Stony Brook University

Rbuff: Improving congestion in sensor networks under event-driven and burst data traffic

A Thesis presented
by

Abhay Ravi Chandran

to
The Graduate School
in Partial Fulfillment of the
Requirements
For the degree of

Master of Science
in
Computer Engineering

Stony Brook University
May 2011

Stony Brook University

The Graduate School

Abhay Ravi Chandran

We, the thesis commitee for the above candidate for the
Master of Science degree, hereby recommend
acceptance of this thesis

Professor Harbans Dhadwal
Associate Professor
Department of Electrical and Computer Engineering

Professor Alex Doboli
Associate Professor
Department of Electrical and Computer Engineering

Professor Jennifer Wong Ma
Assistant Professor
Department of Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of Graduate School

Abstract of the Thesis

Rbuff: Improving congestion in sensor networks under event-driven and burst data traffic

by

Abhay Ravi Chandran

Master of Science

in

Computer Engineering

Stony Brook University

2011

Advancements in sensor and MEMS technology have enabled high resolution, high data-rate, and complex sensors which enhance the application domain of sensor networks. In addition, these sensors enable sensor networks to capture high quality data with more precision. While increased storage capacities on sensor nodes have previously enabled sensor networks to store and forward data leisurely, many emerging sensor network applications, such as seismic monitoring, real-time object localization and tracking, or pervasive health monitoring, require real-time reporting of this high resolution, event-driven data. The existing communication and radio stack in sensor

network operating systems were designed for simple packet handling; however they fail under high data-rate and burst traffic. In this work, we propose a modified communication stack which includes a receive buffer (RBuff) to handle burst traffic more efficiently, reducing traffic congestion. We present a theoretical analysis on the optimal buffer size based on the properties of the expected burst traffic within the network. In addition, we address the dual scenario; we present analysis to determine the maximum burst size and wait time given a limited fixed buffer size. Experimental analysis on single-hop, multi-hop forwarding trees, and random network deployments demonstrates a 50% increased packet reception rate under burst traffic of the optimally sized RBuff over the existing single packet slot within the Contiki operating system. Additionally, we show that a fixed buffer implementation with predetermined burst sizes and wait times also provide better results than the single buffer implementation. We demonstrate how a modest buffered approach improves packet reception in event and burst traffic scenarios and aids in reducing overall network energy consumption by reducing collisions.

Contents

List of Figures

Acknowledgement

I believe that research can be viewed as a path for finding a solution to a well defined problem by the power of man, mind and machine. Through this thesis, I have been given the freedom and opportunity to do research.

I sincerely thank my advisor, Professor Jennifer Wong from the Computer Science Department and Professor Harbans Dhadwal from the Electrical and Computer Engineering Department for their willingness to support me and allow me to present my research work through this thesis. I thank them for the motivation, encouragement and guidance they have constantly provided without which, this work would not have been possible.

I am also grateful to Professor Alex Doboli for serving on my thesis committee and reviewing it.

I would like to convey my thanks to Mr. Hui Kang, who has helped me out during the course work of my thesis and assisted me in my research work. A special thanks goes to my parents for supporting me unconditionally in the past years.

I would finally like to thank everyone who have believed in my abilities and more importantly those who have challenged them.

Chapter 1

Introduction

Sensor Networks continues to be an emerging field in Computer Science and Engineering. The main driving force behind the research and development efforts is the myriad of applications for which sensor networks can be applied and the advancements in hardware technology continue to facilitate enhanced sensor and pervasive systems. MEMS and silicon advancements have enabled the creation of smaller and increasingly complex sensor devices which can be embedded into environments. Additionally, wireless technology has pervaded successfully, leading to higher bandwidth communications and lower power operation. These factors enable sensors to be deployed anywhere, including the most inhospitable environments, and retrieve information which would otherwise would not be available or observable.

Sensor networks have been applied to a large variety of application areas such as medical sensing [1], [2], environmental sensing [3],[4],[5], scientific studies [6],[7],[8], automobile sensors [9], pervasive computing and for social networking [10], [11],[12].

For example with the help of remote monitoring in the Health Sciences field, doctors are able to remotely and constantly monitor their patients in terms of

vital signs and exercise for example. In the case of automobiles, enhanced sensor technologies have helped in reducing of casualties. Sensors are used to detect tire pressure, flow speed, fuel injection etc in cars. Wireless implementation of these sensors have enabled the reduction in the use of wires to interconnect the sensors. Energy efficiency is improved in power distribution grid by studying the usage levels among customers using remotely deployed sensors [13].

All of these applications of sensor networks have been successful and resulted in working and beneficial systems, however a large number of engineering problems plagued engineers in order to complete development. In addition, there are many engineering and software related challenges which still remain to be addresses. In addition, with each application area and new application of sensor networks additional challenges arise. For example, despite being widely addressed, power efficiency and battery lifetime continues to remain a major issue in sensor networks.

Another major issue is wireless communication bandwidth in high density sensor deployments. High densities, lead to high collision rates, which result in high retransmission rates which are a power drain. Any form of packet loss whether through collision or congestion, will eventually increase energy requirements of the sensors and worsen energy efficiency.

1.1 Motivation

In this thesis we try to improve data reliability and scalability by focusing on the issue of communication congestion. We find congestion is a very prominent problem in sensor networks because of the nature of the type of observation phenomena. Since sensor networks constantly provide data feedback, events which trigger data

aggregation tend to cause multiple sensors to respond to the same event. In such cases large amounts of data are sent simultaneously through the network to fixed sink/collection nodes. In many cases, the nature of this data is bursty. We therefore experience a type of congestion termed popularly as *the funneling effect*. This leads to packets being dropped since the sink receiver is overloaded simply because it cannot process the packets quickly enough. This entire system is not isolated to this form of congestion alone. We would also experience congestion and packet loss if an application developed takes a large amount of time to process received packets and blocks reception of other packets. These packets which arrive in very short periods and large bursts are lost. As a result, the senders will have to resend their data packets when they do not get an acknowledgement for the packet they sent. The situation compounds if the depth of the communication tree is high. If this retransmission is repeated several times, we will spend a large amount of energy to successfully send only a few packets. Such high ratio of energy cost per packet is not acceptable in this energy scarce sensor system.

Most sensor operating systems (such as Tiny OS or Contiki) contain a memory element to hold a single packet only at the network layer of the communication stack. Moreover they perform direct call backs to the application without an execution break. This implies that packet reception rate is dependent on the type of application developed by the user. Packet loss is guaranteed if the user burns CPU cycles performing time consuming tasks and ignoring received packet bursts. This is acceptable design if we are receiving sporadic packets or low rate data streams where a single packet buffer would suffice to retain each packet. Since the packet has to be processed completely for the next packet to be received a direct call to the application is also justified. But if we are to handle high data burst rates which are achievable

with high resolution and complex sensors and the funneling problem described earlier we must isolate the reception process into two separate processes and incorporate a multiple buffer based concept.

We can illustrate with a real world example, which makes use of data intensive applications where burst traffic can be very high. In [14], the author proposes using sensor networks to perform visualization on urban traffic monitoring. Such monitoring will involve a large amount of processing, as the author mentions the analysis of images and video data for visualization of the data in a way that can be made easily intelligible. Further more transport of such data will involve transmission of large amount of data, which is predominantly image and video based.

Example of one simple system that makes use of this kind of monitoring is a traffic violation monitoring system. Such system might use cameras to detect cars passing through a signal intersection. The data from these systems are collected and processed by wireless sensor nodes. We can assume that for busy intersections the frequency of cars can average at around 20 to 30 seconds per car. All cars observed have the image processed for better visualization of the information. A high amount of processing time is involved for each image captured before it is forwarded to the traffic observation station. Also since images are being sent, there is a large amount of packet traffic. These occur at instances when images are captured. This kind of a scenario provides an ideal situation where a buffer based implementation could provide a better performance over a non-buffered approach.

In our work we try to reduce and handle the congestion and pack drops in high burst data scenarios by introducing the Rbuff packet buffer. We propose two approaches: (i) a theoretical optimal Rbuff size based on anticipated data burst sizes and network configuration and (ii) the dual problem: maximal data burst size and

application blocking time given a set Rbuff size based on hardware memory limits.

We implemented the proposed Rbuff on the Contiki operating system. We implement the separation of the Rime Radio Stack (of Contiki) from the networking channel and application. This ensures that when packet reception occurs, multiple packets can be stored before the application begins processing the information. Through experimentation, we find that the addition of the Rbuff reduces congestion and improves performance by up to 50 percent.

1.2 Thesis Overview

In Chapter 2 we provide an overview of sensor operating systems. We highlight the working of Tiny OS and Contiki Operating systems. We discuss the characteristics of the operating systems and how they differ from each other. We also discuss the radio stack design and the packet handling procedures.

Chapter 3 provides a detailed overview of the problem with the current packet reception implementation in sensor operating systems. We discuss how bursty traffic will result in packet loss when addressed with application intensive processing. We then introduce the concepts of Rbuff. We provide a detailed explanation of the improvements it provides and how it has been implemented.

Chapter 4 deals with the theoretical analysis of our problem. It presents analysis for determining the optimal buffer size based on characteristics of the burst data traffic. We then tackle the dual of the problem and derive the burst size and wait times from a fixed buffer size.

All the results obtained through testing are discussed in Chapter 5. We use our theoretical model derived from Chapter 4 to model our experiments and obtain the

results from the model.

In Chapter 6 we mention prior related work. We discuss the contemporary work in this field and mention how our result is different from their work before we conclude our work in Chapter 7.

Chapter 2

Background of OS

In this chapter we first discuss the operating system basics and then we discuss the TinyOS and Contiki operating systems. We end our discussion with the radio stack of Contiki.

2.1 Operating System Basics

There has been increasing complexity of application requirement and need for timely response based on input from sensing systems. The timeliness can ensure life or result in death and mean the difference between millions of dollars in expenditure. Consider the various attributes, sensors in a typical health care unit might measure and how they must intertwine the information into wireless signals to inform the doctor in a timely manner to respond to a critical patient. Similarly consider the criticality of response of a tsunami warning system trying to warn us of an impending tsunami. We must realize that most of these sensor systems operate independently, secured by a limited set of resources (memory, battery, inaccuracy of sensors) and

face rough conditions. If the system has to manage these resources carefully it must be able to control how they are utilized. This exemplifies the need for a sensible light operating system which can provide the resource management. Operating systems in these device do more than just manage the resources, they also help make programming the devices much simpler, abstracted and enable module based development which can help push early market release for appliances that are time critical in the market.

In simple terms, an operating system is a system software that manages various resources of the system and ensures programs are executed correctly and in a timely fashion while abstracting lower level details of the hardware. The program can be any code which twiddles with the sensing systems to obtain meaningful information in the case of a sensor network. Operating systems are very common in large personal computer, servers and mainframes. They are not considered important in smaller embedded devices. With the rise of the mobile phone market, the operating system based design has gained more precedence. Since mobile phones have become extremely complicated, there is a need to manage multiple applications being run on the phone and orchestrating that with the basic functionality of the phone to receive and initiate phone calls. We find that some applications in a mobile phone might be more important than others and receive precedence when executing. Such systems are said to be real-time in nature and the operating systems are generally flexible and capable of scheduling application tasks on the fly. Such systems consume a large amount of memory incorporating a scheduler program and kernel space.

Conditions such as the wide range of sensor network applications, the variety of hardware configurations and constraints, and data collection latency constraints, to name a few, require that sensor networks and sensor systems be structured

and designed differently than traditional operating systems. Most of these sensor systems operate independently, secured by a limited set of resources (memory, battery, inaccuracy of sensors), and face rough environmental conditions. If the system has to manage these resources carefully it must be able to control how these resources are utilized. This exemplifies the need for a sensible light operating system which can provide the resource management. Operating systems in these device do more than just manage the resources, they also help make programming the devices much simpler, abstracted, and enable module-based development which can help push early market release for devices that are time critical in the market.

The most important feature of the operating systems (OS) is that it must be light weight. Memory itself is a constrained resource. But nonetheless the OS should be able to perform resource allocation and abstraction for the user programs. The radio stack for example must not have heavy cluttering of headers and error prediction and should implement basic error recovery for least amount of processing time. Sensor OSs will not have heavy real-time schedulers and do not generally allow preemption. They do however allow thread based execution and simple scheduling on a first-come first-serve basis. Unlike real-time systems which generally give each user a process space and unique memory for each application, sensor Oss make different threads share common process space. For example, real-time OSs generally range from 100 KB to 10-12 MB in space, while sensor OSs roughly occupy around 10-30 KB at most. Figure 2.1 provides a list of main differences between sensor and real-time OSs.

Real Time OS	*Sensor OS*
100 KB to 10-12 MB in space	around 10-30 KB at most
Radio stack contains error correction, encryption and buffered packet reception at the cost of high over head of header length and memory.	Radio stack contains basic error correction single packet reception and implemented as a simple callback stack.
Real time schedulers which schedule tasks the moment they are called. The schedulers allow basic forms of preemption.	Basic Schedulers which just queue tasks one after another. Preemption is too costly and hence generally non-preemptive.
Each application is given a separate process space to execute in. Context switching occurs between processes.	The programs are divided as threads but each thread is given only limited process space and the space is shared by all threads.

Figure 2.1: The list of differences between Sensor and Real-time Operating Systems

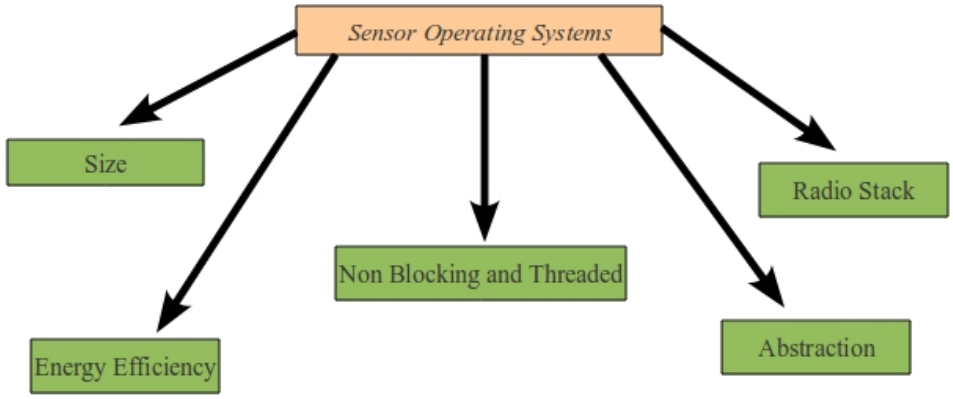

Figure 2.2: Characteristics of a sensor based operating system

2.2 Sensor Operating System Characteristics

A number of sensor network operating systems have been proposed by the research community. The most popular OSs are TinyOS from UC Berkeley, Contiki from the Swedish Institute of Computer Science, Mantis from the University of

Colorado Boulder [15], and Nano rk from Carnegie Melon University [16]. Most of the operating systems are free open source software, are built on the same principles, and are implemented on similar low power hardware platforms.

Four of the main system components are:

Size: As sensor nodes are made of simple hardware, the amount of memory (ROM and RAM) available for them is limited. Most sensor nodes use have flash program memory and are limited to around 32-64 KB and around 4 KB of RAM. Operating systems developed are also limited in size to with-in 30 KB at most. These operating systems are hence termed "lightweight". This means that in the attempt to reduce operating system sizes, the functionality of the sensor operating systems generally are stripped down versions of their larger counterparts.

Energy Efficiency: Energy efficiency is a very important aspect of sensor design due to limited battery life. Most sensor operating systems allow energy efficiency by putting the micro-controller to sleep when the operating system is not executing any operation. Similarly the sensor is awoken from sleep when a task gets scheduled or an interrupt due to a message arrives. The most energy intensive component of a sensor node is the radio which is switch off until a send is required or a periodic receive is to be performed. TinyOS and Contiki make use of special MAC protocols which use a duty cycled radio scheme which helps reduce energy efficiency.

Radio Stack: What sets apart real-time operating systems from sensor operating systems is the radio stack. All sensor operating systems must support a wireless radio stack. Most modern sensor OSs support IPv6 and IPv4 at the network layer. The radio stack has to be simple and lightweight and capable of powerful abstraction of the radio functionality. For example the TinyOS radio stack is segmented into multiple layers. It contains the driver layer which handles

radio hardware, a MAC layer which handles the collision and energy efficient radio operations, a message reception layer which ensures that the message is received with acknowledgement, a powerful network layer which can use IPv6, etc. Errors in the packet are detected by simple CRC codes. Since the radio stack is simplistic complex error detection or encryption techniques are not utilized. Most sensors use the Zigbee physical radio protocol.

Driver Abstraction: What makes application development in sensor OS easy is the abstraction of the lower level functions. Most sensor operating systems enable sending and receiving without the need for users to know the underlying concept of the OS or the radio stack. In Contiki for example we can achieve reliable communication using a specific channel called "rudolph". The beauty of this channel is that by using the constructs of the channel, the user can send packets between two sensor nodes and not worry about whether the packet was received or not. A system of acknowledgement based communication is setup between the nodes. A failed send is returned if the receiver does not acknowledge after a timeout period. This kind of abstraction helps make the software more reliable and efficient. Probably the most important form of abstraction is the abstraction of the driver level software. Most sensor operating systems abstract away the hardware level detail of the radio, sensors, serial communication (USB, SPI) and LEDs. These features of the system are made accessible through the use of wrapper functions available at higher application layers.

Event based and Threaded: Most sensor operating systems are required to process non-blocking applications and are event driven. They do not allow preemption among different threads. Although a very basic form of preemption using interrupts is still enabled. It is very important that the application is non-blocking. Blocking programs might lead to the overuse of CPU resources and also result in packet loss.

Tiny OS supports non-blocking programs only and will lose packets if an application blocks the CPU even though an event will be flagged for execution of the packet read. Generally complicated processing must be broken into blocks of code which can be executed separately. Most sensor operating systems support threading. Although multi-threading is very hard to achieve, a single threaded environment is supported which supports a loose kind of multi-threading using interrupts.

2.3 Tiny OS

TinyOS is a very popular open source Linux-based sensor OS. It is one of the first sensor OS developed. It was proposed in 1999 in the University of California at Berkeley in the sensor research group with collaboration from Intel research and Crossbow technologies. TinyOS is written in nesC which is a dialect of C. It is an event-driven operating system. It is non pre-emptive and does allow pre-emption. Applications developed must be non-blocking in nature. This means they cannot execute for long periods of time. TinyOS is extremely lightweight and can fit into most small micro-controller memory. TinyOS has inbuilt support for Texas Instruments MSP430 family, Atmel's Atmega128, Atmega128L, and Atmega1281, and the Intel px27ax processors. TinyOS also supports Zigbee standard radio and has support for the Texas Instruments/ChipCon CC1000 and CC2420, the Infineon TDA5250, the Atmel RF212 and RF230, and the Semtech XE1205 radios. TinyOS supports threading using TOSThreads library.

Among all the components on a sensor mote, the wireless radio accounts for a large portion of the energy cost, because the mote needs to exchange data with other motes during most of its lifetime. The radio driver layer of TinyOS is shown

in Figure 2.3. On receiving a packet on the physical radio, the radio driver sends an interrupt signal to MAC layer through a SPI bus. The MAC layer is responsible for checking FCF information and the destination address of the received packet. Other functionalities can be added after this step such as auto retransmission and acknowledgement. Next, if the packet is a routing packet, it is passed to the network layer which defines the routing behavior of the network. Otherwise, the data contained in the packet will finally be passed to the application layer for some specific purpose.

Unlike most network interface cards (NIC) for a desktop or server, the radio device (e.g., CC2420) on the mote does not have a cache to buffer multiple packets under high network traffic. In addition, each component passes the pointer to the other one in a split-phase call mechanism, which means that the pointer can not be modified until the associated data are processed by the upper layer. Combining these two characteristics, the single pointer of TinyOS's radio driver can not support high network traffic, resulting in performance degradation in terms of packet loss. Furthermore the application cannot be blocking; if the application is blocking the blocking time will result in packet loss.

2.4 Contiki OS

Contiki is an open source event based operating systems meant for memory constrained systems such as sensor networks and embedded systems. The operating system has been developed by Adam Dunkel from the Swedish Institute of Computer Science. The OS has normal memory foot print of about 2KB of RAM and 40KB of ROM. The low memory foot print makes it very efficient in terms of memory. It is an event-based operating system which means that the operating system runs processes

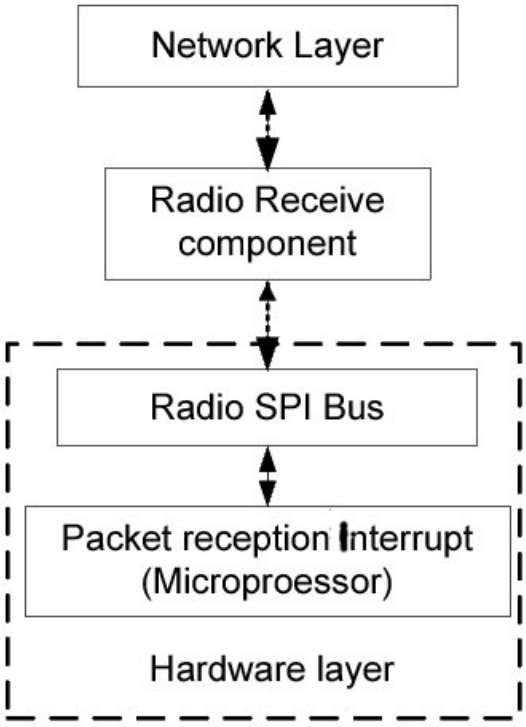

Figure 2.3: TinyOS radio stack for receiving packets

based on events which occur in the system (such as timers, interrupts, button presses, etc).

The basic unit of the Contiki OS is its proto-thread based architecture. The proto-thread provides both the flexibility and multitasking nature of normal thread based applications and also encompasses the low memory consuming properties of event-based systems. The OS supports power management and sleep modes and also allows power measurement. It also provides a number of tools to simulate the

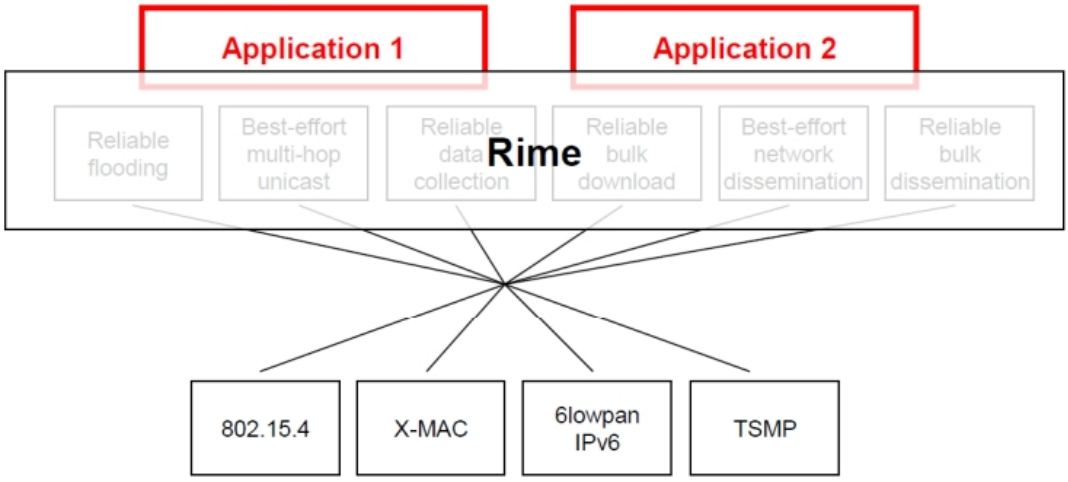

Figure 2.4: Rime in Contiki

operation of the application before being ported onto other platforms. Currently the OS supports the T-mote sky, Mica Z platforms and the native Linux platform. It has a radio protocol stack called "Rime" which provides short range radio communication in Zigbee. The operating system is one of the very few which provides a complete robust IPv6 and IPv4 stack. It features a running browser capable of sending TCP/IP and UDP packets.

In our work we have used the Contiki operating system for experimental analysis. In the following subsections, we provide a detailed explanation of the Contiki radio stack.

2.4.1 Rime

The main feature of the Contiki OS which is most relevant to this work is the radio communication mechanism. The radio operates on Zigbee or 802.15.4 standard.

The radio is named as the “Rime low-power radio networking stack” and provides number of features for Zigbee based communication. The Rime model is shown below in Figure 2.4.

The Rime features a working network, MAC and physical driver software for supported platforms. The OS completely abstracts the sending and receiving of the packets to a set of high level API calls. The type of transmission can also be selected from the upper layer by declaring only the required type of connection. The various types of connections supported are,

1. Single-hop broadcast (broadcast)
2. Single-hop unicast (unicast)
3. Reliable single-hop unicast (runicast)
4. Best-effort multi-hop unicast (multihop)
5. Hop-by-hop reliable multi-hop unicast (rmh)
6. Best-effort multi-hop flooding (netflood)
7. Reliable multi-hop flooding (trickle)
8. Hop-by-hop reliable data collection tree routing (collect)
9. Hop-by-hop reliable mesh routing (mesh)
10. Best-effort route discovery (route-disovery)
11. Single-hop reliable bulk trans fer (rudolph0)
12. Multi-hop reliable bulk transfer (rudolph1)

All the 12 different types of network connections are provided by a basic kind of connection called ”Anonymous Broadcast based Connection” (or abc). The anonymous broadcast connection is a simple broadcast protocol where the identity

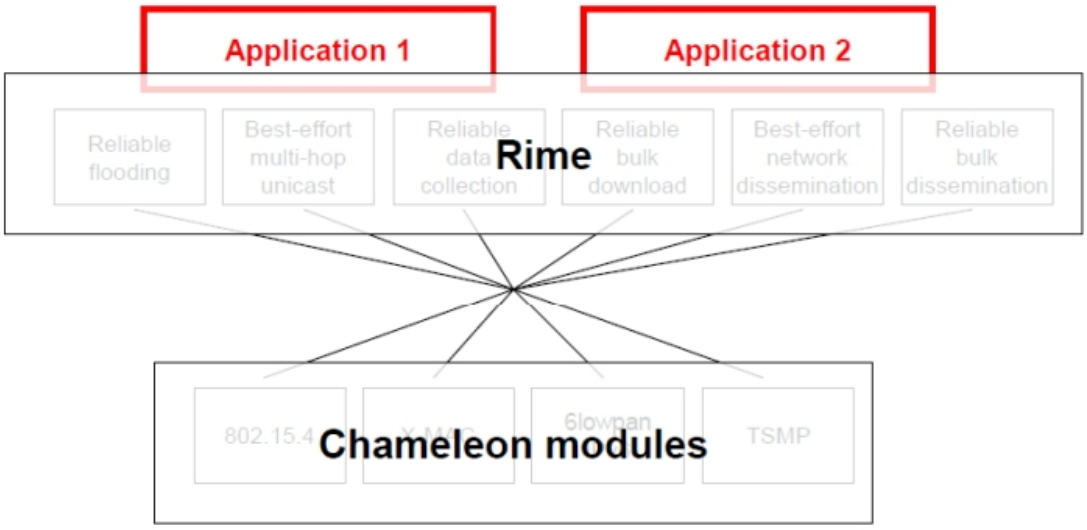

Figure 2.5: Chameleon in Contiki

of the sender is not sent and information is broadcast to all nearby nodes. All other connections are built upon the basic framework of the abc broadcast and hence use the same set of call back functions from the lower layers. The Rime only abstracts the transmission of data but does not handle the header and CRC addition to the packet. In Contiki the header addition and CRC calculations are separated from the actual data transmission by splitting the bit level operations into another module. This is called as the Chameleon module.

2.4.2 Chameleon

The "Chameleon" module handles the bit level header additions. It adds the relevant header information and calculates the CRC to the data and attaches this information to the data being sent. On reception the data is unpacked from the

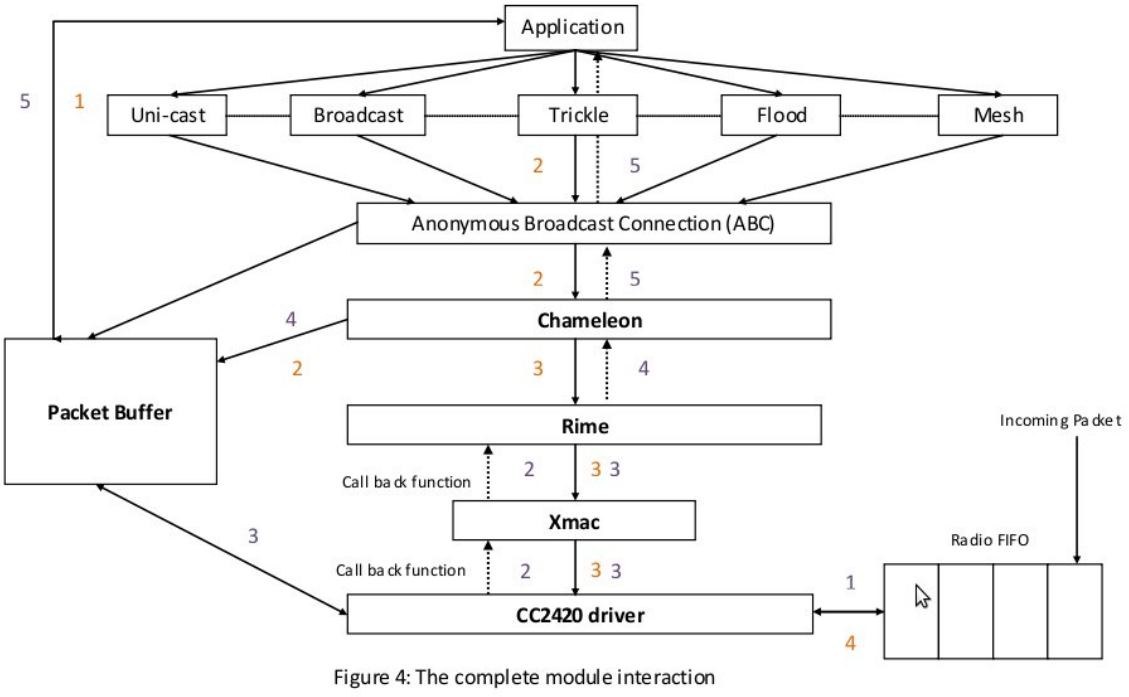

Figure 2.6: Contiki radio stack for receiving packets

header and the data is forwarded to the upper application layers. The chameleon model is as show in Figure 2.5

Confining the lower layer operations into distinct modules serves a very useful purpose in making the development of the application only focus on the protocol and data dissemination technique and not on how the packet is sent and the headers are built. These also help in confining the changes to these specific modules thus making Contiki development much easier to understand and implement.

2.4.3 Contiki Radio stack

The Figure 2.6 shows how the reading and sending of packets are performed. The figure represents the entire flow of packet data up and down the Rime stack. Incoming packets are stored into Radio FIFO to be processed by the radio cc2420 driver. The following occur during an event of an incoming packet (numbered appropriately in the figure in purple)

1. Radio driver is interrupted by the radio. The OS is woken up from sleep mode and the receive packet process is put into the queue to be executed in order.
2. The process is executed and the process calls a call back function to the Xmac module which again calls back the Rime module to initiate a packet read. The rest of the process is executed sequentially without any break in execution.
3. Rime calls Xmac to read the incoming packet. Xmac in-turn calls the radio driver to read the data from the FIFO. The data is read and the data read is put into the packet buffer. The packet buffer is a buffer 160 bytes in size and packet size is 128 bytes with a 32 byte header. It can store only one packet.
4. The information is checked for CRC in chameleon modules and the headers are removed to only expose the data part of the packet.
5. It passes to the application through the abc module and other connection modules and calls a call back function in the application. The call back function can be used to read the data from the packet buffer. As mentioned earlier all steps from 2 to 5 are executed consecutively. Step 5 produces the greatest amount of latency in the packet read as it is completely dependent on application developed.

The following occur during an event of an outgoing packet (numbered

appropriately in orange).

1. Application copies data into the packet buffer. It then calls APIs to send the data in the packet buffer.
2. The connection modules and abc module call the chameleon module which attaches relevant CRC and header to the data in the packet buffer.
3. The data is then sent to Xmac through the rime. The Xmac checks if previous data is being sent. If it is being sent it stops the current packet and puts it into a queue to be sent later by Xmac. When the packet is ready to be sent it sends the packet to cc2420 driver. The queuing prevents loss of outgoing packet when the radio is busy sending another packet.
4. The driver sends the data to the radio and it is sent by the radio to the destination.

The radio stacks of Contiki and Tiny OS are quite similar. They operate on a similar architecture. Most of the Contiki radio model has been derived from TinyOS but the radio stack has improved enhanced. Once again the problem in Contiki is the non-blocking requirement of the application program. We see the packet buffer can hold only one packet at a time. This does not cater to bursty data traffic and packet loss due to application blocking.

Chapter 3

Rbuff Approach

In the previous chapter we discussed how the radio stack in Contiki and TinyOS have been developed. We will now highlight the issues with this implementation under event-driven burst traffic resulting in packet loss. Then we discuss the Rbuff design and implementation in Contiki and TinyOS.

3.1 Issues With Existing Radio Stacks

Most sensor network applications leverage on bursty traffic to enable low duty cycle operation. However, during a single cycle sensor traffic can be extremely bursty in nature. Most commonly applications are event-driven and as a result large amounts of data are generated at multiple nodes in a short period of time which must be transferred to a collection center, generally a large single fixed nodes called a sink. As mentioned earlier this leads to the funneling effect. As data gets funneled, a large number of packets will be sent together in quick succession. This can cause congestion within the network and lead to packet drops. What is also noticed in such networks

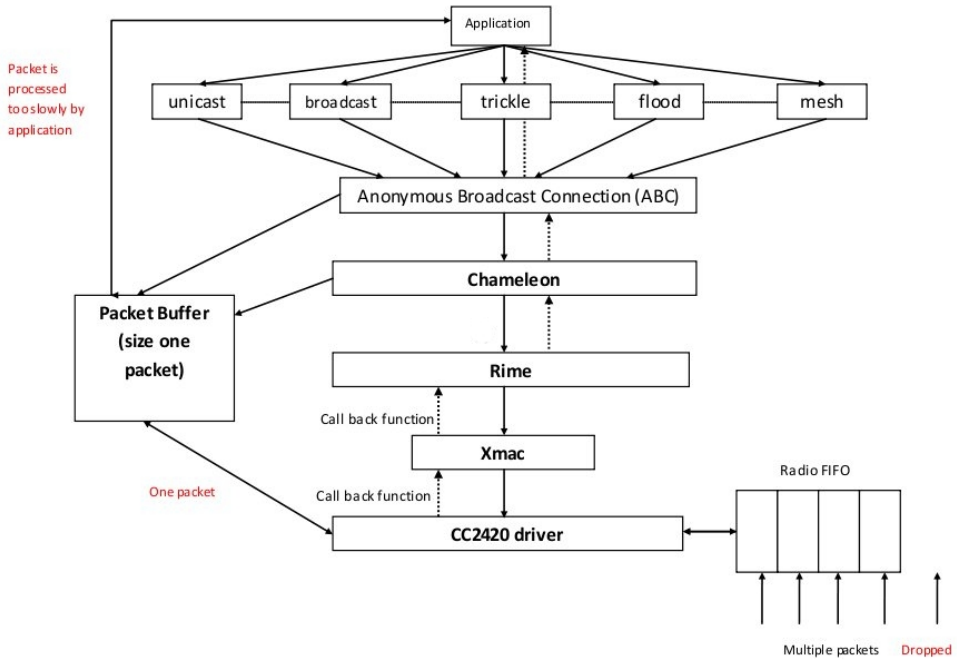

Figure 3.1: Drawback of current Contiki radio stack implementation

is that the sending of data is very random, due to random event occupancies, and are separated by long time intervals of minimal or no network activity. The main issue in the current sensor operating systems implementation is that there has been no mechanism developed which can actually hold these packets which are produced in such short periods or bursts. For example we find that in the case of of the current implementation in Contiki, a burst rate of 64 pkts/sec leads to a packet drop of up to 50%. The problem substantiates the need for buffering the network stack used in sensor networks.

Additionally, both Contiki and TinyOS are structured around a non-blocking principle. Since these operating systems don't allow pure pre-emption, the system as a whole is vulnerable to highly bursty traffic. The blocking of an application

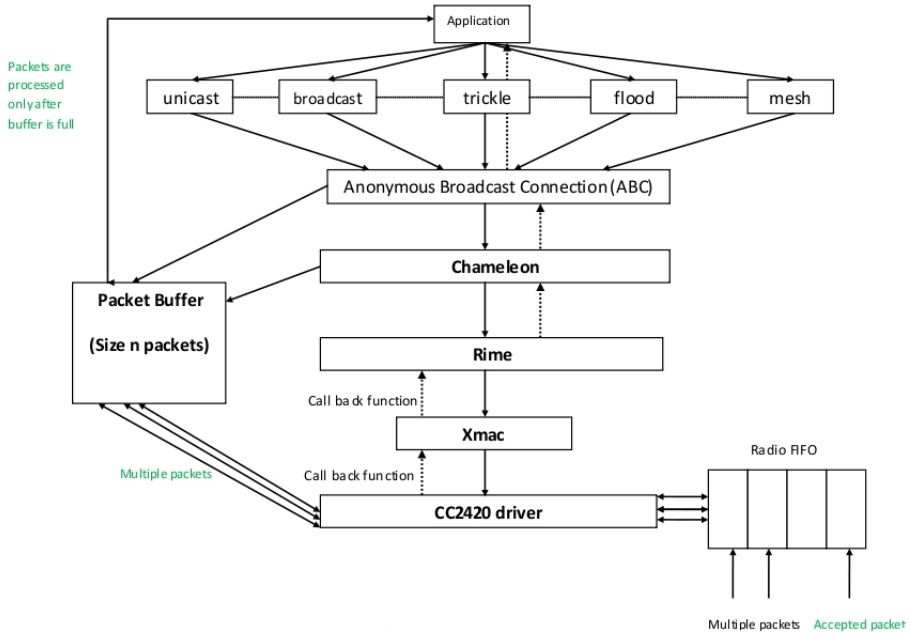

Figure 3.2: Contiki radio stack Rbuff implmentation

can lead to clogging of the sensor node's resources. When incoming packets are not processed quickly and stored they will be lost when the application layer blocks. This is a major reason why the current design fails. With only space to hold one packet, we are forced to process the packet before we allow reception of other packets at the higher communication layers. Handing the control to an application can have disastrous consequences if, the application blocks for too long.

Clustered packet traffic can also lead to large amounts of collision. Since we have many nodes trying to send together and nodes higher in the forwarding tree forwarding on the packets they have received, we experience a large amount of collisions in the network. Higher collisions can further accentuate the packet loss and lead to greater energy consumption.

Figure 3.1 shows the typical Contiki radio stack and how the current design will

affect the reception of packets in bursty conditions. If multiple packets arrive then handling one packet at a time can prove costly. If the packet buffer size were increased to accommodate more than one packet then this problem can be reduced and packet reception can be improved. If multiple packets can be stored into the packet buffer and if the packets are processed in groups during low or no traffic times then packet loss can be reduced considerably.

Consider the figure 3.2 with the changed implementation. Although it looks straight forward, the packet read is not done simultaneously from the radio driver. The packet is read one at a time and reaches the application layer. But the application layer does not process the packet until the buffer is full. Hence with no application overhead involved the buffer can be filled in quick succession making the FIFO empty for the next set of packets to arrive. Though some time overhead is incurred during packet processing, we assume that this processing happens during the radio silence period. Once the buffer is full or when no more packets are received within a stipulated time the content of the packet buffer is processed and the process repeats.

3.2 Rbuff

We have deigned a new communication stack by modifying the existing principles and adding features to address the event-driven burst traffic environment. The primary insight behind our design is that we store each incoming packet in a buffer before it is processed by a lengthy application. We explain how we estimate a suitable buffer size based on the model derived in the next section. We call the new communication stack Rbuff.

Different data structures for holding packets can be used by Rbuff. We currently

use a simple variant of FIFO queue that we found to work well in practice: we implement the FIFO queue as a circular array, like a ring. The ring buffer is a fairly standard lock-less data structure for producer-consumer communications. Figure 3.3 illustrates the structure of the Rbuff, which has a buffer size of B. It has three main components: head and tail pointers as the consumer and producer, and the buffer itself.

The buffer is initialized by setting both the head and tail pointer to the Bth slot. When an incoming packet generates an interrupt, the tail of Rbuff is checked by the radio driver to see if there is enough space ahead of the packet being processed. If there is, the slot is allocated to hold this packet and the low level radio driver is freed to receive the next incoming packet immediately. The tail pointer is then decremented by one space. The task of processing the incoming packets in Rbuff is usually deferred to facilitate concurrency of the whole system. The other end reads the packet from the head pointer. After processing the packet, the caller then removes the packet from the buffer and decrements the head pointer. The buffer slots in between the head and tail are occupied by packets already received, but not processed.

The operations of allocating and releasing a slot are represented by the following equations.

$$tail = (tail - 1) \ mod \ B \tag{3.1}$$

$$head = (head - 1) \ mod \ B, \tag{3.2}$$

where B is the capacity of the Rbuff. Thus, testing of buffer overflow and availability relies on the relative position of the head and tail pointer. If the head and tail point

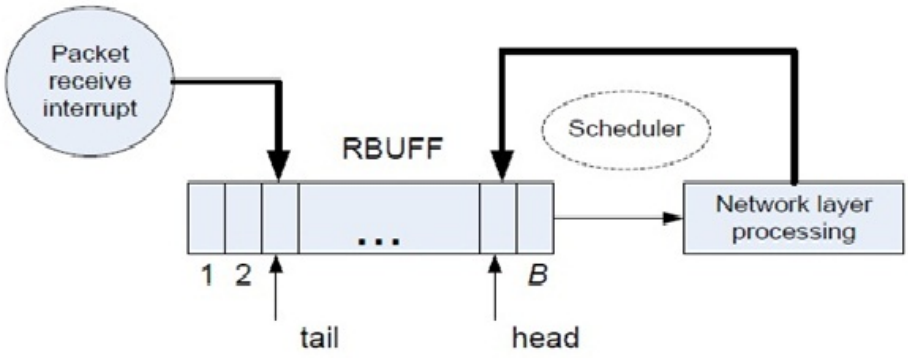

Figure 3.3: Packet receiving using a Rbuff with size B. Rbuff allocates a empty slot for the incoming packet at the tail; processed slot is released at the head.

to the same index, the buffer is empty. If the tail is one slot before the head pointer, the buffer is full.

3.3 Implementation

The proposed receive buffer is implemented as a separate component in both TinyOS and Contiki operating systems, which use different programming strategies. TinyOS is purely event-driven and non-blocking, while Contiki uses a sequential code structure that allows for blocking functions. In each case, Rbuff is added in between the MAC layer and the network layer. Thus, the message pointer to hold the next incoming packet is returned by the Rbuff component, instead of the upper layers. The network layer is able to process several or all the incoming packets in Rbuff.

The packet buffer is currently designed to hold a single packet of information which is extracted from the radio buffer. During the transmission of a data packet the sent packets are queued if the lower layer (i.e. FIFO) is sending the previous information. During reception only one packet can be received and be processed at

a time. The design has a drawback that if the application spends too much time handling the received data in the packet buffer, later packets that arrive can be dropped if the FIFO in the radio driver becomes full. This can occur when there is a very high amount of congestion in the network or if the data is very long and contains multiple packets. One way to alleviate the problem is to fetch the next packet immediately after the application gets the first packet and process the data buffer at a later stage for the application purpose. We do this using the Rbuff implementation.

To enable this process, the data buffer Rbuff must be made to handle multiple packets at the same time. Making the packet buffer size greater than one packet should help reduce the congestion problem. The application should also know or must be capable of obtaining which buffer its data is located in. It can then process the information at a later stage avoiding the congestion created from dropping packets at that given instant.

In Contiki the buffer is implemented in the interface between the network and MAC layer in our experimental setup. We modify the current "packetbuffer" module, which is currently built to handle a single buffer, into a multi-buffer based setup. We introduced additional APIs to handle multi-buffer scenario which includes the "top pointer" operation mentioned above. Any packet reception generates a callback to a receive function in the Contiki's "rime" layer. In our experiments we made the receive function (in the rime layer) copy the packet from the received FIFO buffer and copy it into the packetbuff. The operation was followed by a subsequent increment of the "top pointer" to the next position. Processing of the packets (for forwarding etc) was done in a separate process in the application layer called cyclically in set time. This way we avoided overhead of the application processing.

Chapter 4

Buffer and Time Interval Analysis

In this chapter we present the mathematical analysis to derive the buffer size which is optimal for loss free packet reception. We also further derive a suitable burst size and wait time for each node given a fixed buffer size for each node in the entire network.

4.1 Buffer Analysis

In this section, we present theoretical analysis for the selection of the buffer size in order to achieve a targeted reception ratio under bursty traffic. We present theoretical analysis for modeling packet forwarding at each node under bursty traffic, and determine the minimal buffer size at a node, B, required to achieve lossless forwarding within the forwarding network under worst case.

First, consider a single node L, who has n children as illustrated in Figure 4.1. When an event occurs, these n children nodes each generate a burst of traffic which is forwarded to node L. The traffic generated might be due to two reasons: i) the

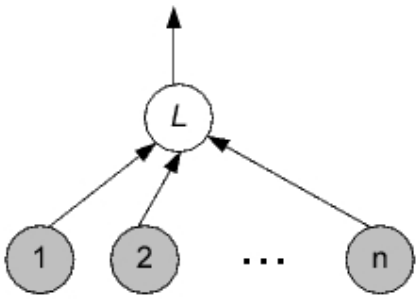

Figure 4.1: Node L forwards the bursty traffic from n children nodes.

child nodes generates the packet due to the detection of an event, or ii) the child node forwards packets it received from its own children due to events detected locally.

Because the information observed by each node is different due to the node's position to the event, the sensor used, or even the storage capacity at the node, the number of forwarded packets during the burst period will vary. Therefore, let $b_i, i = 1, \cdots, n$ denote the number of packets to be transmitted from node i. We can arrange these children in increasing order based on the number of packets they will generate. Without loss of generality, we assume their ordering is $b_1 \leq b_2 \leq \cdots \leq b_n$. In order to determine the maximal buffer size which is required to minimize the packet loss, we must consider the worst case scenario. Under worst case, all children nodes begin sending their burst of traffic simultaneously. We must have the buffer handle this situation by holding the excess packets till the packet can be forwarded or consumed.

To simplify the analysis, we assume that all nodes i have the same sending rate, b_r and let $S_j = \sum_{i=j}^{n} b_r$ to denote the total incoming packet rate (sending rate of the children) of the $(n - j + 1)$ nodes. Because the nodes are sorted by increasing

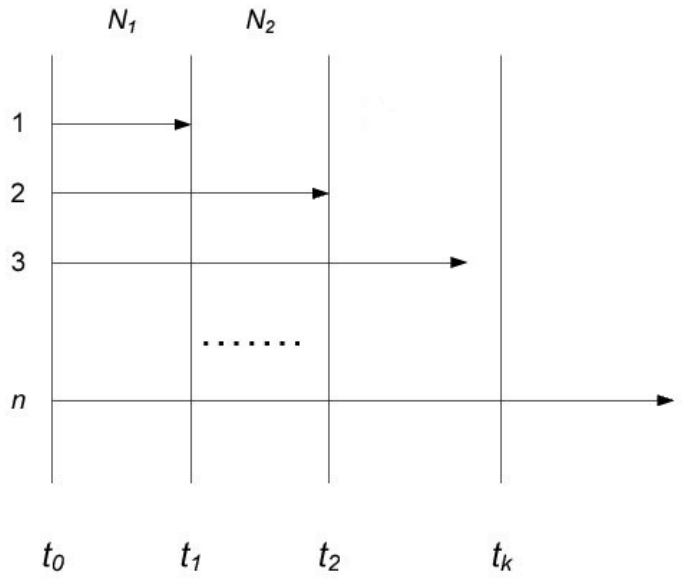

Figure 4.2: Worst case time series of burst traffic to Node L from n children nodes.

burst size and all burst begin simultaneously, S_j represents the packet arriving rate at L for all children $j, \cdots, n$. For example, node 1 has the smallest burst of traffic to send, therefore it will finish sending packets first. After node 1 completes, the packet arriving rate at node L is $S_2 = \sum_{i=2}^{n} b_r$.

On receiving each packet, node L must process the packet and then forward it to the next node; we use S_L to denote the rate of processing and forwarding packets at node L. Ideally, if S_L is greater or equal to S_j, the single buffer slot size is able to successfully handle the burst traffic from all children. Otherwise, an appropriate size of Rbuff is required to store the excessive burst packets. Figure 4.2 illustrates a bursty traffic scenario for n nodes in time series. The nodes are ordered by the packet burst size (i.e., b_i). We denote the end of a burst, as the time t_i a node completes

sending its packets as shown in Figure 4.2. Consider the period $[t_0, t_1]$, all children nodes are sending packets and thus the total number of packets arriving at node L is

$$N_1 = (t_1 - t_0) \sum_{i=1}^{n} b_r = (t_1 - t_0) S_1. \tag{4.1}$$

Since node L can process the packet at the rate of S_L, the number of excessive packets which must be buffered at t_1 is

$$E_1 = N_1 - S_L(t_1 - t_0). \tag{4.2}$$

The second term in the above equation is the number of packets processed and forwarded by node L in period $[t_0, t_1]$. Since node 1 finished its burst of packets at t_1, the number of packets arriving between t_1 and the end of the next shortest burst t_2 is:

$$N_2 = (t_2 - t_1) \sum_{i=2}^{n} b_r = (t_2 - t_1) S_2. \tag{4.3}$$

Similarly, the excessive packets at t_2 is

$$E_2 = N_2 - S_L(t_2 - t_1). \tag{4.4}$$

In general, the excessive packets is

$$E_i = N_i - S_L(t_i - t_{i-1}). \tag{4.5}$$

To guarantee that the buffer can store all the excessive packets, the buffer size B needs to be large enough to hold all the excess packets accumulated in each t period

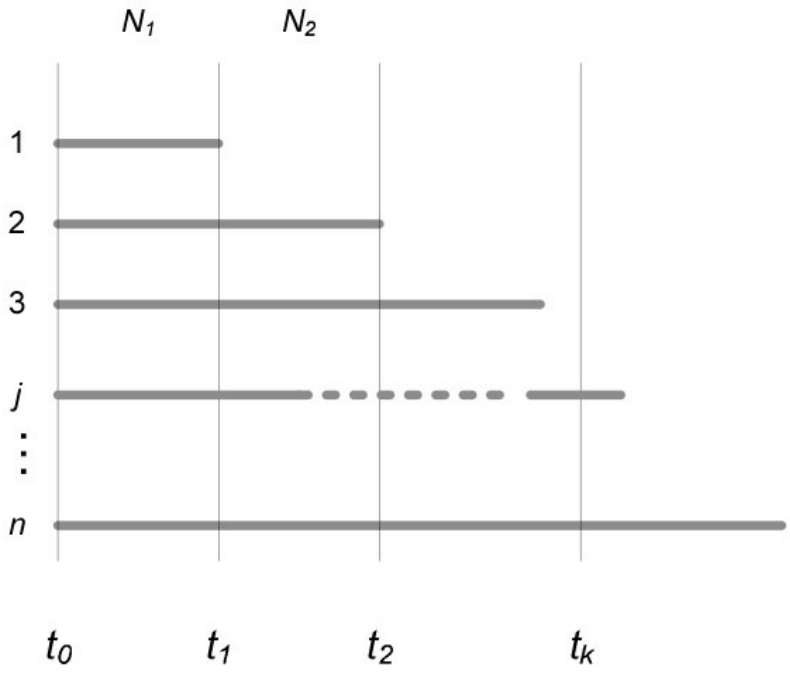

Figure 4.3: Sporadic burst and how they are assumed.

until $S_L \geq S_j$. Without loss of generality, we assume that at time t_k, $S_L \geq S_k$. So the total number of excessive packets at t_k, which is also the optimal buffer size, can be obtained as

$$\begin{aligned} B_{opt} &= E_1 + E_2 + \cdots + E_k \\ &= N_1 - S_L(t_1 - t_0) + N_2 - S_L(t_2 - t_1) + \\ &\cdots + N_k - S_L(t_k - t_{k-1}) \\ &= \sum_{i=1}^{k} N_i - S_L(t_k - t_0). \end{aligned} \tag{4.6}$$

Hence we get the expression to find the optimal buffer size above. The only issue

this does not consider is the fact that multiple bursts can occur between a single burst period of the node taking the longest amount of time to send. We then have to simplify the case by assuming intermittent small bursts can be assumed similar to a long burst of average burst rate. If we do so we can simplify the mathematics and hence use the above equation to find the optimal buffer size based on this.

For example consider Figure 4.3; we see that the burst from node j is sporadic. We find that it ends a little after time t_1 and begins a little before t_k again. But for our simplicity sake we will assume that the burst is of size a little greater than t_k so we can assume it to be one long burst. We thus have found that we can successfully find the buffer size based on these equations. The buffer sizes obtained are all the worst case values.

Using this model we can find the ideal buffer size for perfect reception. However, since memory in sensor devices is a scarce resource, practical implementation of Rbuffs of these sizes may not be feasible.. This brings us to a very compounding problem which requires us to engineer a solution which can use fixed buffer sizes but will still not result in packet drop or will limit the packet drop.

4.2 Burst Size determination

Let us assume we are now only given a fixed buffer size of B. We want to achieve the minimal packet drop given the constraint on buffer size which can be achieved. Our goal is to find out what is the maximum number of packets we can send per burst (burst size per child node) in order to attain optimum packet reception. Firstly, when we consider a burst we assume that rate of the burst is fixed for all nodes. Secondly, we consider that the forwarding within the network is sufficiently balanced. This makes

sure that the number of levels in the network forwarding tree is as minimal as possible. This is an important assumption for two reasons. It reduces the number of multi-hop packets being forwarded and secondly, it reduces the overall energy consumption due to communication. Thirdly, we assume that each child node has knowledge of the number of siblings it has.

4.2.1 Bursts due to events

Consider a single node consisting of n children as shown in figure 4.4. Each child has a burst size of $b_1, b_2, b_3 \cdots b_n$. When an event occurs (as shown by the grey area), all the children nodes start to sense the event and start forwarding bursts. We are interested in only a single event. Let the parent node receiving the burst have a buffer of size B. We may find various burst lengths. In addition, we can have multiple children sending same burst while other children may send longer or shorter bursts. Devices may require large bandwidth to send data or may need to only piggy-back the data, within a single packet based on their bandwidth requirements. Determining the burst size can vary based on classification of the nodes on these factors. Consider a camera, as an example, sending an image over the network. It will require 2 or 3 packets to send one complete image when compared to a temperature sensor, which can fill data from multiple events into a single packet. We therefore should distribute the packet bandwidth among the nodes based on the function they perform.

Each child sends bursts based on the bandwidth it requires. This bandwidth is dependent on the function the sensor node performs and the amount of information the node wants to send through the packets. In general we can assume that one node sends the smallest burst size and represent other bursts as a multiple of the smallest burst. This way we can solve for the smallest burst and obtain the other set of burst

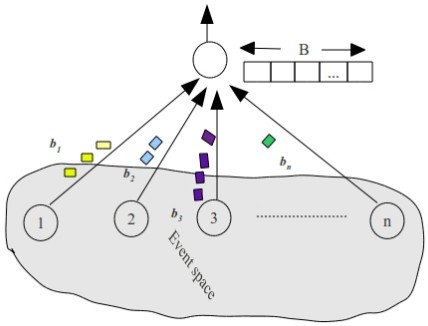

Figure 4.4: Bursty traffic.

sizes.

We can generalize the above condition and represent the total burst size b_t which the buffer must be capable of storing. This can be given by the expression.

$$b_t = \sum_{i=1}^{n} b_i \tag{4.7}$$

We represent the burst size of each child in terms of the smallest burst size b_l in the whole network by a ratio r_i such that $b_i = r_i b_l$ then we can rewrite the above equation as:

$$b_t = b_l \sum_{i=1}^{n} r_i \tag{4.8}$$

We note that the above expression is a function of the smallest burst size the parent receives. Since the buffer must be big enough to hold these burst packets:

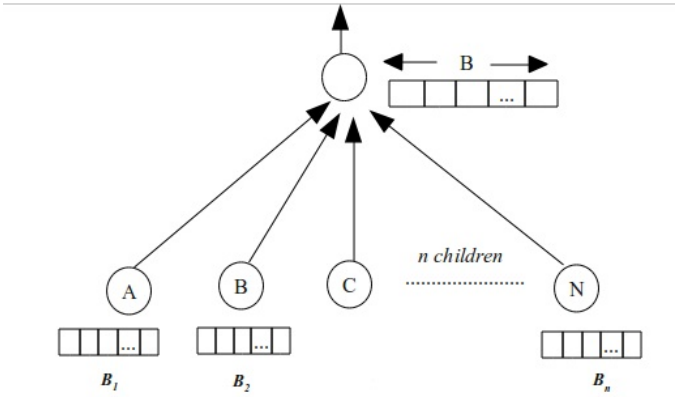

Figure 4.5: Buffer representation.

$$B \geq b_t.$$

Forwarding nodes will have higher amount of traffic because they must cater to bursts which children forward from lower nodes. This increases the buffer requirement for such nodes.

4.2.2 Burst from forwarding nodes

Before we look at the expression for deriving burst size, we discuss about the forwarding technique we will incorporate. The forwarding technique plays an important part in determining the buffer size. Further more the forwarding technique we choose helps us reduce buffer requirements. Let us assume we start with a simple tree based setup as shown in figure 4.5. Let the root node have n children with buffers sizes of $B_1, B_2, B_3 \cdots B_n$ respectively. The root node will have a buffer size of B. Let

B_f be the part of be which will accumulate all the forwarded packets. One simple way of forwarding would be to forward all the packets from the children to the parent simultaneously. This means the parent would have to accommodate the packets in the n buffers $B_1, B_2, B_3, \cdots B_n$, simultaneously. We have

$$B_f = B_1 + B_2 + B_3 + \cdots + B_n$$

Consider nodes in the upper levels of the tree which forward large quantities of data. This would increase the buffer sizes drastically because you will be fusing packets from the buffers of all the children. This will waste memory space.

We can also send one packet from buffer of each child, one at a time, wait for the buffer of the parent, to empty and perform the process again. The issue with this is that the buffer would be under utilized. We also find, that the time between two packet forwards will reduce considerably and result in blocking. This behavior would be similar to a single buffered approach. Better improvement to this would be to send k packets in each send. This will increase the time between two sends but it will increase buffer requirement to

$$B_f = nk$$

A good optimum solution would be to use as little buffer as possible and also provide enough time between two forwards so that we can buffer as much of the bursty traffic as possible. We use a heuristic approach to forward the packets. This is represented diagrammatically in the figure 4.6. The number of children the node has with a buffer

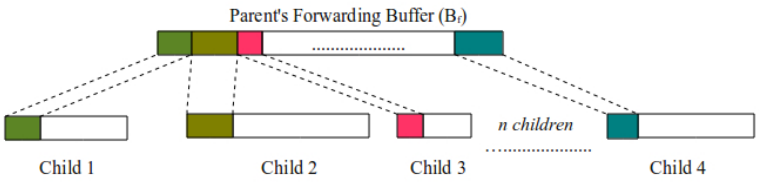

Figure 4.6: Optimum way of buffering.

of size greater than 1 is n_b. We achieve a smaller buffer size to capture forwarded packets by breaking the buffer of the children into several equal segments. We will consider the number of segments to be equal to the number of children the node has which have buffers limited to a minimum of 2. If a nodes has only one child we still consider the buffer of the child as the union of two separate equal segments.

$$B_f = \frac{B_1 + B_2 + B_3 + \cdots + B_n}{max(2, n_b)}$$

$$B_f = \frac{\sum_{i=1}^{n} B_i}{max(2, n_b)} \tag{4.9}$$

This implies that we utilize less buffer space if we split the sending of a bulk of packets into smaller segments. We will however reduce the time between two sends. This can have the implication that we might be blocking when we receive a packet and end up losing the packet. We draw a fine balance between the two parameters and try to get the best optimum solution.

Now that we have arrived at an optimum solution for the buffer size we can put together all the equations we have derived and write the effective buffer size for a node containing n children and n_b children with buffers, in general. We can hence write this as

$$B = B_f + b_t$$

$$\mathbf{B} = \frac{\sum_{i=1}^{n} B_i}{max(2, n_b)} + b_l \sum_{i=1}^{n} r_i \tag{4.10}$$

The expression obtained is a function of b_l. This is because the values of all buffers of the children will be a function of b_l. Our aim is to find b_l so we can calculate the burst sizes using the ratios. To do this we are given the largest buffer size we can use as B. We find out the largest buffer size being used in the network, which will be a function of b_l using the analysis described above. By equating the two values and solving for b_l we can obtain the value of the smallest burst size in the network. Using the b_l value we can obtain the respective burst sizes for each node.

We must also take note that leaf nodes don't need a buffer as they do not receive any packet. They merely generate packets. We must take care that we ignore leaf nodes when we are doing the above analysis and consider their buffer size to be 0.

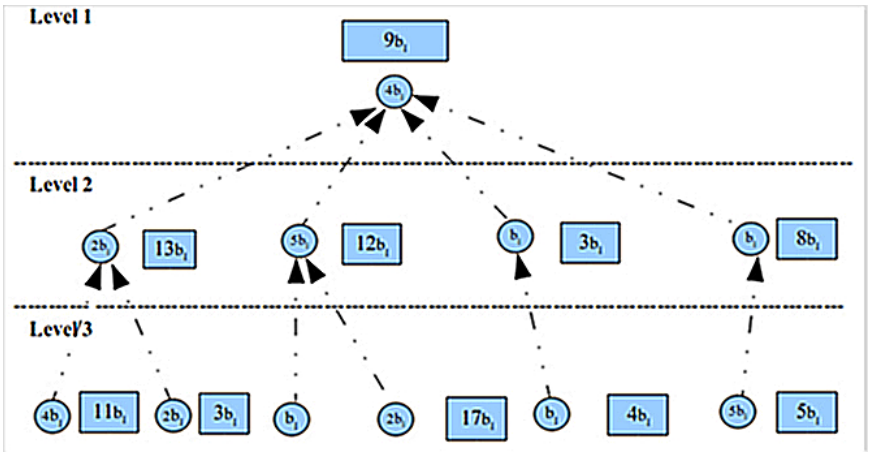

Figure 4.7: An example of how optimum burst size can be obtained.

4.2.3 Example Network

An example network which has the burst sizes and buffer sizes represented is shown in 4.7. Here we use b_l to represent the smallest burst. Initially we categorize all nodes as four different types based on the burst rate or bandwidth they may require. This process gives us the value of r_i which we discussed earlier and makes the burst size in the entire network a function of b_l. We consider r_i values to be 1,2,4 and 5 respectively in the above example. By applying the formula at each node we obtain results for the size of the buffer. We note that each node has the buffer size and burst size in terms of b_l. On observation we find that the highest buffer size we have consumed is $17b_l$. We then equate this to the given buffer size of say 15. We obtain b_l to be about 1 packet. The four burst sizes can hence be written as; 1, 2, 4 and 5 packets. Fractions can be rounded off to obtain a generic result.

4.3 Wait time analysis

Another parameter which is a function of the Rbuff size is the time the Rbuff holds packets before we pass the buffer to the application layer for forwarding and processing. Wait time is only significant for nodes which receive forwarded packets. The buffer size and wait time are inversely related. If a node has a smaller buffer it has to process much quicker and hence the time we can wait will be much shorter. If we increase the buffer size we can handle or backup more packets and hence we can wait longer to capture more bursts. In our analysis we will divide the approach into two parts. In the first part we will try to see how we can decide what should be the wait time that nodes, consisting of only leaf nodes as children, must send, given a fixed average time between two events. Once we establish the sending times of the lower nodes we will then proceed to find the time of the upper nodes which must gradually decrease so that it can accommodate the excess packets that arrive suddenly.

4.3.1 Lower nodes and leaf nodes

Nodes which are composed of only leaf nodes as children, in the network, will generally tend to have the least amount of packet and traffic activity. Further more leaf nodes tend to send data bursts randomly. We hence cannot have a deterministic time with which these nodes will send. Nodes which contain only leaf nodes however, receive random inputs but can send data out in a determined fashion. We can hence forward data in a deterministic fashion. We derive a wait time for such nodes. Let t_e be the average time between two events. In most cases the events can be modeled as a Poisson distribution. We are only interested in catering to one event. A single

event will lead to a single burst from all the leaf nodes in the same vicinity. The bursts are then collected by the buffers and then processed and forwarded. The time to do this must be such that it completes the forwarding of packets within the time the next event can possibly occur. Since we are dealing with random event we cannot pin point the exact time events will occur. We can however model our events and predict a suitable likelihood for time between two events which will work well for our purpose. Let us assume this time is denoted as t. The time t is hence the time within which there is very less likelihood that more than one burst will arrive. Since t is the maximum time the parent node can incur, it must also include the processing time of the parent. Let us denote the processing time the parent takes to process a single packet as t_p. Since we have to process B packets of the buffer, in total we will incur $t_p.B$ time to process the packets. Let us call the remaining time as t_w.

$$t_w = t - t_p B \tag{4.11}$$

We must now return to the method of forwarding we described in the previous section. Since the parent itself will form the child to a upper level node, we need to split the forwarding of the buffered packets into n_b parts. Here n_b represents the number of siblings the parent has which have a buffer size greater than zero. This value is subject to a minimum of value of 2. We need to do this because the forwarding of the packets in the buffer is done in similar segments. We can evenly distribute between each burst segment, the time wait as,

$$\frac{t_w}{max(2, n_b)} \tag{4.12}$$

4.3.2 Higher level nodes

The approach to higher level nodes is the same. The main difference will be the way we choose the value of t. We have to make sure that the value of t is less or equal to the least time the children take to forward a single burst. In general we can write,

$$t = min(t_1, t_2, t_3 \cdots t_n)$$

We proceed to find waiting time t_w and split t_w by using the method we mentioned for lower nodes. We can obtain t_w by using the equation (4.12) given above. We see that firstly, lower nodes send at much slower rates. This rate should be as high as possible but small enough to capture only one burst at any given moment, with highest likelihood specified. Secondly we will find that the wait time and burst sizes are dependent on the number of children. Nodes must forward faster and wait lesser time when the number of children are more. Thirdly, the buffer size and wait times are inversely related. If we want to reduce buffer space we reduce the time we wait and end up processing most of the time and waiting for very little time. If we want to increase wait times to capture more bursts we have to increase buffer which will reduce the maximum packet size we can possibly send.

Chapter 5

Experimental Results

In this section, we present experimental analysis of the Rbuff approach using the Contiki OS. We will first present the experimental setup and the platform we performed our tests on. We performed two specific type of analysis. First, we have test the optimum Rbuff size and examined average buffer utilization. Then we adjust the burst rate and sending times for a given fixed Rbuff length. We will show that we can achieve very good performance in terms of packet reception using any Rbuff.

5.1 Experimental Setup

5.1.1 Physical motes

We briefly describe some features of the TelosB motes. The motes are small sensor network nodes which run on the MSP430 micro-controller. They have a limited size of memory in the order of 512 kilobytes and vary for different types. Motes are equipped with a Zigbee based 802.15.4 radio and have an built-in Texas Instrument CC2420 radio chip capable of performing the physical layer operations of Zigbee. The

CC2420 radio has 128 bytes of buffer space. This space can be used to store incoming packets.

The motes contain a number of sensors including temperature sensors. They have in-built LEDs which can be used for debugging purposes. The motes are programmable through USB. The motes are equipped with flash memory for program memory. This flash memory can be written a very large number of times. Each mote has access to SPI bus. Debugging of printed text can be done through the SPI interface. When a "printf" statement is used, the mote uses the SPI to send the characters through the USB to the computer the mote is connected to. This allows powerful debugging and study of the motes operation.

To get an idea of the physical limitation of the operating systems radio stack on the mote we will also mention the send and receive times. A single send takes the order of 8 ms with all the protocol consideration taken into picture. A single receive will take about 10 to 15 ms to reach the "rime" layer of the Contiki operating system. Any higher level processing will depend on the application blocking time.

5.1.2 Simulation

We perform the experiments using the simulation tool bundled with Contiki OS. The simulator is developed in Java and is able to simulate the exact characteristics of an MSP430 micro-controller on the TelosB mote. The simulator represents each mote as a node, display the LED functionalities, simulate a unit disk based radio environment and also registers collisions occurring in the network and in the radio environment. The simulator is extremely powerful and can generate random nodes and simulate real-time operations [17].

We assign each node a unique ID number and use that node ID number to perform

routing operations for the packet transmission. We created a code template which contains a routing table and a set of parameters such as send rate, burst rate, burst size, and node level to make the entire process generalized. The routing table is used by each node to figure out the next node to which the packet has to be forwarded. We use unicasting to forward the packets.

In the experiments we cater to packets lost through collisions by observing the timeline. The timeline in the Contiki simulator marks collision events and we can use it to clearly identify packet loss due to collision.

5.2 Optimum Buffer Results

We began our evaluation by measuring the effect of a Rbuff on receiving packets in a single-hop network, to validate our claims that a optimum large Rbuff can reduce the packet drop rates considerably. We then evaluated Rbuff performance in a multiple-hop networks consisting of an unbalanced tree topology.

5.2.1 Single Hop Test

In order to validate the effect of Rbuff in a tightly controlled environment, we use a straightforward single-hop network topology to study the performance of Rbuff. The network consists of 5 senders and one receiver. We generate the burst traffic by letting 5 nodes send a set number of packets (average of 40 packets) in a short period (50 ms between each packet).

We need to find the optimal buffer size using equation we derived above. We assume that the number of packets from each sender during the burst period follows a normal distribution, $\mathcal{N}(\mu, \sigma)$. Here we set $\mu = 10$ and $\sigma = 5$. The senders transmit

packets at a constant rate of 2pps. Since we have 5 senders, the burst rate arriving at the receiver is 10pps. To mimic a random occurred event, the time when the burst packets are sent is randomized. Thus, in the worst case all senders are transmitting packets; thus, $(t_k - t_0)$ is the whole period to completing the burst traffic, ie., $\sum N/S_1$ and $S_1 = 5 \times 2\text{pps} = 10\text{pps}$.

In practice, the receiver node usually needs to process and forward the received packets, which is denoted by S_L in equation for optimum buffer. Thus, we can vary the burst load at the receiver by setting S_L as a fraction of the burst arriving rate (S_1).

We plot the optimal buffer size (B_{opt}) against different values of S_L in Figure 5.1. As shown in the figure, if $S_L = S_i$, the optimal buffer size is one as the rate of processing and forwarding packets at the receiver is fast enough to pull all the incoming packets from its radio driver. On the contrast, when S_L is reduced to $\frac{1}{10}$ of the burst rate, it requires a much larger buffer size to handle the excessive incoming packets.

We compare Rbuff and the existing Contiki's communication stack under three burst loads. In addition, to illustrate the effect of the buffer size, we set the actual buffer size as a fraction of the optimal value, e.g., $B = pB_{opt},\ p \in (0, 1]$. The results are shown in Figure 5.2 with varying p for Rbuff. Even $p = 0.5$, i.e., we use only half of the optimal buffer size, Rbuff improves the packet reception percentage by more than 50% under all the burst sending rates. The gain achieves more than 90%, when $p = 0.8$ and $p = 0.9$. For the optimal Rbuff size, we observe no packet drop rate. This confirms our analysis of the optimal Rbuff size, which is large enough to hold all the excessive packets. The second observation is that for the single buffer case, the packet reception ratio is increased from around 37% to 18%, while for Rbuff with

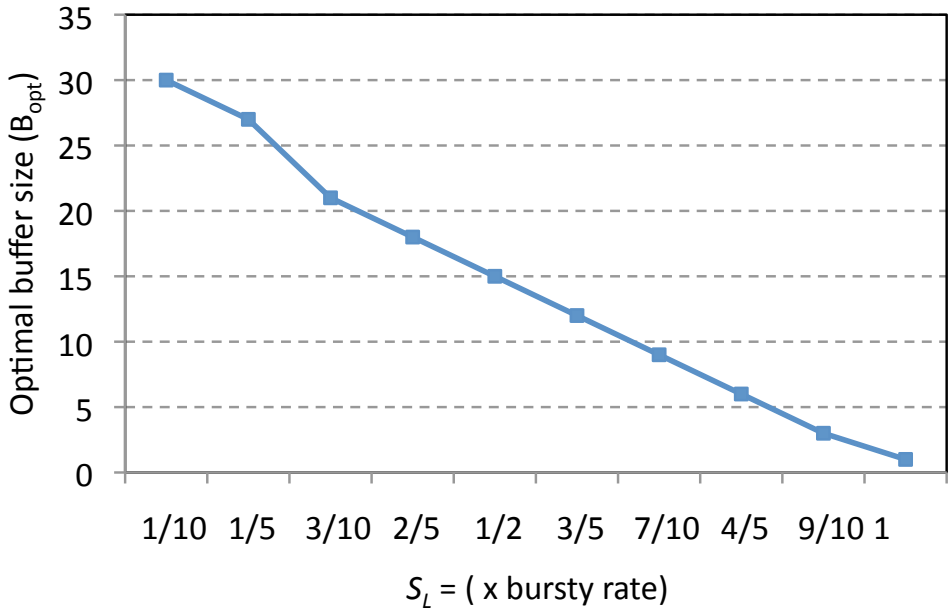

Figure 5.1: Optimal buffer size against S_L. x-axis is the fraction, denoting S_L relative to the burst rate.

all p, the effect of processing and forwarding rate (S_L) is reduced as p increases. For example, when $p = 0.5$ and 0.8, the differences of drop percentage between S_L=1 packet/sec and 4 packets/sec are 25% and 8%, respectively. All the three processing and forwarding rates have no packet dropped with the optimal Rbuff.

In conclusion, we can see that Rbuff with various p delivers a better performance than the default Contiki communication stack.

5.2.2 Multi-hop Test

In the second set of experiments, we use an unbalanced tree network topology to evaluate the performance of Rbuff in the multi-hop environment. The network topology is shown in Figure 5.3. All nodes, except the leaf nodes and the root,

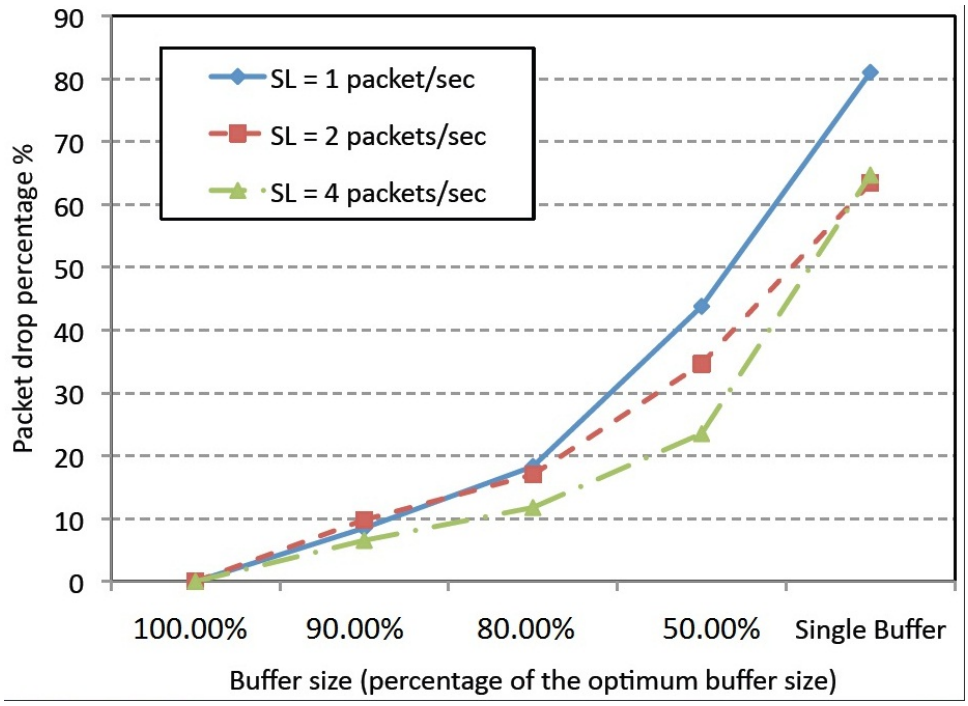

Figure 5.2: Packet reception ratio for different buffer sizes (in single-hop network test.)

forwards packets to the root. Data packets are sent from the leaf nodes at a constant rate of 2pps and follows the normal distribution with $\mu = 6, \sigma = 3$. Since the burst arriving rates to each intermediate node depends on the number of children, we calculate the optimal buffer size for the intermediate nodes individually.

As in the previous case, the results of packet reception ratio at the root node for varying S_L and buffer sizes are plotted in Figure 5.4. The results exhibit two similar patterns as in the single-hop case. First, we find that using half of the optimal buffer size cant reduce the packet drop rate by around 40%. Second, the Rbuff succeeds in reducing the packet drop rate as the S_L is decreased for each buffer size. However, the improvement from $p = 0.8$ to $p = 1$ in the multi-hop case is less than that of

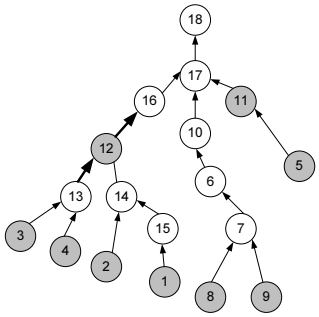

Figure 5.3: Unbalanced tree topology

the single-hop case. This is because for the single-hop case, the optimal buffer size is calculated based on its children, which are the only senders, while the child of the root in Figure 5.3 aggregate the packets from three paths with different burst sizes. Thus, the optimal buffer size on the root may not be large enough to hold the excessive packets from all the senders.

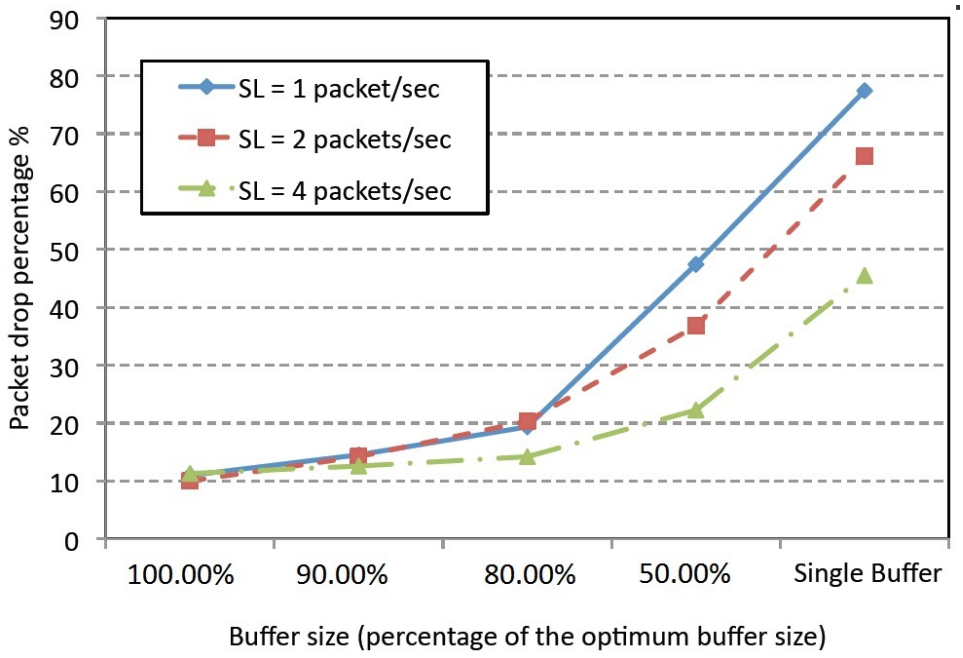

Figure 5.4: Packet reception ratio for different buffer sizes (in the multi-hop unbalanced tree topology)

5.3 Fixed Buffer Results

We have verified the benefits of the fixed Rbuff based approach. Given a fixed buffer size we can calculate the burst sizes and wait times. In the next set of experiments we verify, that the fixed buffer approach we described earlier, is more efficient than a single buffer approach, in conditions where application is blocking and traffic is bursty.

5.3.1 Experimental Setup

The experimental setup consists of 8 nodes. The 8 nodes are arranged as an unbalanced tree. Each node sends a fixed burst size represented as a ratio of the

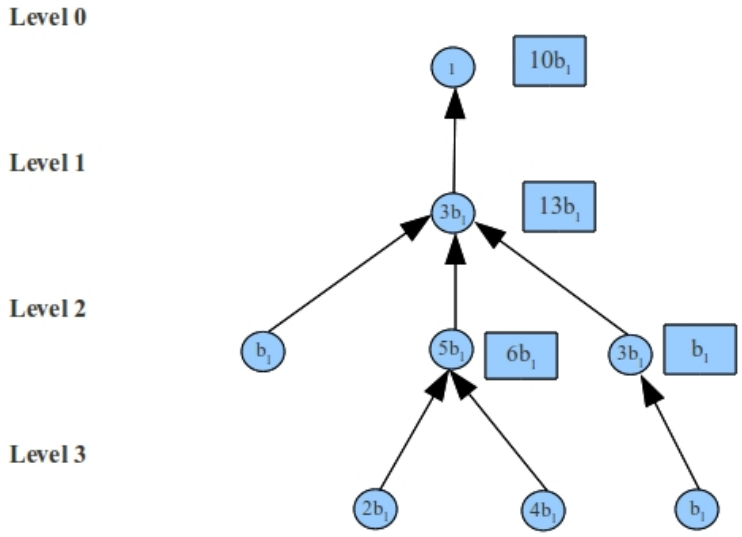

Figure 5.5: The test setup for fixed buffer, consisting of 8 nodes in an unbalanced tree.

smallest burst size (b_l). Nodes at level 2 and above have buffers to hold the bursts generated by their children. We hence calculate the buffer sizes for each node.

In our experiments we selected various parameters for the above network which we used to derive buffer sizes and the forwarding times. We select buffer sizes of 12, 15 and 18 packets. Considering a burst size of b_l we calculate buffer size in terms of burst size we need to store. The solution for the buffer sizes is in figure 5.5. By equating this to maximum buffer size we are permitted we find the burst sizes. In some cases we find that the burst size is a fraction. To represent such burst sizes we extend and change the wait times of these nodes such that we obtain whole number values for their burst size.

We also have to find out how many events will occur in total and the interval

between these events. The event times are generated by a random Poisson distribution which has a fixed mean time. We vary the mean time to different values such as 6 and 10 seconds. We also generate ten random intervals which are about the mean times generated above. We find the wait times for each node based on this event time. For wait time we consider a value equal to about two-third the average event time. We further calculate the wait times for level 2 and level 3 nodes based on the wait times for level 1 nodes. We make use of our model to derive these values.

To make the events localized we trigger the bursts from selected nodes only. The set of nodes which are selected for being triggered are also random. This simulates the real world sensor environment. Overall we run several different random combinations of the localized set and the randomly generated event times. We then average the results to get the overall performance of the entire network. We perform several experiments to show that.

5.3.2 Latency

We define latency as the time it takes a single packet to reach its destination from its source. We consider latency as it is an important factor in the network when real-time delivery is significant. An application that would require such a real time requirement is voice.

Latency in our network depends on the wait times. Since we wait to buffer packets, we do not dispatch the packet immediately and make the packet wait. This increases the latency in buffered approach. This is not a problem in single buffered approach, because the packet is immediately forwarded. We can improve latency by reducing the wait time between processing. This will however increase the chance we will drop packets due to application blocking.

Burst Rate	Single Buffer	Multi – Buffer	
16 pkts/sec	41.5	474	***ms***
32 pkts/sec	73.5	421	***ms***
64 pkts/sec	88.25	432	***ms***
128 pkts/sec	97	451	***ms***

Figure 5.6: Table showing latency for different burst rate for single and multi-buffered approach.

We perform the experiment by assuming an application processing time of 36 micro seconds. We also assume the time between events is 6 seconds on average. We generate a Poisson distributed series of random values with this as the mean value.

Experimental results are shown in figure 5.6 show that for higher burst rates the amount of latency is almost 4 seconds. This is explained as the wait times we consider is close to around 4 seconds ($\frac{2}{3}$ of 6 seconds) . At lower nodes we wait for 4 seconds before forwarding the packet to capture at least one event. In the case of lower burst rates (such as 16 $pkts/sec$), since we receive lesser packets we forward the packet earlier. The packet gets forwarded in the wait period it is received in. In later cases (higher burst rates) the packets are stored for delivery in the next wait time period as the buffer is filled quickly.

In the case of single buffer approach the packet is immediately sent up the tree. The latency is very small and in the order of the application processing time. We see at maximum a latency of about 100 ms.

5.3.3 Collisions

Collisions occur when multiple nodes send simultaneously. When nodes sending bursts and those nodes forwarding data send almost simultaneously we lose the packet due to collision and expend more energy sending the packet again. Collisions are very

Figure 5.7: The traffic between two events when single buffer is used.

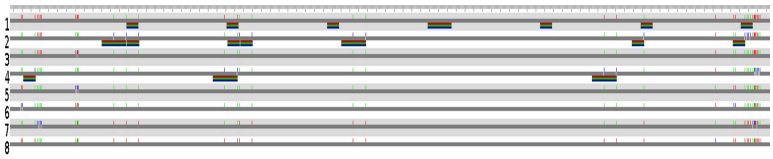

Figure 5.8: The traffic between two events when fixed-buffer is used.

common in bursty traffic because many nodes send data simultaneously.

We observe in the case of single buffer that the traffic is clustered to the time the event occurs. In the figure 5.7, we observe how packet traffic is sent between two events. We see that since packets are send immediately to the parent, we see a distinct band of radio silence separating two sets of events. During this period there is no packet traffic.

In contrast in our approach by buffered forwarding, produces a more distributed radio traffic as shown in figure 5.8. We see that the radio silence which existed in the single buffer cases is filled with areas where packets are being forwarded. We perform the experiment to measure collision by assuming an application processing time of 36 micro seconds. We test for burst rates of 16, 32, 64 and 128 pkts/sec. The result is as shown in figure 5.9

We find that in most cases collision is very high in single buffered implementation because of the clustering effect we observed above. The distribution of the packets, the amount of collision in fixed-buffer case is more reduced. We also find that at high bursts rates the collision loss is smaller. This is because the burst rate becomes very

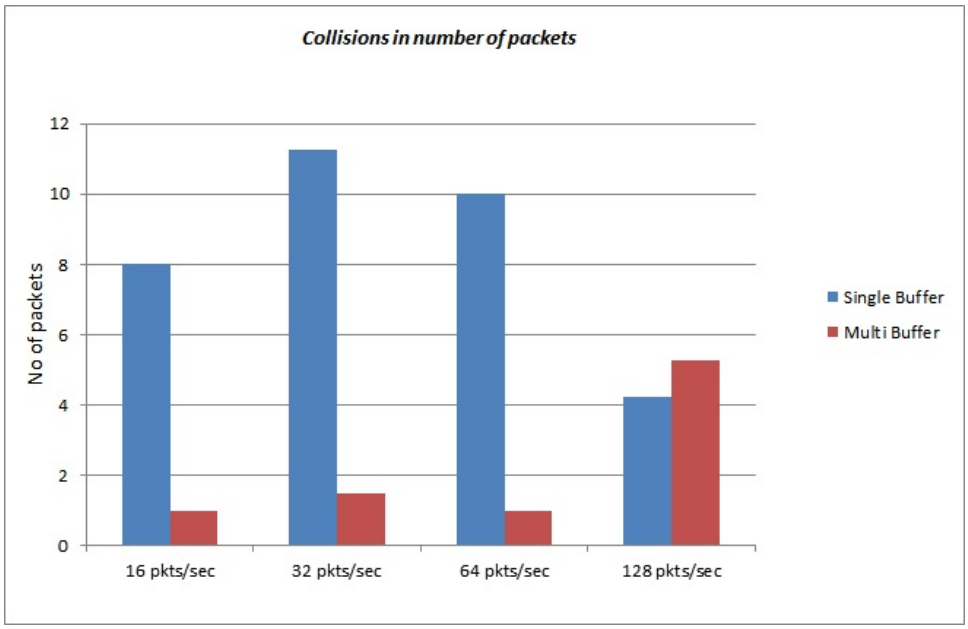

Figure 5.9: Number of collisions in both cases

small compared to the forwarding rate and hence collision between burst packets and forwarded packets is reduced. We also see increased collision at higher burst rates in fixed buffer case. This is due to the larger influx of packets and longer processing time which collide with the next set of bursts. In general fixed buffer implementation is better at reducing collision than single buffer implementation. Therefore, the implementation is more energy conservative.

5.3.4 Packet loss with varying burst size

The network we tested on represents a good estimate for burst size in the network which will provide acceptable packet loss for given constraints in buffer memory. We however want to study how the network will behave if we increase the burst traffic without increasing the buffer size.

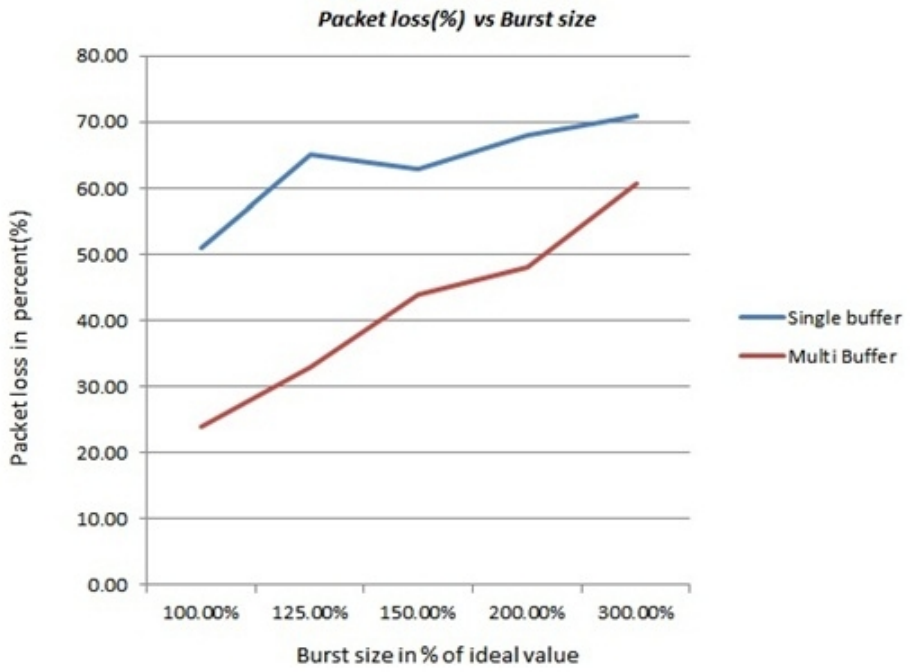

Figure 5.10: Packet loss versus the burst size in percentage

To see how our buffered approach performs over single buffer approach, we try to increase burst size proportionally. We initially begin with a burst size of 100 percent of the calculated optimal burst size. We increment the burst size by 125, 150 200 and 300 percent and then see how both the setups perform. We perform the experiment to measure collision by assuming an application processing time of 36 micro seconds. We test for burst rate of 64 pkts/sec. The result is as shown in figure5.10 First, we observe that we experience loss of about 20-30 percent in the most optimal case. This is because the events occur randomly and tend to occur asynchronous to the wait time periods. In some cases, nodes end up processing packets when packets are arriving. Due to the randomness of the network this cannot be avoided. Furthermore we also lose packets due to the collision we explained earlier.

We also find that as burst size increases the amount of loss sustained by the fixed buffer model is more sharper. In the case of single buffer, the loss is very drastic initially but later becomes more stable (flatter slope in the graph). In the fixed buffer implementation the loss rises sharply and tends to become equivalent to the single buffer case at high burst sizes. This is because in at high burst sizes the buffer is completely occupied and the nodes behave very similar to the single buffer implementation.

5.3.5 Burst Rate performance

The purpose of buffering is to handle high burst rate traffic. When traffic burst rate and the application processing time is large, then the application will drop packets if only a single buffer is used. We can clearly see that a buffered approach would improve the performance when burst rate is so high, that the time difference between two packets arriving is much smaller than the application processing time. We perform our tests with a fixed application processing time of about 36 ms. We also keep burst sizes at 100 percent optimum value. We vary the burst rate between 16, 32, 64 and 128 pkts/sec. At around 32 packets per second the time between two packet arrivals just about coincide with the time it takes to process a single packet. The result as shown in figure 5.11.

The single buffer approach actually performs better at low burst rates. This is explained by the lower application processing time compared to the forwarding time. At 16 pkts/sec two packets are sent with a very large time period of about 60 ms. Since the application processing time is small (38 ms) the forwarding happens faster than the burst. Fixed buffer however experiences a slight reduction in performance as the burst period is very long and results in potential overlap between forwarding

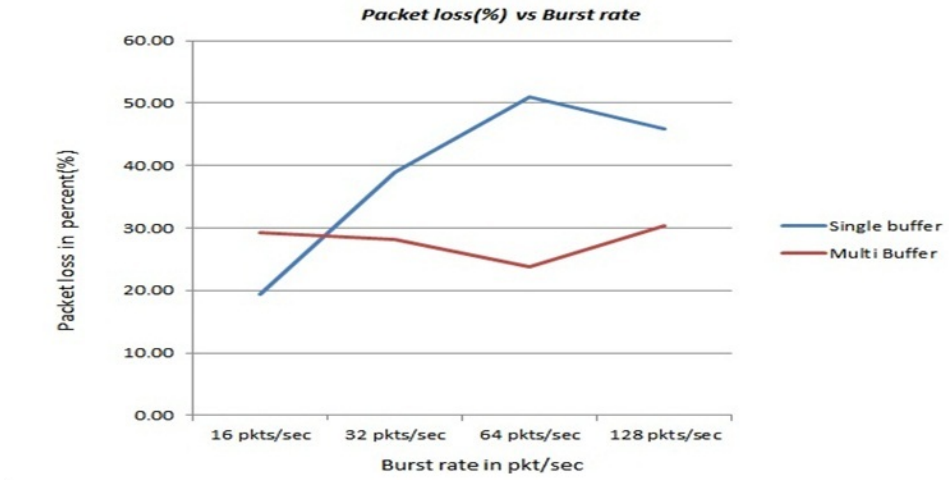

Figure 5.11: Packet loss versus the burst rate

of upper nodes and the next burst of the lower nodes.

However as the burst rate rises the we find the single buffer implementation deteriorate rapidly in terms of packet reception. The fixed buffer however maintains a stable packet loss at even higher rates. This is due to the buffering of the bursts and the post processing of the packets. Traditionally application requiring higher burst rates will also require larger processing time per packet for decoding purposes like in the case of video. However a single buffer approach would give a better result at high burst rates, only if the application processing time is very small. This is why single buffer implementation is less efficient as lowering processing time is difficult at high burst rates.

5.3.6 Packet loss with varying application processing time

A blocking application can lead to packet drop. When an application is processing we cannot receive any packets. Because we buffer the packets we are able

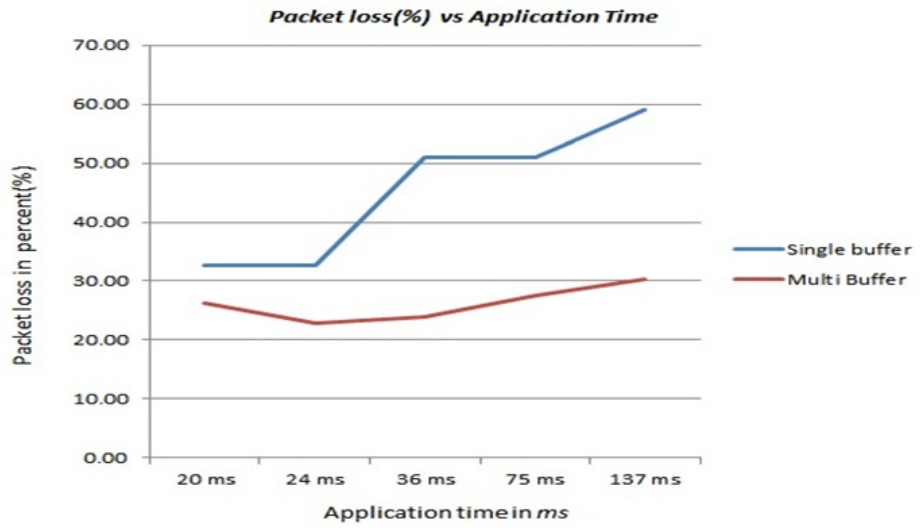

Figure 5.12: Packet loss versus the application processing time

to hold more packets before application begins execution in our fixed buffer model. If burst sizes are large and burst rates are high, we encounter very high packet drop as we have shown in single buffer case. Longer application time has an effect on the packet loss for single buffer case.

We make the application perform for a longer time and block the processor. We then test for different application times in both single buffer case and fixed buffer case. We keep the burst rate at 64 pkts/sec and the burst sizes at 100 percent of the optimal values. We vary the application times between 20, 24, 36, 75 and 137 ms. We find the result in figure 5.12

We clearly find that at low application time single buffer model approaches the multi-buffer model. In fact at lower burst rates and lower application time the single buffer approach would perform as well or in some cases better than the fixed buffered approach. As the application time increases the packet loss in the former case begins increasing. In the case of fixed buffer we see the packet loss virtually remain stable.

The buffered approach hence stymies the effect of a blocking application. Only at very large application processing times (which almost near the wait time) the fixed buffer approach shows increase in packet drop.

5.4 Random Network Test

The last experiment we perform is a large randomly generated network topology, which is shown in Figure 5.13. The setup consists of 50 nodes and one sink node. Data from all nodes are routed to the sink node, which is shown in the red circle. We consider the scenario that there are several interesting events happened at some random locations in the network. When the nodes around detect the events, they start to send packets along the forwarding paths to the sink. Forwarding path is shown as grey line with arrow in the figure.

Note that routing paths from two different source nodes may joint at the same forwarding nodes as we can see in Figure 5.13.

Each event induces a burst of packets from all nodes around. Like the above two experiments, we also assume that the burst size of each source node follows a normal distribution with certain mean and variance. All sending and forwarding nodes transmit packets at a constant rate of 1 packet per second. We do not choose a high rate because a high rate would cause a large number of collision due to the density of the network, and makes it difficult to distinguish from packet drop due to the inadequate buffer size.

To verify that Rbuff is able to hold excessive packets under burst traffic situations, we chose to measure the buffer occupancy of each node. The reason is that by this point, we have already seen that Rbuff can improve the packet drop rate

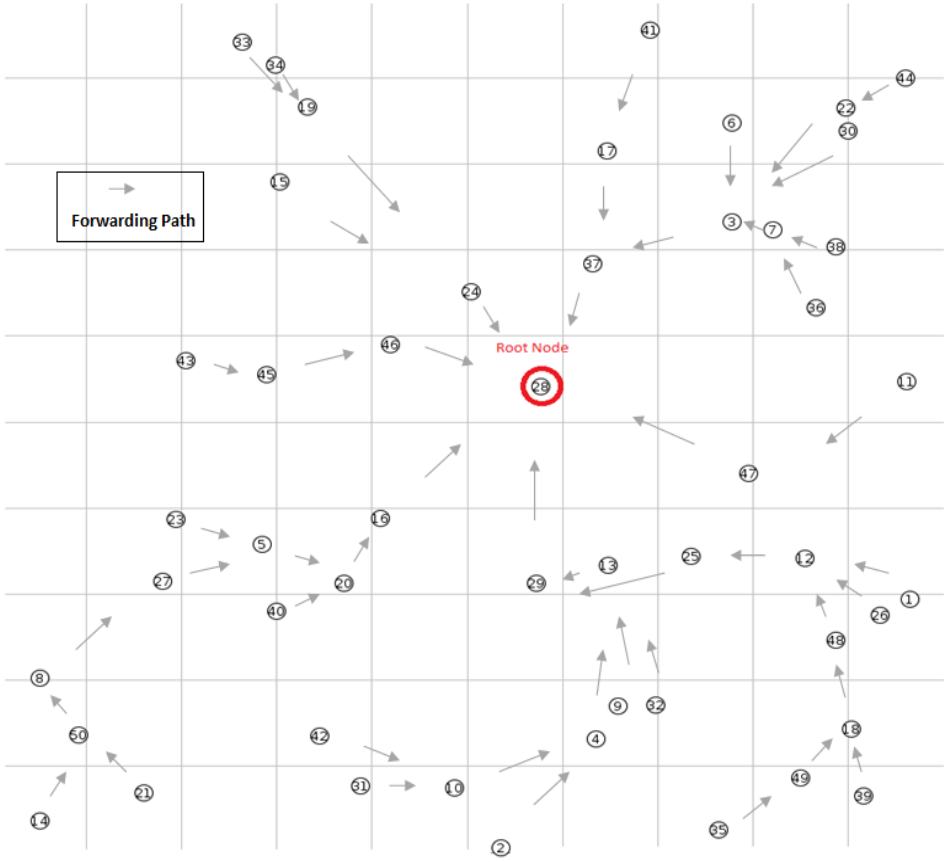

Figure 5.13: A random network topology.

through the previous experiments. We have also seen that a fixed buffer approach performs better in most cases compared to a single buffer approach. In those more controlled environments, however, because the burst traffic from the source nodes are homogeneous in terms of number of forwarding nodes and network topology, we can expect that the buffer occupancy of each node to be almost the same. In this random network, because of the randomness of events and the heterogeneous forwarding paths, it is helpful for us to understand the behavior of Rbuff by measuring the buffer occupancy of each node in the network.

We vary the burst size by setting $\mu = 6, 8, 10$, and $\sigma = 3, 4, 5$, respectively. We repeat each burst size 5 times and calculate the average. The results are shown in Figure 5.14. In each figure, we plot the occupancy of Rbuff for each node, which we defined as the percentage of used buffer of the node.

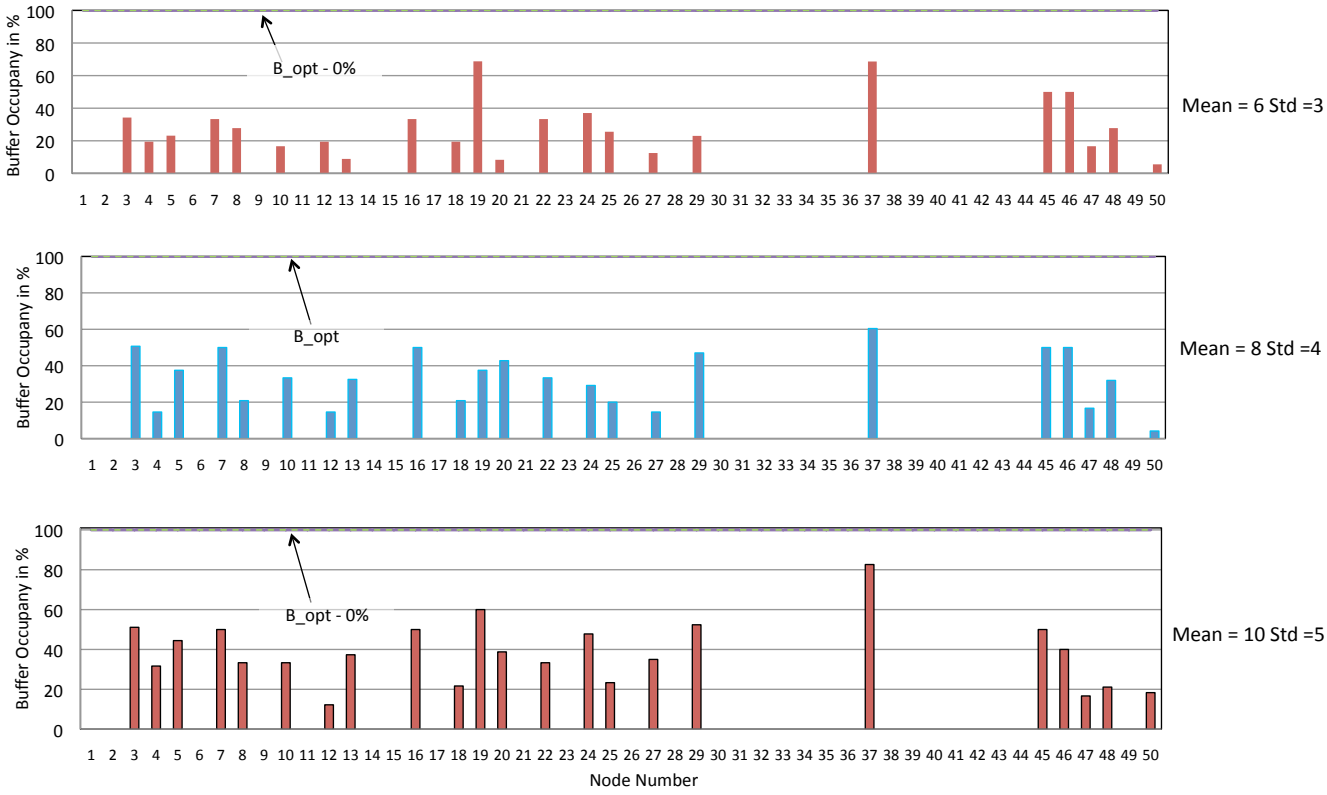

Figure 5.14: Buffer behavior in the random network with respect to different burst traffic

We see that the occupancy percentage of each node varies from 0% to around 60%. Nodes, which are neither the source nor on the forwarding path, does not use the Rbuff, having a 0% occupancy. Otherwise, the occupancy of nodes depends on its position on the forwarding path. For example, most source nodes have the occupancy percentage around 20%, while the forwarding nodes have higher percentages as burst

traffics aggregate at the forwarding node. It is also interesting to note that nodes in the case of (μ=10, σ=5) have higher occupancy Rbuff than those in (μ=6, σ=3), because we have more burst traffics with a larger mean and variance of the number of packets from the source nodes.

Finally, we observe that the buffer size is over-provisioned as the optimal buffer size is not fully used. This is because we average the buffer occupancy over a number of trials. Events occur randomly and it is not necessary for a given node to forward data to the root all the time. On an average case over multiple runs we will find that buffer usage is mostly, well below 100 percent and averages at 20 to 30%.

Nonetheless, we believe that the buffer size could be determined either based on the network traffic in a specific application, or configured at run-time. Both have its pros and cons. For example, although the run-time configuration of the Rbuff size does not waste unused memory allocated for Rbuff, it may not be adequate to handle sudden burst traffic when workload changes drastically. On the other hand, the static Rbuff is not flexible enough because we need to reprogramming the sensor nodes as the expected workload changes.

Chapter 6

Related Work

Sensor Networks has become a growing field of study in the last decade. Monitoring and detection has improved thanks to vast advances in sensor technology and hardware. Volcano monitoring [6],[7],[8] , Habitat monitoring for migratory birds and animals [3],[4],[5] Wearable Computing [10], [11],[12], Health Monitoring [1], [2], Intrusion Detection [18],[19] and Industrial Monitoring [13] are some of the common applications sensor networks find use for in the modern world. As sensor networks become more popular they are ushering in a new era of computing which is ubiquitous and pervasive in nature [20],[21]. As these the applications become more complex demand for data from these networks will increase. With voice and video becoming available through these sensors, data rates will eventually increase and data sizes to be transferred will go up drastically. Higher data rates will lead to higher collision, data loss and congestion. We find that congestion still remains a very serious problem in the research community. Increasing the number of sensors for sensing will result in much better resolution and accuracy of data but will inevitably lead to congestion. It becomes a very prominent issue when data is bursty and large number of nodes

try to forward data to a common sink. In [22] the author explains how congestion is a predominant problem in sensor based systems along with collision. He mentions a form of congestion control is necessary to improve throughput. A large amount of research has been dedicated to to finding a solution to the problem of congestion and improving packet reception rate [23], [24], [25], [26], [27]. One approach to addressing the problem are MAC layer protocols [28], [29], [30], [31], [32] which try to provide collision-free communication mechanisms to sensor network application. Various MAC layer implementations have been proposed. The synchronous protocols T-MAC [31] and S-MAC [32] allow energy efficiency by synchronizing the wake time of the nodes. During the period they are awake the packet is sent and recieved and collision is avoided by the use of RTS and CTS acknowledgement. In T-MAC [31] the wake period is adaptive, leading to a five fold improvement in performance. Some of the MAC protocols, such as B-MAC [28], *Wise*MAC [29] and X-MAC [30], use asynchronous duty cycle approach to let nodes periodically wake up and communicate with each other. These protocols achieve energy efficiency under light traffic, while the long occupancy of the preamble packets until the actual data delivery makes them inadequate in case of bursty traffic. B-MAC [28] uses Clear channel assessment to decide if the medium is free for transmission. It also implements a variable Low Power Listening (LPL) duty cycle. Using this variable duty cycle, the time for which the radio is switched on, for idle listening, is adjusted. *Wise*MAC [29] makes use of the assumption that access points have unrestricted power supply. Thus the access points regularly wake the nodes from sleep, thus try to reduce amount of energy consumed and avoid collision. In X-MAC [30] the message preamble is made much smaller. The initiating preamble is converted to a train of short pulses reducing radio usage. When the listener wakes up from sleep the header is read and the listener

node responds. Some MAC protocols [33, 34] use a receiver-initiated transmission such that the sender node stays actively silently until it receives the beacon message from the intended receiver node. This results in reducing the amount of time a pair of nodes occupy the medium and allow more of the contended data to be sent during traffic bursts. Another work in which media access based technique is used is [35]. It tries to provide reliable packet delivery using adaptive rate control. It focuses more on access control and less on Congestion reduction. Some solutions implement efficient algorithms and protocols, within the network, which try to avoid congestion. Most of this is implemented in transport or network layer. In [36] they use three specific steps to avoid congestion. They propose that *i*) Congestion is detected using channel sensing *ii*) the status of the network is sent back to source termed backpressure *iii*) Enables acknowledgements from sink which is used to reduce the rate of the source to counter the congestion occurring. Their approach is more holistic, characterized by the way the system as a whole behaves. Many transport layer protocols have also been developed to reduce congestion such as [37], [23], [24], [27], [38], [25], [26] and [39]. In [37], they propose congestion control to reduce congestion. They also claim to reduce the energy expended based on controlling the acknowledgement rate to an extent where meaningful data from the collective set can be obtained to required accuracy. The authors in [27], deal with eliminating congestion by leading the data to areas of lesser activity. The process is called Virtual sinking. A generic TCP based transport layer protocol using congestion control is used in [24] to reduce congestion. *"Directed diffusion"* is used in [23] to improve congestion. Data obtained is tagged by attributes and caching of the data is done using the attributes based on interests. Nodes of a particular interest get data of that attribute forwarded to them. Other nodes having the same interest obtain their data former node's cached data. In [38]

the author tries to use information about the number of downstream and upstream nodes to reduce congestion. The congestion levels are found out using the buffer levels. RCRT [25] is a reliable transport protocol, which adaptively detects and controls the congestion at the sink nodes to avoid congestion collapse. Further, Zhang *et al.* in [26] designed a window-less block acknowledgement scheme, called RBC, to address the congestion and contention of bursty traffic in multi-hop sensor networks. Approach in [39], uses a moving sink to tackle the congestion problem. By distributing the sink among differnt nodes they prevent crowding of data towards a single node. The above improvements are based on the fact that the packet loss is due to wireless medium and collision [40]. They also do not offer a generic solution to problems in sensor networks as their approach is specific to a certain protocol. The proposed Rbuff in this paper is orthogonal to all those approaches, as we consider the incoming packet has already arrived at the radio driver of the sensor node and the software communication stack causes the packet drop. The idea of using a buffer to hold ingress packets in the memory buffer is not new as it appears in Linux and other desktop operating systems [41]. Some amount of insight into buffer performance and buffer management has been illustrated in [27]. It gives very detailed explanation of the buffer limits using mathematical Markov chain simulation. They discuss fairness of the buffer and also how the size of the buffer can be used to perform congestion control. Our work is closely related to their findings and adds more weight to a buffer based approach above MAC layer to capture fast packets and avoid drops due to quick packets. Our work makes the first attempt to add this idea in the resource constrained sensor OS for handling bursty traffic. For example, in the Linux kernel, an incoming packet at the NIC is first placed in a ring buffer, which belongs to the kernel address space. The used space of the buffer is emptied when the packet is processed by higher layers;

incoming packets are dropped if the buffer becomes full. To avoid extra memory copies, packets remains in the buffer when being processed. Sensor OSs differ from general-purpose OSs, e.g., no separation between user and kernel space, constrained memory size. Besides TinyOS and Contiki, there have been many other sensor OSs dedicated for sensor nodes such as SOS, MantisOS [15], and Nano-rk [16]. Although they use either the event-driven mode, thread mode, or the combination of both, they all use the single fixed memory space. As the data rate and burst sizes increase the single memory buffer model will have much lower throughput. These sensor OSs, with their current radio stack design, will not be suitable for burst data or high data rates. Our study shows that independent of the type of Sensor OS we consider, a multi-buffer based approach at MAC/network level improves the performance (in terms of packet reception) by up to 50 percent. Hence we find that this approach is a very good way of reducing congestion in sensor networks which occur due to burst traffic.

Chapter 7

Conclusion

Modern day sensing requirements have grown to encompass highly bursty traffic applications such as video streaming and voice transmission. Such applications require large amount of processing and also produce a large burst of traffic. In our work we consider the high data rate requirements of such devices and demonstrate the failure of existing sensor operating systems in handling such high burst rate traffic.

Our study shows that we can achieve greater reception quality with the help of multi-packet buffer (Rbuff) over the current single slot implementation. Our model can be used to obtain a very accurate buffer size that would be required to reduce packet loss to less than 10 percent. We observe that in most cases, the optimal buffer is not used completely. Through our experiments, using a buffer of optimal size, we have shown that we can reduce packet loss significantly, up to 60%. We understand that memory in such systems is a luxury. Therefore we consider the practical limitations of a sensor node having limited memory and processing capabilities. We show that, given a set limited memory constraint on buffer size, we can calculate an ideal burst size and processing wait time for each node. Experimentation on the Contiki operating

system clearly show that in case of such high rate and bursty traffic, a conservative buffered approach provides superior packet reception when compared to a single buffer approach. Our experiments show that even in memory limited scenarios the proposed Rbuff approach performs significantly better at high burst rates and longer application processing times than the single buffer implementations. We also achieve considerable reduction in collisions improving the energy efficiency of the network. We observe up to a 25% improvement in packet reception with a moderate fixed size buffer.

Bibliography

[1] Alexandros Pantelopoulos and Nikolaos G. Bourbakis, "A survey on wearable sensor-based systems for health monitoring and prognosis," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, pp. 1 –12, 2010.

[2] S. Rost and H. Balakrishnan, "Memento: A health monitoring system for wireless sensor networks," in *Sensor and Ad Hoc Communications and Networks, 2006. SECON '06. 2006 3rd Annual IEEE Communications Society on*, 2006, vol. 2, pp. 575 –584.

[3] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, 2002, pp. 88–97.

[4] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin, "Habitat monitoring with sensor networks," *Commun. ACM*, pp. 34–40, June 2004.

[5] Hanbiao Wang, Deborah Estrin, and Lewis Girod, "Preprocessing in a tiered sensor network for habitat monitoring," *EURASIP J. Appl. Signal Process.*, pp. 392–401, January 2003.

[6] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 381–396.

[7] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 381–396.

[8] Wen-Zhan Song, Renjie Huang, Mingsen Xu, Andy Ma, Behrooz Shirazi, and Richard LaHusen, "Air-dropped sensor network for real-time high-fidelity volcano monitoring," in *Proceedings of the 7th international conference on Mobile systems, applications, and services*, 2009, pp. 305–318.

[9] Mitsugu Terada, "Application of zigbee sensor network to data acquisition and monitoring," *Measurement Science Review*, pp. 183–186, January 2009.

[10] Jovanov Martin and Raskovic, "Issues in wearable computing for medical monitoring applications: a case study of a wearable ecg monitoring device," in *Wearable Computers, 2000. The Fourth International Symposium*, 2000, pp. 43 –49.

[11] Dirk Trossen, Dana Pavel, Glenn Platt, Joshua Wall, Philip Valencia, Corey A. Graves, Myrna S. Zamarripa, Victor M. Gonzalez, Jesus Favela, Erik L?vquist,

and Zsuzsanna Kulcs?, "Sensor networks, wearable computing, and healthcare applications," *IEEE Pervasive Computing*, vol. 6, pp. 58–61, 2007.

[12] James A. Davis, Andrew H. Fagg, and Brian N. Levine, "Wearable computers as packet transport mechanisms in highly-partitioned ad-hoc networks," *Wearable Computers, IEEE International Symposium*, vol. 0, pp. 141, 2001.

[13] V.C. Gungor and G.P. Hancke, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *Industrial Electronics, IEEE Transactions on*, vol. 56, no. 10, pp. 4258 –4265, 2009.

[14] M Huang, "Visualization of urban transportation data generated by wireless sensor network using modern approaches," .

[15] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han, "Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms," *Mobile and Network Applications*, vol. 10, pp. 563–579, 2005.

[16] Anand Eswaran, Anthony Rowe, and Raj Rajkumar, "Nano-rk: An energy-aware resource-centric rtos for sensor networks," in *IEEE International Real-Time Systems Symposium*, 2005, pp. 256–265.

[17] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón, "Cooja/mspsim: interoperability testing for wireless sensor networks," 2009, pp. 27:1–27:7.

[18] Ana Paula R. da Silva, Marcelo H. T. Martins, Bruno P. S. Rocha, Antonio A. F. Loureiro, Linnyer B. Ruiz, and Hao Chi Wong, "Decentralized intrusion detection

in wireless sensor networks," in *Proceedings of the 1st ACM international workshop on Quality of service & security in wireless and mobile networks*, 2005, pp. 16–23.

[19] Chao Gui and Prasant Mohapatra, "Power conservation and quality of surveillance in target tracking sensor networks," in *Proceedings of the 10th annual international conference on Mobile computing and networking*, 2004, pp. 129–143.

[20] Satyanarayanan, "Pervasive computing: vision and challenges," *Personal Communications, IEEE*, pp. 10 –17, 2001.

[21] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme, "Connecting the physical world with pervasive networks," *IEEE Pervasive Computing*, vol. 1, pp. 59–69, 2002.

[22] Sameer Tilak, Nael B. Abu-Ghazaleh, Wendi Heinzelman, Sameer Tilak , Nael B. Abu ghazaleh , and Wendi Heinzelman , "Infrastructure tradeoffs for sensor networks," 2002.

[23] Ramesh Govindan Chalermek Intanagonwiwat and Deborah Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Annual International Conference on Mobile Computing and Networking*, 2000, pp. 56 – 67.

[24] S. Venkatesan Yogesh G. Iyer, Shashidhar Gandham, "Tcp: A generic transport layer protocol for wireless sensor networks," in *International Conference on Computer Communications and Networks*, 2005, pp. 449 – 454.

[25] Jeongyeup Paek and Ramesh Govindan, "RCRT: rate-controlled reliable transport for wireless sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2007, pp. 305–319.

[26] Hongwei Zhang, Anish Arora, Young ri Choi, and Mohamed G. Gouda, "Reliable bursty convergecast in wireless sensor networks," in *The ACM International Symposium on Mobile Ad Hoc Networking and Computin*, 2005, pp. 266–276.

[27] Andrew T. Campbell Jon Crowcroft Wan, Shane B. Eisenman, "Siphon: Overload traffic management using multiradio virtual sinks in sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2005, pp. 221 – 235.

[28] Joseph Polastre, Jason Hill, and David Culler, "Versatile low power media access for wireless sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2004, pp. 95–107.

[29] Amre El-Hoiydi and Jean-Dominique Decotignie, "Low power downlink mac protocols for infrastructure wireless sensor networks," in *ACM Mobile Networks and Applications*, 2005, pp. 675 – 690.

[30] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han, "X-mac: A short preamble mac protocol for duty-cycled wireless sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2006, pp. 307–320.

[31] Tijs van Dam and Koen Langendoen, "An adaptive energy efficient mac protocol for wireless sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2003, pp. 17 – 180.

[32] Wei Ye, John Heidemann, and Deborah Estrin, "An energy efficient mac protocol for wireless sensor networks," in *IEEE International Conference on Computer Communications*, 2002, pp. 1567–1576.

[33] Yanjun Sun, Omer Gurewitz, and David johnson, "RI-MAC: A receiver-initiated asynchronous duty cycle MAC protocol for dynamic traffic loads in wireless sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2008, pp. 1 – 14.

[34] Prabal Dutta, Stephen Dawson-Haggerty, Yin Chen, Chieh-Jan Liang, and Andreas Terzis, "Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless," in *ACM Conference on Embedded Networked Sensor Systems*, 2010, pp. 1 – 14.

[35] Alec Woo and David E. Culler, "A transmission control scheme for media access in sensor networks," in *Annual International Conference on Mobile Computing and Networking*, 2001, pp. 221 – 235.

[36] Andrew T. Campbell Chieh-Yih Wan, Shane B. Eisenman, "Coda: Congestion detection and avoidance in sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2003, pp. 266–279.

[37] Ian F. Akyildiz Yogesh Sankarasubramaniam, zgr B. Akan, "ESRT: Event to sink reliable transport in wireless sensor networks," pp. 177–188, 2003.

[38] Damla Turgut Mohammad Z. Ahmad, "Congestion avoidance and fairness in wireless sensor networks," in *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, 2008, pp. 1 – 6.

[39] Majid I. Khan, Wilfried N. Gansterer, and Gnter Haring, "Congestion avoidance and energy efficient routing protocol for wireless sensor networks with a mobile sink," .

[40] Jerry Zhao and Ramesh Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in *ACM Conference on Embedded Networked Sensor Systems*, 2003, pp. 1 – 13.

[41] Sandeep Sirpatil, "Implementation of IEEE 802.15.4 protocol stack for Linux," 2006.