducting Experiment 3 illustrate a decrease in the percentage of successfully forwarded packets as the packet rate reaches beyond a certain point, depending on the packet size. The graphs of the CSW model and real mote in Figure 7.5 show declines in the same contexts also when changing the packet size. That is proof that the model accurately simulates packet loss. The real mote starts to drop some packets a bit earlier than the CSW model, but the big dip where the packet drop rate increases consistently with the packet rate is accurate.

Although the experiment is conducted with only a single mote sending packets, packets start to drop in Mote B because it cannot handle processing packets fast enough. That means if an application needs to send several packets from Mote A to C via B, and it sends them as fast as it can, it is likely that packets will drop in Mote B as a result. The reason why this does not normally happen in TinyOS is that a CCA feature is enabled by default that causes the mote to back off a random number of milliseconds before each packet is sent. That does, however, not entirely explain why Mote B prevents packet loss, because the random backoff is used for it as well.

To understand why the random backoff prevents packets being dropped in Mote B, we go back to the analysis of the CSW in Chapter 5. The CSW consists of two parts, the receiving and sending parts. Each part can process one packet at a time. The sending part includes the radio transmission, and if backoff is enabled, the waiting period before transmission as well. In both the radio transmission and backoff periods, the mote is free to process a packet in the receiving part. Table 1.1 illustrates how insignificant the transmission delay is compared to the intra-OS delay. As the receiving part takes a bit more than half of the full intra-OS delay (excluding transmission delay), a packet cannot be fully processed in the receiving part before the transmission of one is finished. By adding the random backoff, however, it appears that the mote can perform the receiving part of the CSW in the backoff waiting and transmission time and thus avoids packet loss.

The results from Experiment 1 and 3 can be compared to give some insight. To process packets of UDP payload sizes 80, 40 and 24 bytes, takes 14.9 ms (67 pps), 10.3

ms (97 pps) and 9.6 ms (104 pps). The results from Experiment 3 illustrate that, indeed, the big drop in packet forwarding rate starts in both the CSW model and real mote when the packet rate increases above 67 pps, 97 pps and 104 pps when the size of the UDP payload is 80, 40 and 24 bytes. It indicates that if the CSW receives more packets than the number it can forward, the IPAQ will fill up which eventually leads to packet loss.

To sum up, the results of Experiment 3 are significant because the CSW model starts to drop packets at the same combination of packet size and packet rate. The CSWMM focuses mainly on intra-OS delay and only packet loss as a consequence of it. The reason why this is not expected to be realistically simulated is that the behavior of devices becomes non-deterministic at high packet rates. To our benefit, the model accurately simulates packet loss.

7.4.2 Scalability of CSW model

The point with the scalability experiments is to push the capacity of a regular PC to its limits to demonstrate potential bottlenecks in the CSW model. Results from Experiment 4 illustrate a linear increase in simulation execution time both when increasing (1) the number of packets that an active node processes and (2) the number of simulated seconds that an inactive node is idle. Results from Experiment 5 show how the simulation execution time and memory consumption increases linearly as the number of nodes that are idle or send packets increases. As such, the experiments prove that the model is scalable even for large-scale simulations.

Figure 7.6a and 7.7a illustrate the simulation execution times when processing packets. The size of the packets does not matter because the same number of events execute, just that three processing stages vary their processing times by the packet size. A larger packet size causes a higher intra-OS delay, but a discrete-event simulator such as ns-3 is not affected by time, only the number of events executed. In [KPG13a], the results indicate a difference in execution time because of the packet size. That is likely to be because their CSW model is multi-threaded and requires a scheduler that gives each thread a time quantum, which results in simulation events to be scheduled consistently as the simulation proceeds. Therefore, the time added to the simulation execution time is the same regardless of the packet size in our CSW model.

Table 7.1 compares the time it takes to run a real idle mote with an emulated inactive node in Cooja/MSPSim and an inactive node in ns-3 that runs the CSW model. The results indicate, unsurprisingly, that there is a sharp distinction between the execution time by them. Although emulation with Cooja/MSPSim takes much shorter to execute than a real mote, it is still resource intensive to execute. If scalability is an issue, the best alternative is ns-3 with or without the CSW model. An emulated Cooja/MSPSim mote takes longer to execute than a node in ns-3 with the CSW model because it runs the actual hardware instructions of the compiled OS whereas the ns-3 CSW model only provides the temporal behavior aspect of it.

To sum up, the results from Experiment 4 are significant because (1) the time added to the simulation execution time is tiny per processed packet and per simulated second with idle nodes and (2) that the simulation execution time increases linearly with the number of packets and simulated seconds. The results from Experiment 5 are significant for two reasons. First, the time added to the simulation execution time, and memory consumption per node is low enough that 100,000 nodes can be instantiated

and the simulation only occupies 11.6GB of RAM. Second, the simulation execution time and memory consumption increase linearly with the number of nodes. Since the simulation execution time is small and increases linearly, the CSW model can be used for large-scale simulations.

7.4.3 Impact of CSW model

The purpose of Experiment 6 is to assess the impact the CSW model has when introducing it to simulations by replicating an experiment conducted previously without the model. We measure the possible as 124 bytes packets are sent with and without the CSW model. Three runs are executed, one without the CSW model and two with it. The first two runs are at the same data-rate, 100kbps. When not including the CSW model, all packets arrive at Mote C. When including it, however, almost half of all packets do not arrive at Mote C. That is an indication that the CSW model does impact a typical simulation. The final run decreases the data-rate to 65kbps with the CSW model to show that even then, not all packets arrive at Mote C. The data-rate must be reduced to 64kbps before all packets arrive at Mote C when including the CSW model. That means including the CSW model reduces the throughput to 64% of what is possible when not including it. In conclusion, the CSW model has a discernable impact when it is added to a WSN simulation.

While Experiment 6 has the task of determining the impact of the CSW model, all the other experiments and results contribute to it. Preliminary results in Table 1.1 at the beginning of the thesis describe how the intra-node delay is an essential part of the temporal behavior of intermediate nodes in WSNs. The accuracy experiments (Experiment 1–3) indicate how accurate the model is and show how significant the delay added is. The following scalability experiments (Experiment 4–5) show how the CSW model can be used in large-scale WSN applications on regular PCs. Finally, Experiment 6 replicates a previously conducted experiment to put the model in the context of existing literature.

To sum up, the results are significant because they question the correctness of similar results in the published literature. By introducing the CSW model, the throughput reduces to 64% of what is possible without it. As the forwarding scenario in the experiment is a typical one, adding the CSW model to the simulation in ns-3 is bound to affect the performance in most scenarios. Thus, if throughput is important for a WSN application to function correctly, including the CSW model will have a significant impact.

7.5 Summary

This chapter presents the evaluation of the CSW model, which is done by assessing its accuracy, scalability, and impact. The results indicate that the behavior of the model is similar to a real mote, the scalability is high, and the impact of adding it to a simulation is significant.

Part III

Conclusions

This thesis aims to discover the significance of CSW models for WSN and create one that adds realistic software execution delay to ns-3. We believe that we have successfully reached that goal, and showed that in Chapter 7.

Chapter 8

Conclusion

In this thesis, we set out to with the aim to create a CSW model of $^{TinyOS}/_{TelosB}$ that adds realistic temporal behavior when doing packet forwarding in ns-3. As part of the process of creating the model, a tracing framework is created and used to capture the temporal behavior of the CSW both for creating and evaluating the model. This chapter summarizes the contributions: the tracing framework and CSW model. Section 8.1 discusses the final CSW model and Section 8.2 discusses the tracing framework. Section 8.3 describes some of the limitations of our methods. Section 8.4 explains some of the tasks that are not accomplished in this thesis that can be worked on in the future.

8.1 CSW model

In Chapter 1, the claim is made that CSW models are needed to simulate the temporal behavior of WSN devices accurately in ns-3, with Table 1.1 as motivation. After having conducted the experiments that measure the accuracy of the CSW model in Chapter 7, we suggest that Figure 7.3 supports the claim because of how significant the full end-to-end delay with the real mote is when compared with only the CC2420 transceiver model. Furthermore, Experiment 6 replicates a throughput experiment initially conducted in [IG], and the throughput reduces by 46% when including the CSW model. The significant reduction is a sign that the intra-OS delay is non-negligible. Therefore, we have much evidence to suggest the need for CSW models.

This CSW model is more impactful than the previous models because the modeled device is simple and slow. TelosB is a single-core device and TinyOS a single-threaded OS. Furthermore, TelosB does not use a cache for faster memory access, the CPU speed of TelosB is constant at 4MHz, and TinyOS does not use optimization techniques in the instrumented drivers that can cause variable temporal behavior either. On the other hand, the Google Nexus One device with a CPU frequency of 245MHz – 1GHz that runs a multi-threaded OS, is modeled in [KPG13b]. In [Oys16], the dual-core Galaxy Nexus with a CPU frequency of 1.2GHz is modeled. These are much more complex devices compared to TelosB, which makes it easier to demonstrate the impact of this model.

The accuracy of the CSW model is high. There is almost no difference between the CSW model and real mote in the tested scenarios. Several aspects of the device are modeled and assessed in experiments: (1) intra-OS delay, (2) variation in delay due to packet size, (3) variation in delay due to IPAQ fill-level, and (4) packet loss. Experiment 1 focuses on Point 1–2, Experiment 2 focuses on Point 1 and 3, and Experiment 3

focuses on Point 4. Point 1 is implemented since the intra-OS delay is the main feature of the CSW model. Point 2 is implemented in the CSW model because of three processing stages which vary their processing times by the packet size. Point 3 is included in the model because packets that wait in the IPAQ for other packets take longer to be sent. Point 4 occurs as a result of all other elements of the CSW model being implemented correctly. That includes the points above plus the restriction that only one packet can be processed in the receiving and sending part of the CSW at a time. Therefore, we show that the model is accurate in multiple ways.

These experiments do more than prove that the model is correct. They also help us adjust and correct the model by revealing errors. The relative forwarding rate experiment revealed such an error. When a packet is being written into RAM or the TX queue of CC2420, it can look like some of the processing stages are non-blocking I/O events. Meaning, it takes time to write a packet into RAM and the TX queue, but the mote can perform other tasks in the meantime. That is untrue, however, and the experiment revealed that because the CSW model managed a much higher packet rate compared to the real mote. Therefore, we adjusted the model by converting all the non-blocking I/O processing stages to regular processing stages in which the mote is busy processing all the time. The result is that the CSW model drops packets in the same contexts as the real mote. That is one example of how the experiments helped us adjust the model.

Experiment 3 in Chapter 7 illustrates that if Mote A sends packets at a high rate to C via B, Mote B drops packets when the IPAQ is full. The noteworthy thing is that only Mote A sends packets to B, which means that a mote that needs to send many packets must deliberately slow down to avoid packet loss in the intermediate mote. That does not happen when using unmodified TinyOS because it includes an initial backoff feature that causes a random waiting time before sending each packet. With initial backoff enabled, packets do not drop. Figure 8.1 shows the intra-OS delay for 124 bytes packets that are sent from Mote A to C as fast as possible with initial backoff enabled. As a result of the backoff, no packets are dropped, and the intra-OS delay is higher than it would if the initial backoff were disabled.

The scalability of the CSW model proves to be high for any typical WSN scenario. We evaluate the significance of the scalability by comparing our results with the previous proofs of concepts in [KPG13a, Oys16]. Although the devices run at completely different CPU speeds (245MHz – 1GHz, 1.2GHz, 4MHz), that does not affect the simulation execution time because ns-3 is a discrete-event simulator in which the simulation execution time only depends on the number of events that are executed and their complexity. That means the scalability of a CSW model depends on how complex the model is. In the first proof of concept [KPG13a], a CSW model processing 10,000 packets takes almost 10 seconds both when the packet size is 50 and 1450 bytes. As the number of packets increases, the difference in execution time between the packet sizes increases. In [Oys16], it takes 6 seconds to simulate 10,000 packets of 162 bytes, which is less than [KPG13a]. Our CSW model takes around 4 seconds for one CSW node to process 10,000 packets, regardless of the packet size. The reason why the packet size does not matter and that the execution time is lower than the other models is that TinyOS is a single-threaded OS, which means no scheduler is needed to make sure that all threads get to run once in a while. In essence, all three CSW models are scalable when increasing the number of packets processed, considering the low simulation execution times.

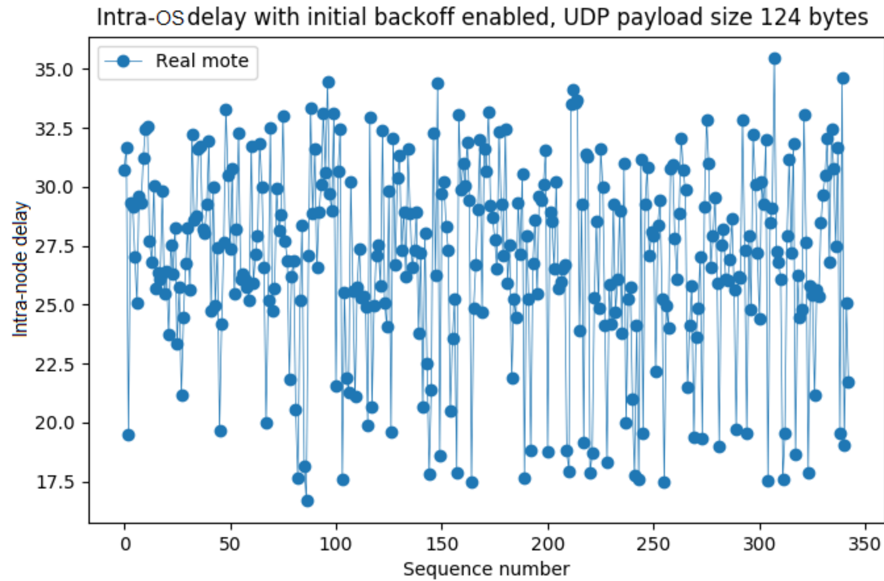


Figure 8.1: 124-byte packets sent as fast as possible with initial backoff in TinyOS enabled, no packets are dropped.

In addition to scalability on a per-packet basis, we also assess the scalability on a per-simulated-second and per-node basis. The simulation execution time increases linearly with all the parameters, which means it is scalable. In addition to the simulation execution time, memory consumption is also measured when increasing the number of inactive nodes. The result is that 100,000 nodes can be executed in the simulation at the expense of 11.6 GB RAM, which we consider scalable. Essentially, when increasing any parameters, whether it is the number of packets processed, the number of simulated seconds being idle, or the number of nodes, the simulation scales adequately.

We have discovered some issues when we emulated with Cooja/MSPSim. We discovered two bugs and inaccurate execution times of two processing stages. The first bug is that the microsecond clock on TelosB displays approximately four times larger value than it should when compared to both the millisecond clock and the Cooja simulation clock, seen in Figure 8.2. The microsecond time does not correspond with the millisecond time nor the Cooja/MSPSim simulation time. The second bug is that the CC2420 CCA feature does not work. When the channel is not clear, packets are still sent, which causes collisions to occur. The inaccurate execution times are when writing a packet from CC2420 RX queue to RAM and from RAM to CC2420 TX queue. These processing stages are too short compared to a real mote, which is the main reason why the Cooja/MSPSim intra-OS delays in Figure 7.3 are different from the real mote and CSW model. In Section 8.3, we discuss how the processing stage times might not be inaccurate, but that MSPSim emulates a different mote with same components. These kind of inaccuracies and bugs are not uncommon for complex software such as the MSP430 emulator MSPSim. One problem might be that the more complex a simulator is, the more error prone it is as well. Therefore, general-purpose network simulation with either Cooja (not emulation) or ns-3 can be better for a more light-weight, transparent and maintainable simulator.

While we have created a CSW model to accurately simulate the temporal behavior

Figure 8.2: Timer issue when emulating in Cooja/MSPSim.

of a WSN device in ns-3, it does not mean that everyone should start using ns-3 for simulation of WSNs. When to choose ns-3, emulation with Cooja/MSPSim or a real testbed to run a WSN depends on the requirements for it. Table 8.1 illustrates some of the suitable application areas for each platform. If it must be large-scale, ns-3 is the best option, Cooja/MSPSim the second best and a real testbed the third best. If the temporal behavior must be accurate, a real testbed is best, CSW model is arguably the second best, and Cooja/MSPSim the third best. The CSW model is second best because the results in Chapter 7 show it. When needing to debug a network application, Cooja/MSPSim is the best, real testbed the second best and ns-3 with the CSW model is not an option because it cannot execute OS code unless extended with an emulator such as TOSSIM [ROM10]. As such, different requirements for an application need different platforms.

Application area	**CSW model**	**Cooja/MSPSim**	**Real testbed**
Debug application in TinyOS	Low	High	Medium
Forwarding on a large scale	High	Medium	Low

Table 8.1: Comparing use cases for the CSW model in ns-3, an emulated Cooja/MSP-Sim mote and a real testbed.

8.2 Tracing framework

The tracing framework is used to capture the CSW behavior of *TinyOS*/*TelosB* and proves to be flexible, efficient with low tracing delay and memory consumption for each trace event. It is flexible because of the two methods of tracing called buffered and instant tracing. The former enables saving trace events with accurate timestamps in RAM until it is full and the latter enables indefinite tracing, albeit with inaccurate timestamps. If someone attempts to model another WSN device in the future, the same design can be reused.

The difference between the tracing framework in this thesis and the ones used in the three previously created CSW models is that the TelosB mote requires compression of traces because it only has 10kB RAM in total with approximately 3.5kB available when the OS and forwarding application are installed. When the traces are collected, they are decompressed to CSW events. Two benefits of this way of tracing are that (1) we can focus entirely on tracing efficiently without worrying about the meaning of the traces and (2) a single trace tuple can be converted to several CSW events, which means even less tracing delay and memory consumption.

As far as the CSWMM is concerned, the compression of traces adds a small step in between the tracing and automatic analysis steps. The step includes decompressing the trace to a list of CSW events that can be used as input to the automatic analysis script. Additionally, the compressed trace can be analyzed more easily, which can reveal errors that occur during execution.

8.3 Research limitations

A problem about the CSW model is that both the creation and evaluation of it rely on the tracing framework that is also created in this thesis. The alternative to using the tracing framework to evaluate the model is to eavesdrop on the packets sent to and from the mote. Tracing the intra-OS delay when listening from the outside is not possible because the captured delay also includes delay caused by the CC2420 radio transceiver. The potential problem about the creation and evaluation being done using the tracing framework is that if the tracing turns out to be inaccurate, it could invalidate the positive evaluation of the CSW model. Nevertheless, it is a simple tracing framework and has proven to be consistent, aside from the timestamp errors that we use a trace analysis tool to mend.

All the experiments are performed without a Faraday cage for two reasons. The first one is that it is expensive and takes extra work setting up the experiments. The second reason is that it is not strictly needed to measure the accuracy of the CSW model because the data-rate is only 250kbit and the packet rate is never over 130 packets per second in the relative forwarding rate experiment in Chapter 7. WLANs [80216] do, however, operate on the same 2.4GHz spectrum as the CC2420 transceiver [Tex14], which means interference from WiFi networks is possible during experiments.

The results from Table 1.1 and Figure 7.3a and Figure 7.3b might unrightfully put the accuracy of the emulator MSPSim that Cooja uses to emulate Tmote Sky/TelosB in a bad light. We do not know the reason why the results are different. We model MTM-CM5000MSP/TelosB/Tmote Sky, which might be a different hardware model from the Tmote Sky emulated in MSPSim. As such, it is hard to tell what causes the difference

in intra-OS delay, and we do not have enough evidence to suggest it is because of an inaccuracy in MSPSim.

If the model turns out to be different, creating a similar CSW model where some of the processing times are slightly different from the model we created is easy to do. The benefit of the CSW model is that it can be modified with low effort to reflect differences between hardware models. Therefore, it might not be necessary to create an entirely new model of TinyOS/MICAz in the future, but rather slightly modify the TinyOS/TelosB CSW model.

8.4 Future work

While the CSW model is accurate and can be useful in many cases, one thing that remains is to create a module for it that has the same format as other ns-3 modules. Currently, the model is specialized for a specific simulation program. Creating a module would enable others to use it and contribute to the framework. The CSW model currently does not support 6LoWPAN fragmentation. In the future, it should be added since 6LoWPAN is used in TinyOS and there is a 6LoWPAN protocol library in ns-3.

We have only compared it to the other models created with the CSWMM and not the solutions mentioned in Section 2.4. That is because it requires extensive research to make a proper comparison between this solution and existing solutions. For future work, however, it might be interesting to investigate how this CSW model holds up to other models that also attempt to add a more realistic simulation of processing delays.

Creating a model of Contiki/TelosB by using a tracing framework with the same design as ours can be interesting. It is likely to demand less effort since the hardware is the same. The services defined in the CSW model will be different since the OS is different. Furthermore, the processing stages will take longer or shorter compared to TinyOS, and the packet queue(s) within the OS will be different from the IPAQ in TinyOS.

The tracing framework can be improved to compress the buffered trace tuples further from five to three bytes and in some cases two bytes, as explained in Section 4.2. That results in the ability to trace more events. It is, however, not prioritized in this thesis because it only helps to minimize the memory consumption, which is already sufficiently low for our purposes with the current solution. If it decreased the tracing delay significantly, it might have been prioritized more.

References

[80216] Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, Dec 2016.

[BBB+10] Thomas Begin, Alexandre Brandwajn, Bruno Baynat, Bernd E. Wolfinger, and Serge Fdida. High-level approach to modeling of observed system behavior. *Performance Evaluation*, 67(5):386 – 405, 2010.

[BRE+15] A. Beifuß, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle. A study of networking software induced latency. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–8, March 2015.

[Cora] Core Working Group. Schedulers and Tasks.

[Corb] Cory Sharp, Martin Turon, David Gay. Timers.

[Dava] David Gay, Philip Levis, David Culler and Eric Brewer. nesC 1.3 Language Reference Manual.

[Davb] David Moss, Jonathan Hui, Philip Levis and Jung Il Choi. CC2420 Radio Stack.

[DDHW08] A. R. Dalton, S. Dandamudi, J. O. Hallstrom, and S. K. Wahba. A testbed for visualizing sensornet behavior. In *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, pages 1–7, Aug 2008.

[DTDM12] M. P. Durišić, Z. Tafa, G. Dimić, and V. Milutinović. A survey of military applications of wireless sensor networks. In *2012 Mediterranean Conference on Embedded Computing (MECO)*, pages 196–199, June 2012.

[DWDH09] Andrew R. Dalton, Sally K. Wahba, Sravanthi Dandamudi, and Jason O. Hallstrom. Visualizing the runtime behavior of embedded network systems: A toolkit for tinyos. *Science of Computer Programming*, 74(7):446 – 469, 2009.

[F.] F. L. LEWIS. Wireless Sensor Networks.

[FK11] Muhammad Omer Farooq and Thomas Kunz. Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930, 2011.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.

[Goo02] B. Goode. Voice over internet protocol (voip). *Proceedings of the IEEE*, 90(9):1495–1517, Sep 2002.

[HC09a] M. Hammad and J. Cook. Lightweight deployable software monitoring for sensor networks. In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6, Aug 2009.

[HC09b] Mustafa Hammad and Jonathan Cook. Lightweight monitoring of sensor software. 01 2009.

[HHKK04] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, June 2004.

[HM06] Jane K. Hart and Kirk Martinez. Environmental sensor networks: A revolution in the earth system science? *Earth-Science Reviews*, 78(3):177 – 191, 2006.

[HMM16] Christopher Hepner, Steffen Moll, and Roland Muenzner. Influence of processing delays on the voip performance for ieee 802.11s multihop wireless mesh networks: Comparison of ns-3 network simulations with hardware measurements. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTOOLS'16, pages 86–95, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[iee12] Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans) amendment 1: Mac sublayer. *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pages 1–225, April 2012.

[IG] Anuschka Igel and Reinhard Gotzhein. A cc2420 transceiver simulation module for ns-3 and its integration into the feral simulator framework.

[Joa] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Osterlind, Thiemo Voigt. MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards.

[KLP+09] Kevin Klues, Mike Chieh-Jan Liang, Jeongyeup Paek, Răzvan Musăloiu-E., Philip Levis, Andreas Terzis, and Ramesh Govindan. Tosthreads: Thread-safe and non-invasive preemption in tinyos. ACM, November 2009.

[KPG13a] S. Kristiansen, T. Plagemann, and V. Goebel. Extending network simulators with communication software execution models. In *2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–10, Jan 2013.

[KPG13b] Stein Kristiansen, Thomas Plagemann, and Vera Goebel. Modeling communication software execution for accurate simulation of distributed systems. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 67–78, New York, NY, USA, 2013. ACM.

[KSKT09] M. Korkalainen, M. Sallinen, N. Kärkkäinen, and P. Tukeva. Survey of wireless sensor networks simulation tools for demanding applications. In *2009 Fifth International Conference on Networking and Services*, pages 102–106, April 2009.

[LAW08] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 344–355, April 2008.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[LMP+05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Mem] Memsic Inc. TelosB Datasheet.

[Mic05] Michael Karl. A Comparison of the architecture of network simulators NS-2 and TOSSIM. January 2005.

[Muh07] Muhammad Hamad Alizai, Olaf Landsiedel adn Klaus Wehrle. Accurate Timing in Sensor Network Simulation. 2007.

[MZ12] Bartosz Musznicki and Piotr Zwierzykowski. Survey of simulators for wireless sensor networks. 5:23–50, 09 2012.

[NFR08] Paulo Neves, Joel Fonseca, and Joel Rodrigues. Simulation tools for wireless sensor networks in medicine - a comparative study. pages 111–114, 01 2008.

[ODE+06] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pages 641–648, Nov 2006.

[Oys16] Oystein Dale. Modeling, analysis, and simulation of communication software execution on multicore devices. 2016.

[PBM+04] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. Atemu: a fine-grained sensor network simulator. In *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, pages 145–152, Oct 2004.

[PE08] 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.

[PPFS12] M. Prats, J. Pérez, J. J. Fernández, and P. J. Sanz. An open source tool for simulation and supervision of underwater intervention missions. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2577–2582, Oct 2012.

[PSC05] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 364–369, April 2005.

[Ras13] Rashid Hussian, Sandhya Sharma, Vinita Sharma and Sandhya Sharma. WSN Applications: Automated Intelligent Traffic Control System Using Sensors. *International Journal of Soft Computing and Engineering (IJSCE)*, 3, 2013.

[RH10] George F. Riley and Thomas R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[ROM10] Laurynas Riliskis, Evgeny Osipov, and Miklós Maróti. Tos-ns3: a framework for emulating wireless sensor networks in the ns3 network simulator. 01 2010.

[RSM14] Alok Ranjan, Himanshu Sahu, and Prasant Misra. A survey report on operating systems for tiny networked sensors. 1, 05 2014.

[RWR+14] Torsten Runge, Florian Wohlfart, Daniel Raumer, Bernd Wolfinger, and Georg Carle. Validated model-based performance prediction of multi-core software routers. 2:1–12, 03 2014.

[SCH+05] Ahmed Sobeih, Wei-Peng Chen, J. C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang. J-sim: a simulation environment for wireless sensor networks. In *38th Annual Simulation Symposium*, pages 175–187, April 2005.

[SEZ09] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Lightweight tracing for wireless sensor networks debugging. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 13–18, New York, NY, USA, 2009. ACM.

[SEZ10] Vinaitheerthan Sundaram, Patrick Eugster, and Xiangyu Zhang. Efficient diagnostic tracing for wireless sensor networks. 9:169–182, 01 2010.

[six14] 6LoWPAN demystified. 2014.

[SK15] Manuel Schiller and Alois Knoll. Emulating vehicular ad hoc networks for evaluation and testing of automotive embedded systems. In *Proceedings of*

the 8th International Conference on Simulation Tools and Techniques, SIMU-Tools '15, pages 183–190, ICST, Brussels, Belgium, Belgium, 2015. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[SKLD+11] Haoyue Sundani, Vijay K Li, Mansoor Devabhaktuni, Prabir Alam, Bhattacharya , Harsh Sundani, Haoyue Li, Vijay Devabhaktuni, Mansoor Alam, and Prabir Bhattacharya. Wireless sensor network simulators: A survey and comparisons. 2, 01 2011.

[SSFM11] R. Sauter, O. Saukh, O. Frietsch, and P. J. Marrón. Tinylts: Efficient network-wide logging and tracing system for tinyos. In *2011 Proceedings IEEE INFOCOM*, pages 2033–2041, April 2011.

[Ste13] Stein Kristiansen. *A Methodology to Model the Execution of Communication Software for Accurate Simulation of Distributed Systems*. 2013.

[Ste17] Stein Kristiansen, Vera Goebel, Karl Øyri and Thomas Plagemann. Event modeling and processing to simplify real-time analysis of physiological signals. 2017.

[Tex14] Texas Instruments. 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. 3 2014.

[TLP05] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 477–482, April 2005.

[YMG08] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292 – 2330, 2008.

[ZBG98] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 154–161, May 1998.

[ZH08] Huiyu Zhou and Huosheng Hu. Human motion tracking for rehabilitation—a survey. *Biomedical Signal Processing and Control*, 3(1):1 – 18, 2008.

[ZN11] Mo Zhu and Srikanth Nori. (re)enabling support for the cc2420 chip on the tinyos simulator. 2011.

[Ös06] Fredrik Österlind. A sensor network simulator for the contiki os. 05 2006.