Data Transfer Methods Between Wireless Sensor Networks and Mobile Sinks

Jorick van der Hoeven

A project presented for the degree of

Masters in Computer Science

Computing Laboratory
University of Oxford
England

May 2010

Data Transfer Methods Between Wireless Sensor Networks and Mobile Sinks

Jorick van der Hoeven

Submitted for the degree of Masters in Computer Science
May 2010

Abstract

This project investigates the use of mobile sinks in wireless sensor networks. It focuses on three different transfer methods - Single Acknowledgement, Double Buffered and Sliding Window. The Single Acknowledgement transfer method is a naive method which acknowledges every packet it receives. The Double Buffered method keeps the same transfer protocol but reduces transfer times by performing time intensive data preparation in advance. The Sliding Window method is a variation of the TCP cumulative acknowledgement sliding window which only uses one timer. The project also proposes a way to select with which node a mobile sink is likely to have the longest connectivity based on several RSSI values taken over a period of time. All of the methods presented in this project are designed to be integrated into the WildSensing project which focuses on monitoring badgers in Wytham Woods, Oxford, UK.

Keywords: Wireless Sensor Network, uIP, Mobile Sink, Transmission Quality, Node Selection.

Acknowledgements

I would first of all like to thank the WildSensing research group for allowing me to work with them on this project. More specificaly I would like to thank Niki Trigoni, my supervisor, for spending the time to advise me and help me see the forest despite all the trees. I would also like to thank Ricklef Wohlers and Andrew Markham for helping me understand the workings of the Contiki operating systems and the WildSensing system they were building. Finally I would like to thank Richard Masters for helping me stay focused and telling me when I stopped writing in English.

Contents

Abstract **ii**

Acknowledgements **iii**

Introduction **1**

1 Background **3**

The WildSensing Wireless Sensor Network 3

Related Work 5

2 System Description **8**

Sink and Responder Behaviours 8

Hardware 9

Software 10

Operating System 10

The Network Stack 12

The Development Platform 15

3 Outline of Implemented Data Transfer Methods **16**

Single Acknowledgement Transfer 16

Double Buffered Transfer 17

Sliding Window Transfer 18

4 Investigation of Transfer Method Throughput **24**

Comparing the Different Transfer Methods 24

Test Results 27

5 Selecting Nodes using RSSI **34**

Motivation 34

Measurement Method 35

Selection Method 36

6 Conclusions **40**

Summary 40

Learning Outcomes 41

Future Work 42

Bibliography **43**

Appendix A: System Flow Charts **48**

Appendix B: Python Scripts **54**

List of Figures **65**

Introduction

Current approaches to gathering data for the purposes of wildlife monitoring and conservation are labour intensive, forcing researchers to collect data from each sensor placed in an area of interest. The WildSensing project of Oxford and Cambridge Universities aims to reduce the labour intensiveness of this task by creating a wireless sensor network which will be able to gather data about how the micro-climatic changes affect the movements of badgers and aggregate all of the relevant data on a few storage nodes in the network which will be able to transfer data to mobile devices carried by researchers in the area.

This project will integrate into the WildSensing project by building a basic system - called a mobile sink - which will be able to wirelessly gather data from storage nodes in the wireless sensor network. There are three main areas of development in such systems: routing algorithms, selection of storage nodes and transfer methods between storage nodes and mobile sinks. The aim of this project is to develop a fast, reliable transfer method for use in such a system whilst building a platform for future development on the former two areas. The project will explore three different transfer methods which will be referred to as Single Acknowledgement, Double Buffering and Sliding Window.

The transfer methods put forward by this project are designed to gather the data reliably from the storage node over what could potentially be a very lossy connection whilst at the same time having the smallest possible memory footprint on the embedded systems being used. The methods are also compared and contrasted in this project to determine which ones fare better in real world environments and why. The metric chosen to measure the implemented methods is throughput[1]. This

[1]The rate of data being transferred in kB/s

is because high throughput is an absolute priority in such data gathering systems as one cannot be sure how much time is available for the data gathering process. After investigating the three implemented transfer methods this project will also briefly touch upon storage node selection mechanisms and propose a system to select storage nodes to maximise the efficiency of gathering data from multiple storage nodes.

Before delving into the minutiae of the systems used and how the methods work Chapter 1 gives some background information about wireless sensor networks and the WildSensing project. Chapter 2 presents the framework created to test the transfer methods and the software and hardware used in this project. Chapter 3 details the way the transfer methods work. Chapter 4 builds on this to present how they were tested and provides a performance analysis of the different methods. Chapter 5 briefly discusses how to select nodes using the Received Signal Strength Indicator (RSSI) values measured from the beacon responses. The report is concluded in Chapter 6, which presents the final results of the project and discusses possible avenues for future investigation.

Chapter 1

Background

This chapter introduces the wireless sensor network used in the WildSensing project and this project. It also discusses some of the research that has been performed related to this project.

The WildSensing Wireless Sensor Network

The WildSensing group is a collaborative effort between the Oxford and Cambridge University Computing Laboratories and the Wildlife Conservation Research Unit of the University of Oxford to reduce the labour intensiveness of wildlife monitoring by automating most of the process using a wireless sensor network. In order to explain what wireless sensor networks are and why they are useful this project will discuss the problems faced by wildlife conservationists and the solutions provided by the wireless sensor network proposed by the WildSensing group (Figure 1.1).

In the presented situation, zoologists want to observe the living habits of badgers in Wytham woods and the effect that climatic changes have on them. Traditional observation methods involve placing temperature and humidity sensors in the area to be observed and checking them periodically as well as observing the areas where the badgers live in order to document their movements.

To reduce the observation time the badgers can be fitted with small radios - called Radio Frequency ID tags or RFID tags - which can be sensed by RFID sensors. This will observe the badger activity for the zoologist who will only need to visit the area

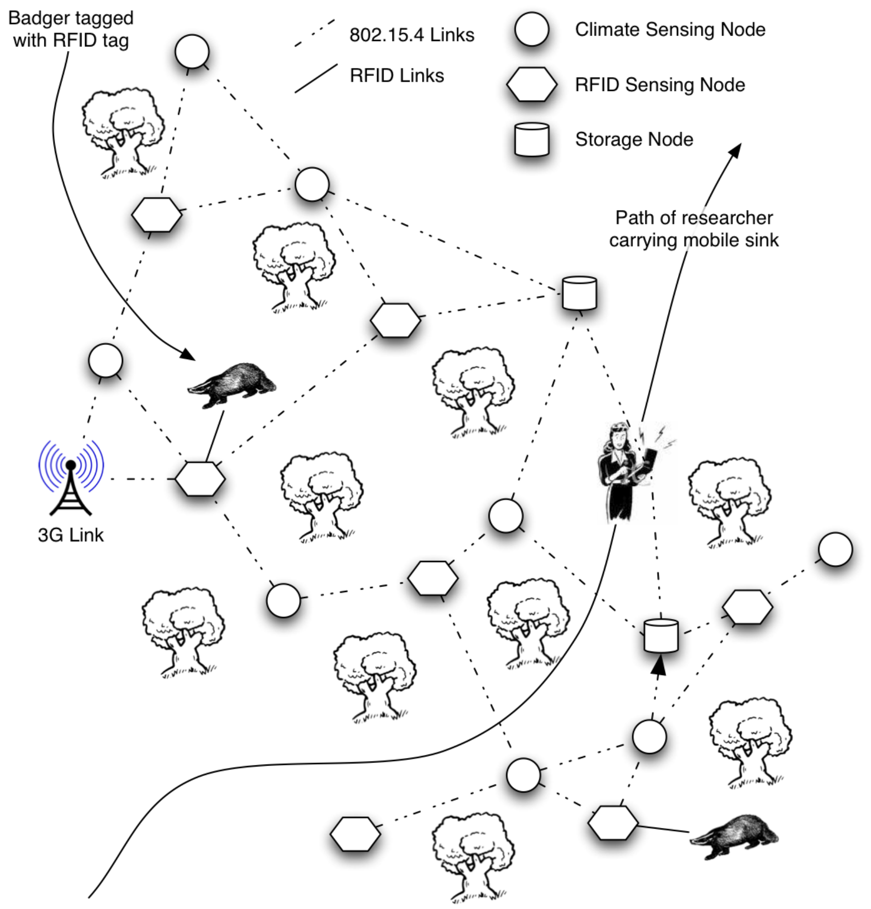


Figure 1.1: A diagram of the WildSensing Wireless Sensor Network

to collect data from the RFID sensors while measuring the climatic changes.

This solution, however, cannot be used for time-critical data about the movements of badgers. If a badger from another group enters the observed area the zoologist will want to be around to see how the badgers behave and thus collecting data about the whereabouts of badgers a day after the fact is too late to be useful.

In order to solve this problem, the RFID sensor nodes can be connected together in a wireless network (represented by 802.15.4 links in Figure 1.1) containing a 3G Link which will take time-critical data from the network of RFID sensors and send it over the cellular network and the internet to the zoologists so that they are notified quickly after a time-critical event happens.

In order to complete the solution, the network of sensors must also include sensors to register climatic changes. These sensors however do not need to send data to the zoologist immediately so it is perfectly fine if the data is collected later. Now that all of the sensors are connected in a wireless network it seems silly for the zoologist to need to visit each climate sensor to download data, the data should be moved to places where the zoologist will be. To achieve this, the WildSensing group added storage nodes to their solution. These storage nodes are sensors in the wireless sensor network which are located in places where the zoologist is likely to be and will therefore gather data from the other sensors in the network so that the zoologist need not visit them.

The final part of the solution - and the objective of this project - is to develop a special node, called a mobile sink, for the wireless network which the zoologist can carry when performing conservation related tasks and will download all of the data from the wireless sensor network so that the zoologist need not interact with the sensors.

Related Work

In the seminal paper by Gupta et al on 'The Capacity of Wireless Networks' [22] one can clearly see the need for mobile sinks in wireless sensor networks. They propose, and prove, that in ad-hoc wireless networks the throughput of the network is inversly proportional to the number of nodes in the network. To address this issue they propose introducing cells in wireless networks which can act as points to collect data from the network and use another transport medium to pass the information on. This is where the mobile sink and storage nodes come in. If all of the information from the WildSensing network were routed through the 3G Link node the throughput of the network would be severely affected. The use of storage nodes allows the sensor network to have a cellular structure where the link between the cells delivering data to the researchers becomes the mobile sink which can travel quickly between the cells gathering information and reducing network congestion.

In order to get data from the sensor network to the mobile sink several different methods have been proposed. In particular Sankarasubramaniam et al [37] propose a new transport protocol[1] which purports to bring information reliably from an

[1]Instead of using the standard internet transport protocols TCP and UDP

event to a sink by changing the states of the nodes in the network based on their connectivity status. This method, however, is of little use in this project as the transfer methods in this project are built for the embedded systems operating system Contiki running the uIP stack[2]. The uIP network stack is a IPv6 networking stack specially designed to run on resource constrained devices. In the paper presenting uIP [15] Dunkels explains that it is beneficial to use IPv6 given that many devices can communicate via IPv6 and therefore one does not need a special node to translate IP communication into the specific communication protocol used by the network. Overhead issues presented by the TCP/IP protocols are also discussed and it is shown that UDP can be used when minimal overhead is required and otherwise the TCP protocol can be compressed to a certain degree to reduce the impact of the overhead.

This project aims to use the code provided by the WildSensing project without modifications. This entails two things, the transport protocol used will have to be the UDP protocol and any modifications to the transfer protocol will need to be performed within the application instead of in the networking stack as is usually the case. In order to make the UDP connections provided by the network reliable. Inspiration was taken from previously proposed RUDP protocol [4] which adds reliable connectivity to UDP transfers.

In order to properly gauge the results of this project it is useful to know what throughputs are physically possible before different transport protocols are involved. Osterlind et al have studied the maximum possible throughput using the uIP stack in [34] and shown that when using the uIP stack in conjunction with the Contiki OS the maximum possible throughput on a cc2420 radio with a theoretical throughput of 256 kBits/s (32 kB/s) is 120 kBits/s (15 kB/s). This significant difference in the theoretical and the useful throughputs is due to the fact that when sending data it needs to be moved from the memory to the radio which takes quite a long time. It has also been stated in [34] that the processing of headers by the different layers reduces the throughput even further.

Using RSSI values to a node in relation to other nodes in a wireless network is

[2]These are explained in more detail in the following chapter

quite common and has been covered by Sugano et al [40]. Kotz et al [28] however show that individual RSSI values will lead to erroneous assumptions about the network given that the connection fall off is hardly smooth. As far as can be determined no work has been produced which uses RSSI values at different distances in order to ascertain the projected communication duration and strength between a mobile and static node.

Chapter 2

System Description

This project was built in two parts. Responder nodes which are nodes running a module running in parallel to the software written by the WildSensing project allowing the nodes of the sensor network to respond to mobile sinks and a mobile sink node which interacts with responder nodes and downloads data from them. This chapter gives a basic explanation of the behaviour of mobile sink and responder nodes followed by an outline of the hardware used as well as a description of the embedded systems operating system and network stack used. The chapter is concluded by a presentation of the tool chain used to program the nodes and the virtual testing environment used to test the nodes.

Sink and Responder Behaviours

Figures 2.1 and 2.2 give a general overview of the behaviours of both sink and responder nodes. The mobile sink described in Figure 2.1 will continuously beacon its presence as it travels through the network. When the sink hears responses from responder nodes it will select the most desirable node[1] and send a request for data to it to initiate the transfer. Once a transfer is complete the sink node will mark the IP address of the node it has just transferred data from and ignore any future messages from that node. If the transfer process is interrupted, because the responder node is not replying anymore, the sink node will reset its connection and begin beaconing

[1]Currently this is implemented on a first come first served basis. Chapter 5 introduces a more advanced system.

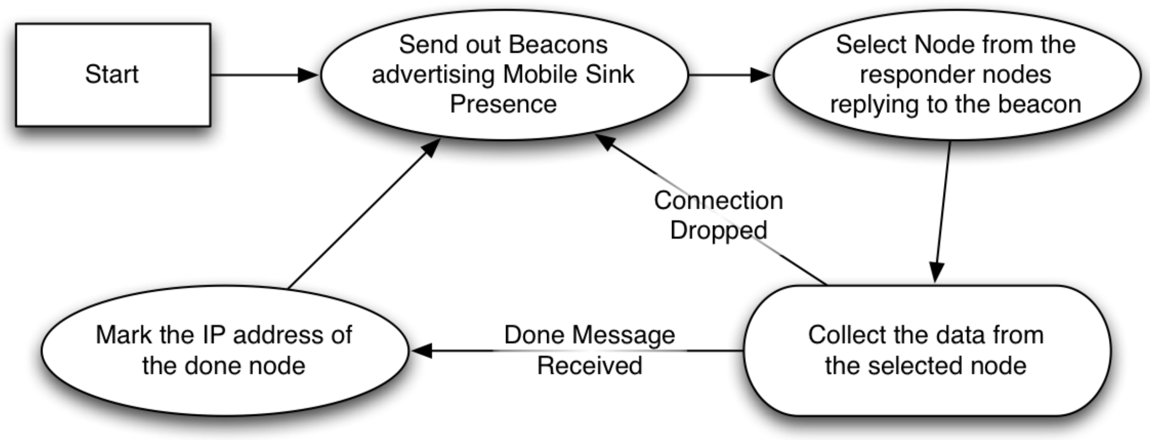

Start
Send out Beacons advertising Mobile Sink Presence
Select Node from the responder nodes replying to the beacon
Connection Dropped
Mark the IP address of the done node
Done Message Received
Collect the data from the selected node

Figure 2.1: The general behaviour of the Mobile Sink

again. The beaconing behaviour of the sink node is possible because, in contrast to the responder nodes, the sink node will be regularly recharged and benefits from the large amount of power provided by the mobile computing device it is attached to.

The responder node in Figure 2.2 replies to beacons it receives from the mobile sink, advertising its presence and willingness to send data. Once it receives a command from the mobile sink node to send data the responder node transfers the data to the mobile sink making sure that every packet it sends has arrived reliably. When the responder node has no more data it will send an end of connection message to the sink and stop listening for beacons from potential mobile sinks for 30 minutes[2]. This 'sleeping' state of the node allows the node to save power by avoiding needless responses to beacons as well as reducing the amount of response packets which could potentially collide after the sink sends out its beacon. If the connection is interrupted during the transfer, the responder will go back to listening for potential mobile sinks so that it can transfer the rest of the data.

Hardware

Both of the modules implemented in this project are built to run on the Tmote Sky embedded systems used by the WildSensing project. The Tmote Sky systems contain an 8Mhz MSP430 CPU with 10kB of RAM, 48kB ROM and 1MB of flash

[2]This value is a network specific value based on the period with which data is propagated through the WildSensing network.

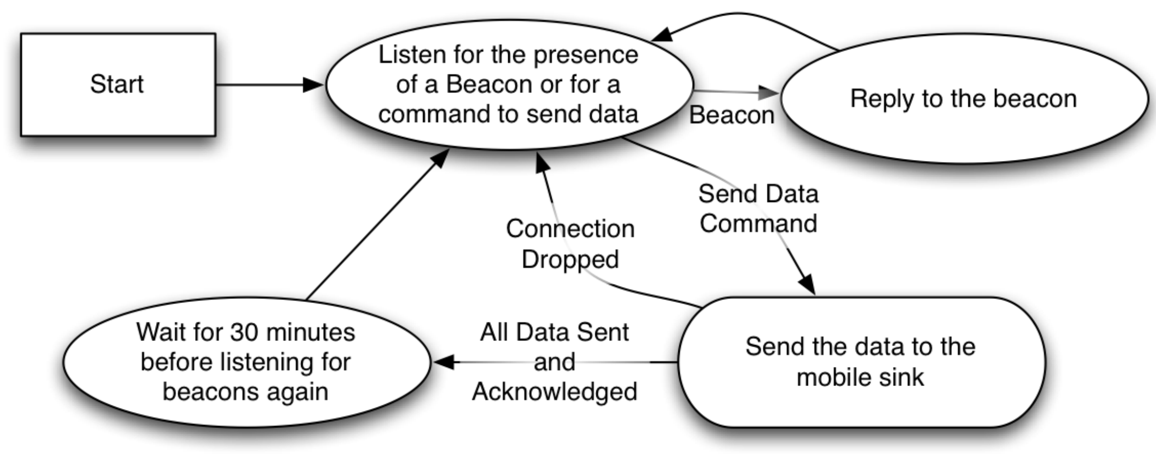

Start
Listen for the presence of a Beacon or for a command to send data
Beacon
Reply to the beacon
Send Data Command
Connection Dropped
Send the data to the mobile sink
All Data Sent and Acknowledged
Wait for 30 minutes before listening for beacons again

Figure 2.2: The general behaviour of a responder node

storage for data. This means that the modules written had to be extremely small and use as little RAM as possible to fit onto the embedded system along with the operating system and the code from the WildSensing project[3].

The radios used by the Tmote Sky nodes are CC2420 radios which use the 2.4GHz band transmitting on multiple channels and comply with the IEEE 802.15.4 standard for wireless sensor network transmissions. The range of the radios varies between nodes given that some are cheaper and contain less powerful antennae and thus have a range of 70m [30]; the more expensive nodes have a reported range of 125m [36]. This difference in ranges could have caused trouble when testing the transfer methods implemented by this project and so it was decided to perform all of the testing on the cheaper systems.

Software

Operating System

Due to the restricted amount of space available on the hardware components a special purpose lightweight operating system needed to be used. There are several such operating systems available most notably TinyOS [24] and Contiki OS [14]. In order to integrate with the WildSensing project the modules were built to run

[3]Dealing with program size and overfilled RAM was one of the most frustrating aspects of this project.

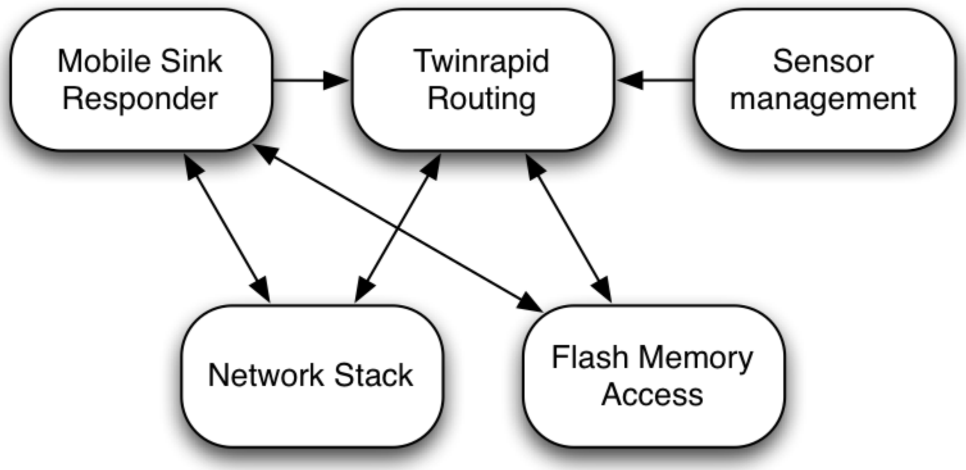


Figure 2.3: The Responder Module in the WildSensing System

on a customised version of the Contiki OS specially developed to cater for the requirements of the WildSensing project.

The need for a customised version of the Contiki operating system in the WildSensing project arose due to the limited space available for adding extra functionality in the few kilobytes left on the Tmote Sky nodes after a default installation of the Contiki operating system. Therefore, all unnecessary features were removed from the operating system in order to provide space in the ROM memory for the different modules of the WildSensing project. The two main elements removed from the system by the WildSensing project were the Coffee file system manager and a networking stack which would not be used by the WildSensing system called Rime.

The Coffee file system was removed because it takes up a lot of space to perform file system maintenance which could be better used by custom code to perform functions pertaining directly to the WildSensing project. To be able to store information on the system instead of using the Coffee file system the WildSensing project devised a light weight interface to the flash memory which stores relevant data whilst having minimal overhead.

The removal of the Rime networking stack is due to the fact that it operates in parallel to the uIP networking stack (explained later in this chapter) which the project had chosen to use. The entire stack could not be completely removed however, as the buffer management system to manage incoming and outgoing data transfers was shared between uIP and Rime as well as the system for managing IP addresses.

The mobile sink part of the project was programmed as an application running on the same customised version of Contiki OS used by the WildSensing project. It uses two connections, one which broadcasts messages out to the nodes surrounding it advertising its presence and another, unicast connection, to perform the actual data transfer. The sink, upon receiving data will send it via the USB port to be captured by a python script. It was decided to create the sink in this way rather than setting up a node as an IP bridge and controlling the data via an application written on the host computer for cross-platform compatibility. The serial reader script works on all of the systems in the WildSensing group, an application written on Mac OS X or in Linux would need to be ported to be able to run on other systems. Also, at the time of writing, setting up the necessary network interfaces to allow a node to act as an IP bridge was not possible on Mac OS X 10.6, the operating system of the main development machine.

The responder part of the project was also programmed as an application running on the WildSensing customised version of Contiki OS (Figure 2.3). It however, needed to run alongside the WildSensing routing code (called twinrapid) and the sensor management code meaning that it required a much smaller memory footprint to fit into the small amount of space left by the operating system and the two WildSensing modules. To ensure that the memory footprint remained as small as possible, the responder module only used 1 network connection and 1 timer[4].

The Network Stack

The networking stack used by this project can be seen in Figure 2.4. There was not much choice in network transmission protocols due to the fact that the mobile sink system needed to integrate with the WildSensing sensor network which uses uIPv6. The whole stack allows for IPv6 communication using the UDP protocol. The code written for this project all lies in the application layer and uses the pre-existing

[4]Many systems use two network conections, one for link maintenance packets such as acknowledgements. Many protocols also use more than one timer such as the cumulative acknowledgement sliding window method from TCP.

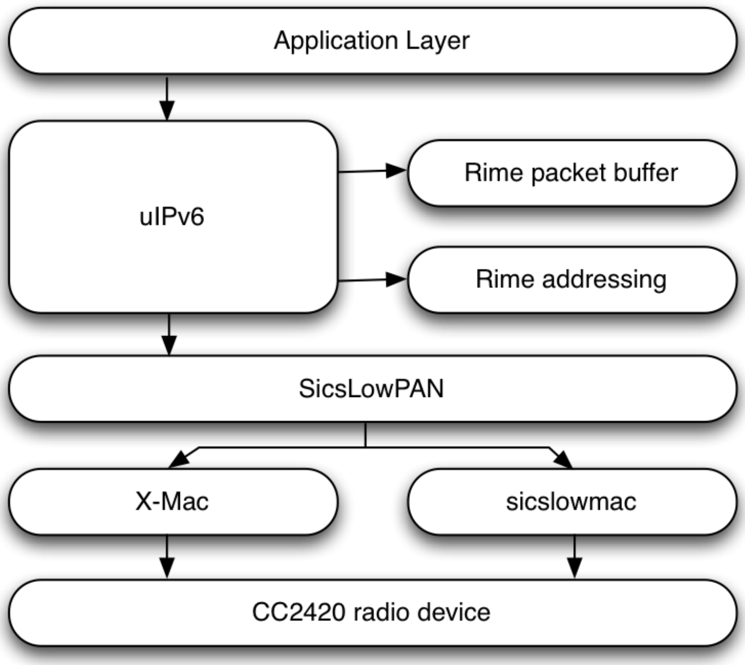


Figure 2.4: The Networking stack used by the project

lower layers.

The uIPv6 stack is a fully functional IPv6 stack designed for embedded systems allowing them to communicate with larger systems that use IPv6 and it only uses 4kB of ROM and 1kB of RAM with all functions. The stack has been modified to remove the ability to use TCP communication in order to allow more space for code pertaining directly to the WildSensing project[5]. uIP handles communication with the application layer and prepares the data packets which need to be sent. It does this by generating the UDP headers necessary for the transfer and other functions specific to UDP transfers. At the moment however, the uIP implementation does not allow for UDP checksums so the data being transferred by the systems in this project will not be checksummed. This is not considered too much of a problem because when a packet is corrupted in a wireless transfer it is exceedingly rare that only the payload is corrupted. This is especially true when the payloads being sent are very small. The systems proposed by this project sends four 48 Byte WildSensing data elements along with a 1 Byte identifier in each packet. The final packet sent over

[5]Leaving the TCP protocol in the stack would have taken up too much space in the ROM memory, also the overhead generated by the TCP headers would have impacted the throughput heavily.

the radio is 308 Bytes meaning that there are 115 Bytes[6] of header information in each packet being sent and thus lowering the likelihood that corruption only affects the payload.

The uIP layer uses the Rime addressing system to manage the allocation of IP addresses and also uses the global 128 Byte buffer provided by Rime. The discrepancy between the packet size and the buffer size can be explained by the fact that the uIP stack only uses the buffer to create the packet header, the rest of the information remains where it has been stored in RAM by the application.

After uIP creates the UDP packets the SicsLowPan layer takes over. The SicsLowPan layer is an implementation of RFC4944 [12] which defines the use of IPv6 over 802.15.4 networks. The important parts of SicsLowPan's operation which concern the project are the header compression and the fragmentation functions. SicsLowPan has the ability to compress UDP headers which means that IPv6 packets have much less overhead. If after compressing the headers the packet defined by the uIP is still too big to fit into an 802.15.4 frame whose payload size is set to 102 Bytes the SicLowPan layer proceeds to fragment the packet. This functionality is present because the maximum packet size in IPv6 is 1280 Bytes. Once the packets have been properly fragmented and compressed the data is passed onto the MAC layer.

Figure 2.4 shows two distinct MAC layers which can be called by SicsLowPan. This is because the WildSensing project uses the X-MAC system [5], a duty-cycled MAC layer which attempts to keep the radio off for as long as possible to save power. The problem with this MAC layer however, is that duty-cycling and the necessity to strobe to wake up a counterpart node has an enormous impact on throughput. Given the need to transfer data from the nodes as quickly as possible it was decided to use the Sicslowmac MAC layer [13] which keeps the radio on all the time and thus achieves a higher throughput.

Regardless of the MAC layer used, the MAC layer frames the fragmented data received from SicsLowPan into 802.15.4 frames and passes the data to the cc2420

[6]Values quoted are including header compression and fragmentation, in this case the initial packet was fragmented into three 802.15.4 frames adding up to 308 Bytes. This also means that 37% of data transferred is overhead data.

radio driver which sends the data wirelessly.

The Development Platform

In order to be able to program the embedded systems appropriately the systems were connected via USB to a computer running Mac OS X 10.6. A serial reader python script written by Andrew Markham was used to read the output from the embedded systems and save it to file. Programming the Tmote Sky nodes was done using the mspgcc compiler and several loading scripts.

Late on in the project, after many hours spent tracing the systems using strategically placed printf statements, a simulator named Cooja was added to the development platform. Cooja can simulate multiple nodes running at the same time in a virtual environment as well as simulating their radio communications. The simulator environment also allows the simulation to be paused meaning that a particular node's exact state can be minutely examined. This greatly increased the speed of the development process in the later stages of the project.

In order to correctly analyse all of the data gathered by Cooja and the serial reader script several python scripts were written to parse the gathered data and plot it on the graphs (which can be seen throughout this report) using the matplotlib library.

Chapter 3

Outline of Implemented Data Transfer Methods

This chapter discusses the main transfer protocols that were implemented to transfer data reliably from the responder node to the sink using unicast UDP connections. Three different methods were created to generate the highest throughputs while ensuring the reliable transfer of data. For the interested reader, detailed flow charts of the behaviours of each of the created modules can be found in Appendix A.

Single Acknowledgement Transfer

The first method implemented to transfer the data from the responders to the mobile sink is a method which acknowledges every single packet received. It is a simple method designed to be a robust platform for future development.

The method follows a very linear execution path allowing each event to happen in turn. Data is retrieved from flash, data is sent to the sink, data is printed to the terminal at the sink and an acknowledgement is sent to the responder. This means that there are periods where either the sink or the responder are simply waiting to hear back as their counterpart performs what could be quite a lengthy task which has a strong impact on the throughput (Figure 3.1). The advantage of this method is that it is easily traceable and is simple to implement.

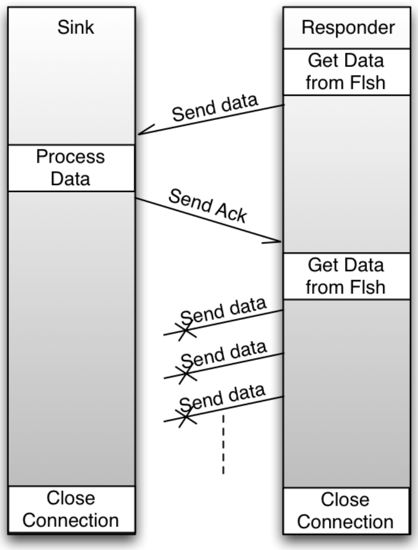


Figure 3.1: An example of the Single Acknowledgement transfer method with a connection fail after the first successfully sent packet.

To increase the throughput, the packet fracturing property of the IPv6 protocol employed in uIP was used. This allowed the responder to create a network packet of 192 Bytes[1] containing four 48 Byte data packets from the WildSensing network to be sent to the mobile sink as one fractured packet. This means that for every packet that the responder sends to the sink it is effectively sending four WildSensing data packets meaning that in effect only one acknowledgement needs to be sent for every four data packets. This increases the throughput fourfold without actually adding any complexity to the responder or the sink.

Interruptions to the connection between the responder and the sink are handled by a timeout function in the responder. The responder resends the network packets at most 10 times if it does not hear from the sink. After which it pushes the data retrieved from flash and stored in RAM back into the flash memory. If the sink does not hear from the responder for a predetermined amount of time it will close the connection and resume beaconing in an attempt to connect with another responder.

Double Buffered Transfer

As can be seen from Figure 3.1 the Single Acknowledgement system spends a lot of time processing the received data in the mobile sink and fetching data from the flash memory in the responder. In an effort to increase the throughput an extra buffer was added to the responder so that it could not only store the packet that had just been sent but also store the next packet to be sent allowing the responder to load the data while it is waiting for an acknowledgment from the sink. An example of

[1]Without headers, with UDP headers and 802.15.4 headers the final packet sent totalled 308 Bytes.

this behaviour can be seen in Figure 3.2.

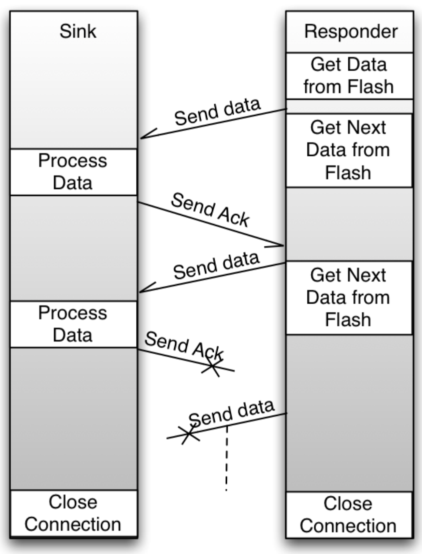


Figure 3.2: An example of the behaviour of the Double Buffered transfer method with connection interruption at the second acknowledgement.

The addition of a second buffer meant that the handling of the final acknowledgement of data needs to be handled differently in this transfer system compared to the previous transfer system. In the Single Acknowledgement method the final done message is sent as soon as the flash memory is empty given that every piece of data preceding a call to load data from an empty flash memory would have been acknowledged. In the Double Buffered system however, when the responder fetches the next data to send and finds the buffer empty it cannot immediately send a done message because the final packet has not been acknowledged. To overcome this, a flag is set in the system which is checked at the reception of every acknowledgement. If the flag is true, the system knows that all of the data has been sent and acknowledged so the done message can be sent. Also, to avoid the storage of random data, if the done flag is true when the connection is interrupted the responder will only push the buffer corresponding to the sent data back to the flash memory and not the buffer corresponding to the data to be sent.

Sliding Window Transfer

The Sliding Window transfer system is the most intricate transfer system implemented in this project. This system is derived from the Sliding Window system implemented in the TCP protocol and reduces the number of packets needed to acknowledge data being transferred. The Sliding Window system in TCP however, uses several timers to determine whether or not a sent packet has timed out. The version implemented in the responder uses only one timer to reduce the amount of memory needed to run it.

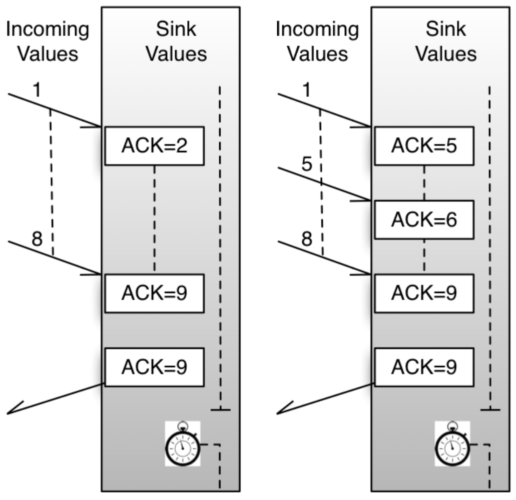
The Sliding Window transfer system works by sending a burst of packets from the responder without waiting for an acknowledgement between the packets. The sink receives these packets and logs each one as received. When the burst is over, the sink sends back an acknowledgement to the responder telling it which packet it expects to see next. The responder uses this information to determine which packets made it to the mobile sink and sends the next burst of packets accordingly. To reduce the number of packets sent, if a burst is interrupted in the middle of the transfer, the responder will only send unacknowledged packets again. To handle such cases appropriately the following scenarios are taken into account:

Sink Scenarios:

Full Burst Received:

In this situation the local sequence number is incremented every time a valid data packet is received. A packet is determined as valid if the sequence number of the data packet matches the local sequence number. When the sink has counted that the correct number of packets have arrived - the sink should know the burst size of the responder ahead of time - it sends an acknowledgement to the responder with the sequence number of the next data packet it expects to receive.

If the sink is expecting data with a sequence number higher than the data arriving from the responder - due to the responder not receiving the previous acknowledgement. The sink will ignore incom-



(a) (b)

Figure 3.3: The behaviour of the sink when a full burst is received. (a) is when all the sequence numbers match and (b) is when the local sequence number is higher than the arriving sequence numbers.

ing packets until one of them has the expected sequence number (Figure 3.3b).

Partial Burst Received:

In this case the sink has received part of a burst of data but some or all of the sequence numbers are wrong. For instance if the sink is expecting packets with sequence numbers 1 - 8 from the responder, and only 2-8 arrive or packet 3 fails to arrive. In such instances the sink will send back an acknowledgement with the sequence number of the failed packet to arrive and discard all packets which have arrived after the missing packet in order to keep the sequence of the data identical to the sequence of the data sent by the responder. If the end of the burst fails to arrive the timer which times the arrival of the burst will run out and the sink will send an acknowledgement for the next packet it is expecting (Figure 3.4). If the timer is triggered a certain number of times, the connection is deemed to have failed and the sink will close the connection with the responder and begin polling for data again. If data packets arrive with a sequence number lower than the local sequence number - again due to the failure of a previous acknowledgement packet to arrive at the responder - these will be ignored until a packet with a valid sequence number arrives.

Responder Scenarios:

The responder, in order to handle the burst transfers and not lose any data, keeps an array of the packets which will need to be sent in a particular burst. The array has the particularity that its start position is dynamically managed by the transfer system to allow the responder to know which packets from the burst have reliably made it through to the sink and thus do not need to be sent again. This is done by reassigning the start position to the array to point to the packet with the same sequence number as the acknowledgement received by the responder in the case of a partial transfer.

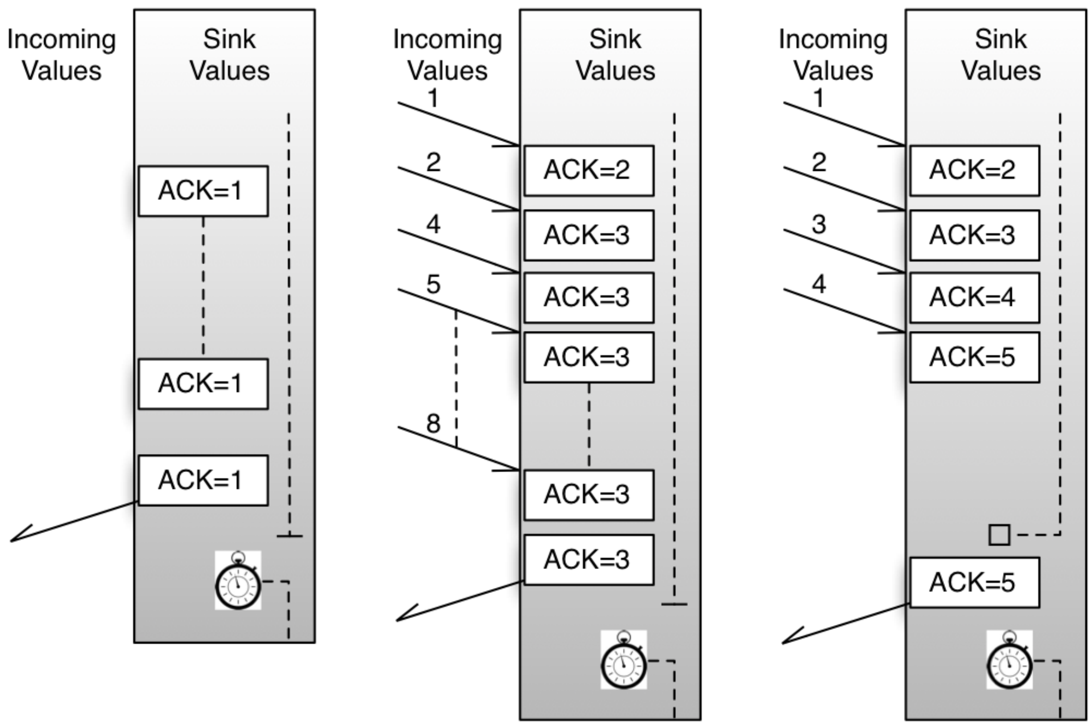


Figure 3.4: The behaviour of the sink when receiving a partial burst. Left is the situation where no data arrives, middle is an example when a packet from the burst is missing and right is the situation when the end of the burst fails to arrive.

Timer Timeout:

In this case the timer which determines the amount of time to wait for an acknowledgement has run out. This means that the previous burst transfer has not been acknowledged and therefore will need to be sent again. The responder will then send out the same burst as it has just sent, increment a resend counter and restart the acknowledgement timer. It will only do this until the maximum number of resends has been reached. If that happens the responder will assume that the transfer has failed, close the connection and push the data from the buffer back into the flash memory - preparing itself for the next mobile sink.

Partial Acknowledgement Received:

A partial acknowledgement is an acknowledgement message which is received by the sink with a sequence number corresponding to the packets stored in the burst buffer. This allows the responder to infer that the burst was not fully received. The responder simply sets the start of the burst array to the packet with the sequence

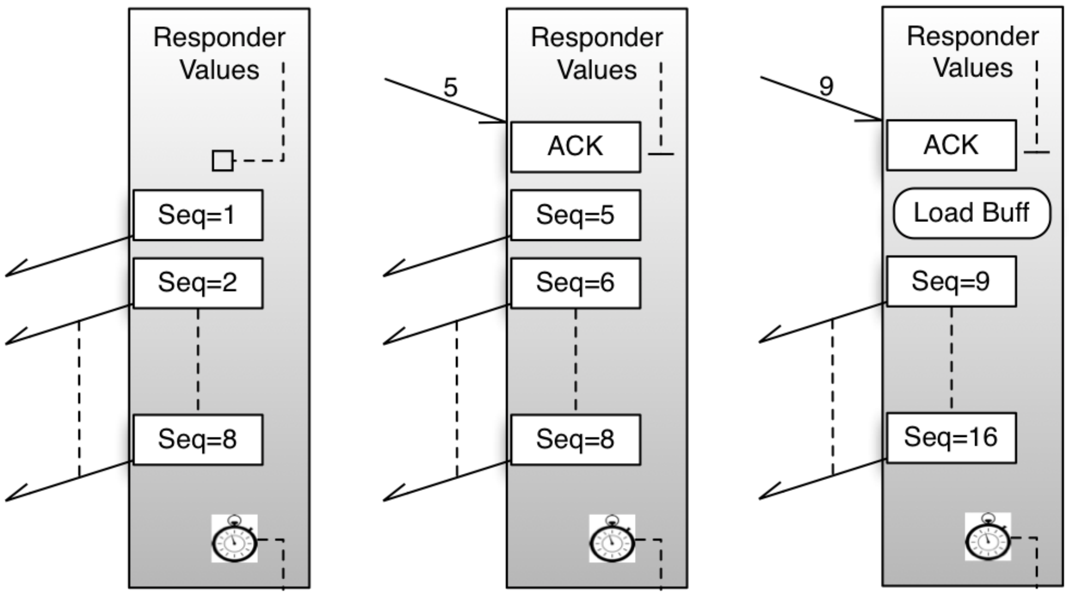


Figure 3.5: The behaviour of the responder in the different acknowledgement situations. Left: Timeout. Middle: Partial Acknowledgement. Right: Full Acknowledgement.

number corresponding to the sequence number in the acknowledgement and sends the unacknowledged part of the buffer again.

Full Acknowledgement Received:

When a full acknowledgement is received the sequence number of the acknowledgment corresponds to the sequence number of the first item of the next burst to be sent. This simply means that the burst array is filled with new data, the start of the array is reset, the next full burst is sent and the acknowledgement timer is restarted.

Special Cases:

There are several special cases which need to be directly addressed by the system. The three most important ones are when an acknowledgement with completely the wrong sequence number arrives, when there is no more data and how to handle the sequence number variable overflowing.

The first special case is quite trivially handled. If an acknowledgement or a data packet arrives with a sequence number that is not in the range of the current sending burst it is completely ignored and considered as a corrupted packet. This will only happen if there is a serious problem with the nodes or if the packet is indeed corrupt so it is safe to ignore such packets. The only thing that can happen is that either the sink or the responder closes the connection and no data will be lost.

Handling a situation when there is no more data is absolutely crucial for the responder because the aim of the system is to reach a situation where the responder has no more data to transfer. There are two possible cases in this situation, either the system remarks that there is no more data when attempting to start creating another burst, or the system runs out of data while creating another burst. In the latter situation, the remaining packets of the burst are filled with null data which will be ignored at the sink. In the former case, the responder knows that all the data sent has already been acknowledged so the responder simply sends a done message to the mobile sink.

The final case is an implementation issue rather than a design issue but is worth mentioning nonetheless. Because of the small amount of RAM available on the nodes, it was deemed prudent to try and minimise the amount of overhead required by the sending process. This meant that an 8 bit unsigned integer was used for the sequence numbers, which only has a range from 0-255. To be able to send more than 256 packets the variable was allowed to overflow and thus loop back around to 0 after 255. This means that it is not possible to simply check to see if sequence numbers are greater than or less than certain values when checking the validity of acknowledgments and data packets. Fortunately, as the number of packets in each burst is known it is possible to determine when the overflow issue will occur and handle it by checking for the sequence numbers up to 255 and then also checking the sequence numbers from 0.

Another implementation issue was the fact that the Sliding Window method could not be implemented using the eight packet bursts presented in the theoretical explanation but had to be limited to two packet bursts due to the fact that there was not enough space available in RAM to buffer more packets. Also, the method could only use one timer given that using extra timers took up too much space in RAM again.

Chapter 4

Investigation of Transfer Method Throughput

This chapter outlines the tests performed in this project to determine the throughputs of the different methods. The three transfer methods were compared in a virtual environment and then in real world environment to determine whether the results from the virtual environment were consistent with a physical environment.

Comparing the Different Transfer Methods

To effectively compare the three different transfer methods implemented in this project a series of tests were performed to measure the throughput[1] (the rate data being transferred in kB/s) of the transfer methods in various configurations. Once the throughput of the different transfer methods was measured, the reasons for the differences in throughput were explored.

In order to measure throughput, tests were performed in two separate phases, a phase of virtual testing loading the transfer methods onto virtual nodes placed in the Cooja virtual environment and a phase of physical testing where two embedded systems from the WildSensing project were programmed with the different transfer

[1]The term throughput is used in this project instead of the term goodput because this project does not alter the headers of the network packets in any way. It should be noted that the volume of useful data actually being transferred is only 62% of the throughput due to the overhead introduced by the IP and 802.15.4 headers.

methods. The methods were tested in these two phases because the data from the virtual environment, which provides a way to test the nodes without having to worry about interference, signal fading or signal scattering, can be used to determine whether the changes in throughput during the physical tests are due to issues with the transfer methods or are due to external influences. The virtual environment also allowed for a more comprehensive investigation of the state of the simulated nodes meaning that the differences between the transfer methods could be more accurately explained. To ensure that the real world tests were all the same the advice given about testing wireless communication in [28, 45, 46] was followed[2].

The nodes were programmed in such a way that the only difference between them was the transfer method. This ensured that any behaviour discrepancies between the tests can be attributed to the transfer systems used and not to differences in the software configuration. Every responder node was loaded with 7500 48 Byte WildSensing data elements which the sink would need to retrieve.

Virtual Tests

The virtual tests are designed to compare the transfer methods in an environment free of interference and environmental factors. The tests were performed by measuring the throughput while transferring the contents of the responder's flash memory to the terminal output of the sink node, the number of times packets needed to be resent and the number of times the connection was interrupted. All of the tests were performed in the Cooja virtual environment with the radio medium set to unit disk graph. A unit disk graph simulation of the radio medium assumes that the connection is perfect in the whole range of the radio and non-existent outside the radio range. This provides an ideal view of the transfer systems and any issues that may crop up can therefore be attributed to the transfer methods and not environmental factors. The simulated environment also allows for a more in depth analysis of the states of the nodes given that it is possible to see everything going on in the nodes rather than just the data printed out.

Given the fact that the simulation uses the unit disk graph method to simulate

[2]To see a full description of all the controlled variables see the Real World Static Tests section.

the radio medium the effect of distance on the transmission will be very unrealistic. Therefore, the methods are only tested on two different levels in the simulation system - both at the same distance. In the first test the sink prints out the data it has received, in the second the data is not processed at all but the data is simply passed into RAM memory. The latter is used to test how much printing affects the throughput.

To measure the differences between the transfer systems the following periods were recorded: the throughput of the transfer, the number of packets needed to be present and the number of times the connection was interrupted.

Real-World Static Tests

These tests add a layer of complexity to the testing environment by performing the tests on actual hardware. This means that environmental factors and hardware issues need to be taken into account to ensure that the testing of the different transfer systems isn't affected by factors such as system voltage, radio orientation or interference from other radios on the same band. In all of the physical tests performed, the following factors were kept constant: node orientation, nodes used, node elevation, node charge, the location of the test, weather conditions and interference from personal wireless devices.

- The orientation of the node was kept constant because the finding in [46] which states that the link quality and signal range are heavily affected by antenna direction, even for multi-directional antennae.

- All of the tests were performed on the same physical nodes because both [46] and [28] agree that one cannot assume that all wireless devices have the same range and afford the same link qualities.

- The elevation of the nodes was kept constant because it was noticed in preliminary testing phases and inferred from [45,46] that the scattering, reflective and fading effects of the ground could have a very large impact on the wireless transmission range and quality.

- To ensure that the transmission power did not vary between tests, the voltage and charge of the batteries in the nodes were kept constant by ensuring that the charge of the batteries remained constant for every test.
- Ensuring that the location of the testing remained unchanged throughout the tests is crucial because different locations will provide different interference patterns based on the physical objects around and the absorption qualities of the materials in the area. To have a minimum amount of interference to deal with, it was opted to perform the tests on an open field with a level ground.
- Given that it is impossible to control the weather conditions the tests were performed directly one after another on the same day to make sure that the weather conditions would remain as constant as possible throughout the tests.
- The final factor taken into account for the tests is present because the computers and personal telephones can have high power wireless transmitters which may start sending data in the 2.4 GHz band during the tests and thus swamping out the signals from the low power nodes being tested and interfering with the communications.

With the major environmental and hardware factors accounted for the tests were conducted with different distances between the nodes. The three distances at which the methods were tested were 10cm, 20m and 40m.

Test Results

Virtual Tests

Figure 4.1 represents the results of the tests in the Cooja simulation environment. As can be seen from Figure 4.1 the Double Buffered method is the fastest of the three methods tested. It can also be seen that the removal of the printing operation on reception of the data increased the throughput by up to 70%.

All of the methods when tested without printing the data suffered from a drop in connectivity in the first ten seconds (Figure 4.4) - the sliding window method

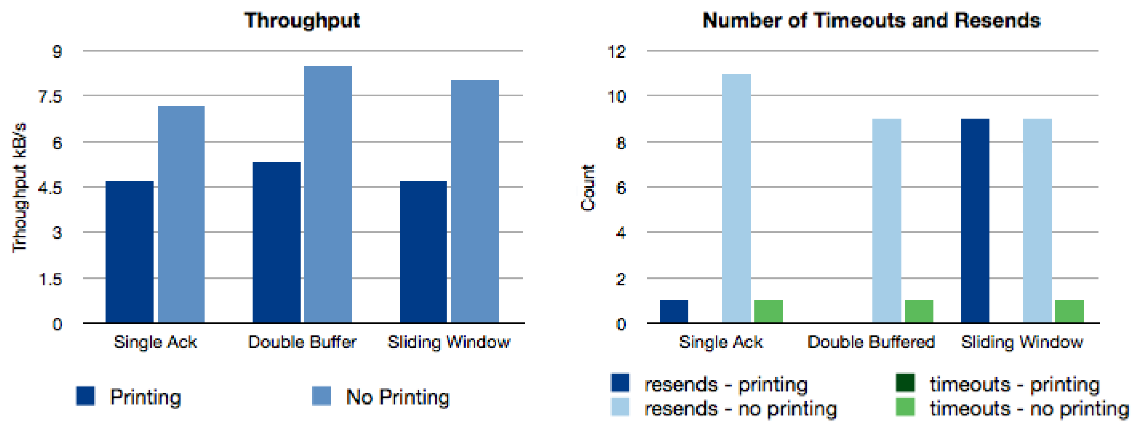


Figure 4.1: Throughput, resend and timeout counts in the virtual environment

with data printing almost dropped the connection but managed to recover on the final resend. This problem is due to an error in the neighbour discovery code of the uIP stack which tries to refresh its neighbour database during this time. In every case where the neighbour discovery caused dropped packets it is because the packet sent by the neighbour discovery protocol collided with one of the acknowledgment packets and caused the sink's uIP stack to stop alerting higher layers of incoming data from what it considered a stale address causing the connection failure.

Another feature that can be observed in Figure 4.4 is that at around second 90, there is a sharp dip in the data rate. This is due to the fact that after receiving an acknowledgment the responder waits for almost a second before replying because the operating system or the WildSensing code is performing a task that blocks the responder process.

Real World Static Tests

In the real world tests, in contrast to the virtual environment tests, the Sliding Window method manages to sustain a data rate that is consistently higher than the Double Buffered method (Figure 4.2). It can also be seen that distance did not have a significant impact on the throughput.

The features discussed in the virtual tests causing disruptions to the transfers can also be seen in the results of the real world tests in Figure 4.5. The neighbour discovery bug can still be seen disrupting the connection in the first 10 seconds. The

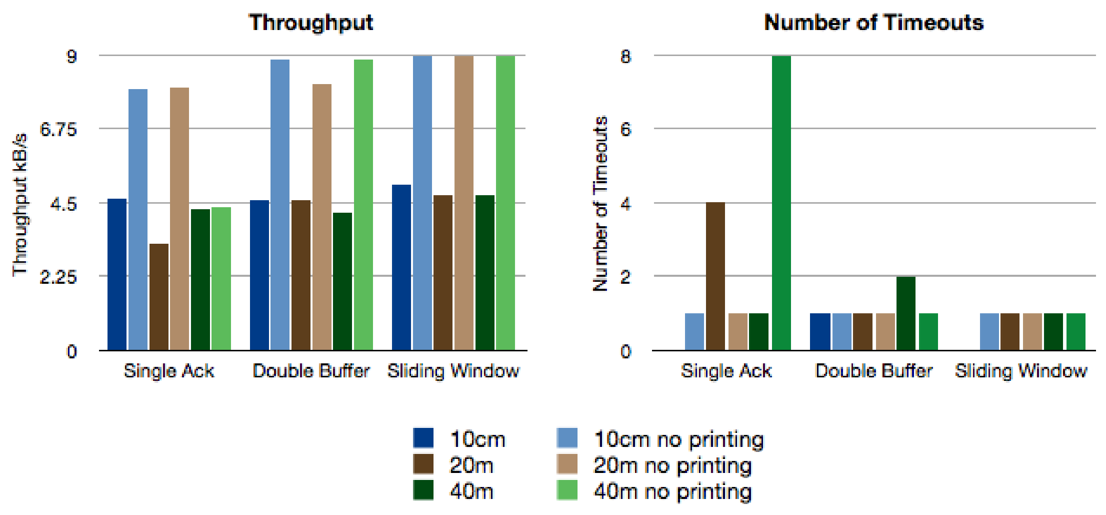


Figure 4.2: The throughput and the number of timeouts in the real world tests

seemingly haphazard occurrence of the drop is due to the fact that the bug relies on timing and may not occur if the neighbour discovery packets are sent while neither of the nodes are transmitting. The drop in data rate due to the responder process being suspended is also present in the real-world data tests but it is much more difficult to spot due to the fact that there are many other issues causing drops in the data rate. It can be seen occurring between the 85th and the 110th seconds on the graphs.

There were consistently more timeouts in the real world tests than the virtual tests due to the fact that radio waves were absorbed, reflected and diffracted by the environment increasing the possibility of collisions [45]. There are two problems with dropped connections. First the sink takes quite a long time to timeout - longer than the responder. Secondly it can take up to five seconds for the sink to reconnect with the responder after the connection is dropped. The first issue is easily solved by setting the timeout timer on the sink to something shorter. The second issue cannot be solved without making changes to the hardware specifications.

Test Conclusions

In order to properly assess the results of the tests a few extra tests were performed to examine the effects of getting data from flash (Fetch), sending acknowledgements (Ack) and printing the data on reception (Print). The impacts of these operations

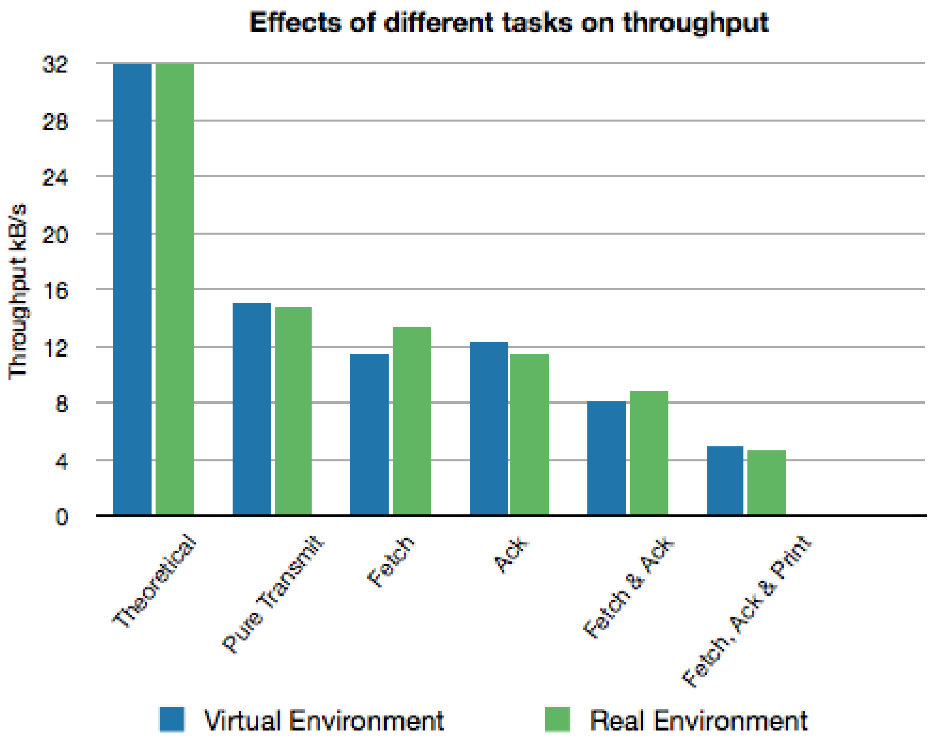


Figure 4.3: The effect of various different operations on the throughput of the nodes

can be seen in Figure 4.3. From this it can be seen that there is a discrepancy between the effects on the virtual nodes and the physical nodes which explains the differences in the transfer method test results.

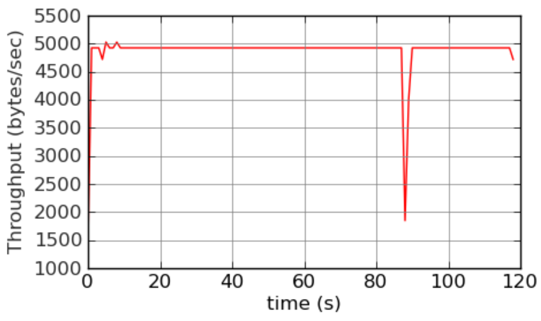
The virtual tests showed that the Double Buffered method had a higher throughput than the Sliding Window but this was not the case in the real world tests. As can be seen in Figure 4.3 this is because the virtual nodes take longer to retrieve data from the flash memory but can transmit acknowledgements faster. In the physical nodes, the inverse is true. This difference causes the Double Buffered method to perform much better in the virtual environment than in the real world environment.

As can be seen from Figure 4.3 it will be impossible for any of the methods to reach a throughput of higher than 15 kB/s without changing the network stack significantly[3]. A value of close to the 13 kB/s performed by simply fetching the data and sending it should be possible with the right transfer method, unfortunately in the real world tests the throughput could not be brought past the 9 kB/s achieved by the Sliding Window method. If however, the Sliding Window method had more

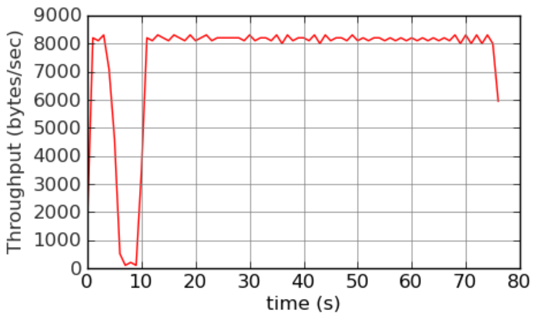
[3]This upper throughput threshold is explained in more detail in [34].

RAM available and could thus transfer longer bursts it would be able to get much closer to the desired 13 kB/s.

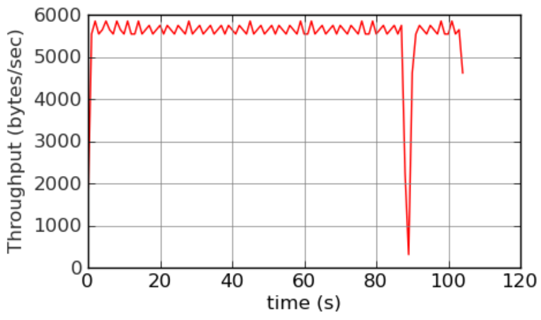
In all, from these tests, it can be concluded that the Sliding Window method is the method of choice to achieve the highest throughput over a reliable connection. This is because it consistently scored the highest throughput in the real world tests and, given more RAM, can be extended to achieve higher throughputs.



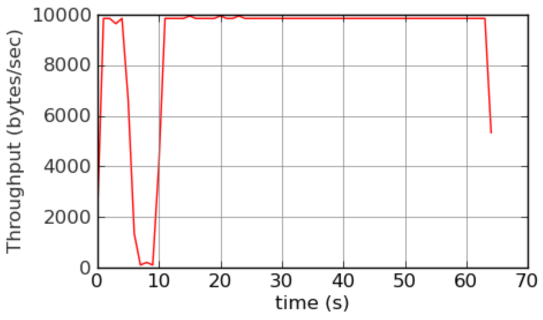
(a) Single Ack method



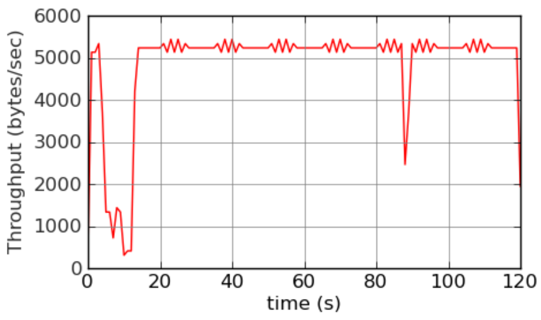
(b) Single Ack method with no printing



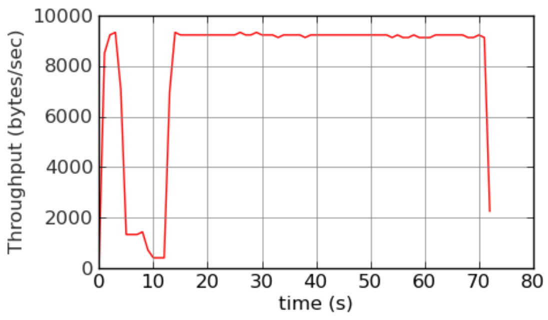
(c) Double Buffered Method



(d) Double Buffered method with no printing

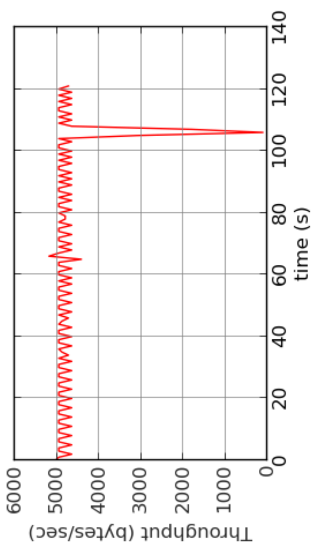


(e) Sliding Window method

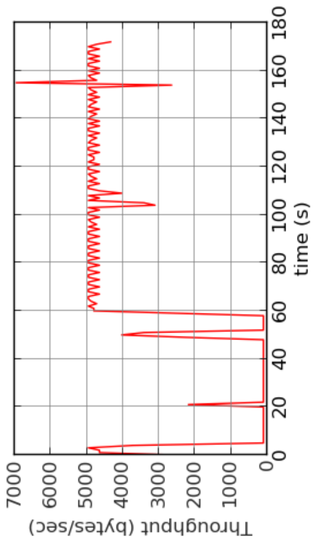


(f) Sliding Window method with no printing

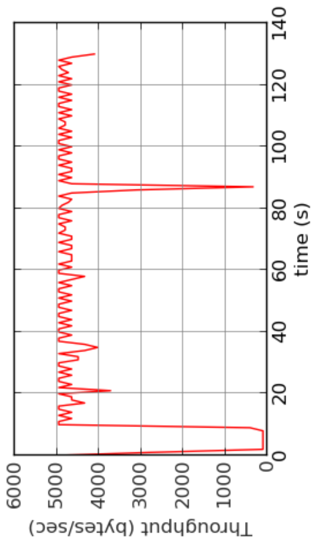
Figure 4.4: The data rates of the different methods tested



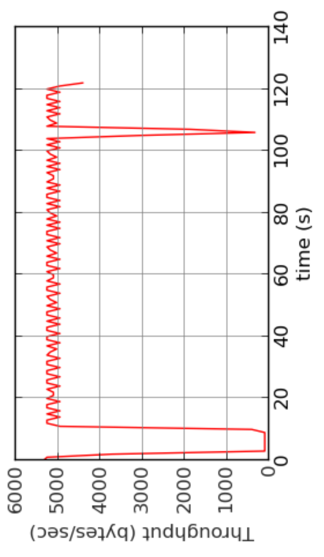
(a) Single Acknowledgement 10cm



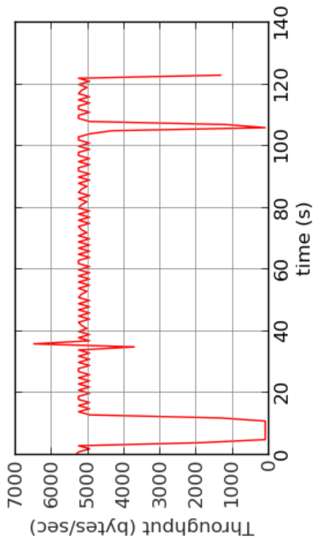
(b) Single Acknowledgement 20m



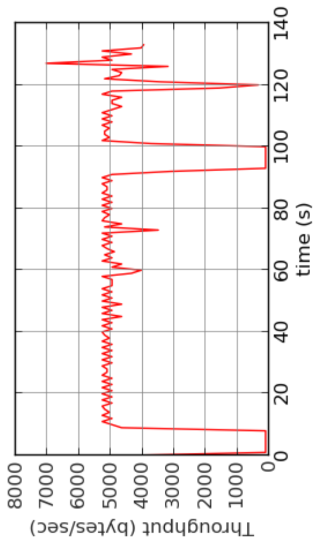
(c) Single Acknowledgement 40m



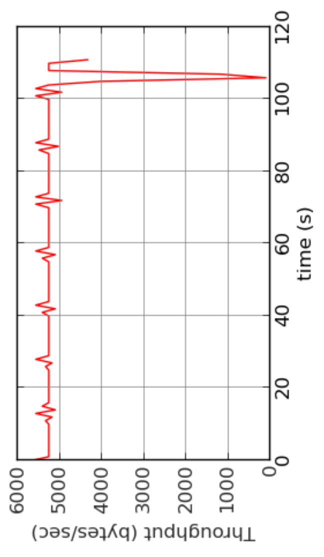
(d) Double Buffered 10cm



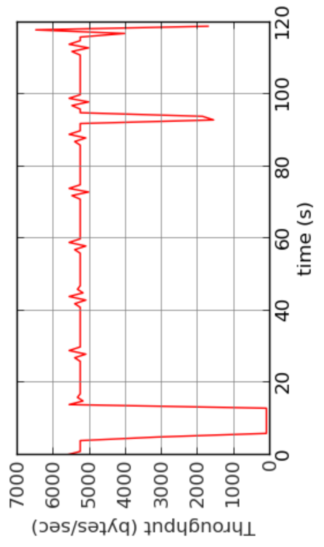
(e) Double Buffered 20m



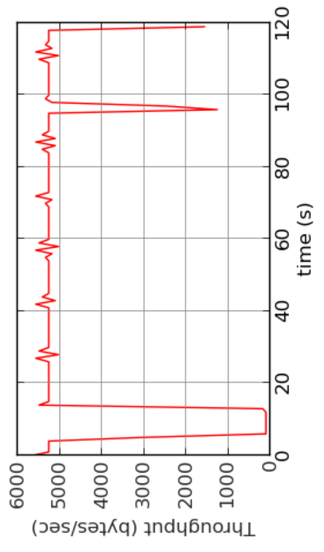
(f) Double Buffered 40m



(g) Sliding Window 10cm



(h) Sliding Window 20m



(i) Sliding Window 40m

Figure 4.5: The throughputs of the different methods at different distances

Chapter 5

Selecting Nodes using RSSI

This chapter discusses tests performed to investigate how the Received Signal Strength Indicator - RSSI - can be used to select the right nodes to download data from. It starts by giving the motivations for selecting the right storage nodes. The chapter then goes on to describe the methods used to investigate RSSI values at different distances and how the data was analysed. Finally the chapter ends by proposing a node selection method based on the information gathered in the investigation of RSSI values and the situations outlined in the motivation section.

Motivation

When downloading data from multiple storage nodes it is important for the mobile sink to choose nodes from which it will be able to maintain a connection for the longest period of time. This ensures that the sink spends less time polling for new nodes or dealing with interrupted connections and more time transferring data from storage nodes, increasing the overall throughput of the mobile sink.

The selection method implemented in this project, is a naive first come first served method, the mobile sink will gather data from first responder node to reply to the mobile sink's beacon. Given the situation presented in Figure 5.1 there is a possibility that as the mobile sink travels through the network determined by nodes A, B, C and D that it will spend most of the first half of its journey attempting to connect to nodes B and C (which will result in poor throughput given that the sink is at the very edge of their ranges and many packets will be dropped) instead of connecting to node A - which will be closer to the sink for a longer period of time and thus will be able to transfer more data to the sink. In order to create a system for the selection of nodes this project assumes that the mobile sink is travelling along a predetermined path through the nodes following a roughly straight line. Deducing a system which will be effective for more complex paths is left for further research.

It is known from [30, 36, 40, 42, 46] that the RSSI value of a connection generally gets smaller as two nodes connecting wirelessly are placed farther apart. It is also known from [28] that this fall off is not uniform. As the mobile sink is moving it can be assumed that if the RSSI values are polled for long enough the trend of the recorded RSSI values will be able to indicate whether or not a mobile sink is moving towards a node or away from it.

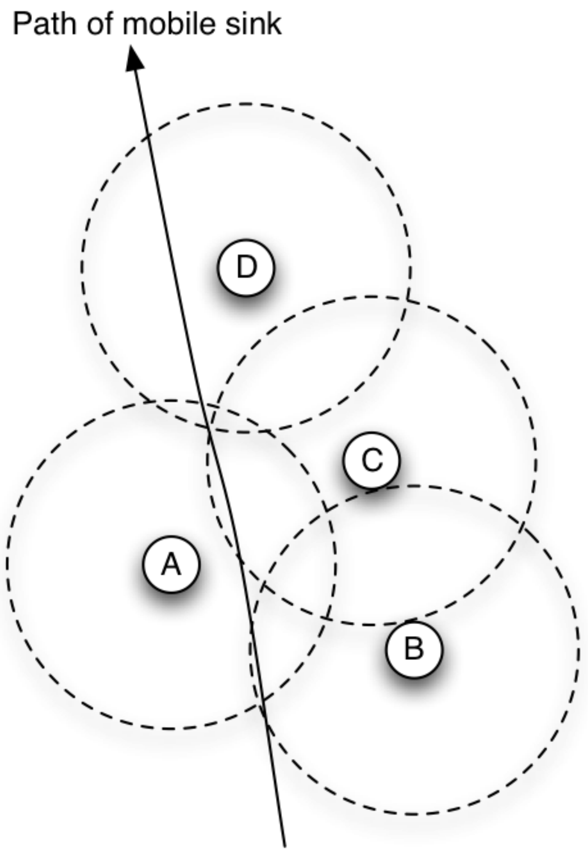


Figure 5.1: A situation where a smart node selection method is needed to select node A over B,C & D

Measurement Method

To measure the RSSI levels of the connection the same test set up and location was used as for the throughput measurements of the transfer methods. Instead of standing statically at various different distances from the responder node however, the mobile sink was moved - taking care to ensure that the height and the orientation of the node did not change - towards the responder node from a distance of roughly 100m away. This distance was chosen because it is outside the communication range of the responder and therefore it would be possible to observe how the RSSI values changed as the sink was moved into range of the responder. In order to measure the distance at which the RSSI values were recorded, a GPS tracker was used to log the longitude and latitude of the mobile sink. The GPS data was processed to give the distance between the two nodes at a given time by calculating the distance between the coordinates of the mobile sink and the coordinates of the responder node for every second.

The RSSI values were gathered by a module written for the embedded system every time an arriving packet was reported to the application layer by the networking stack. This was done to ensure that the RSSI values recorded would reflect the RSSI values which would be presented to the mobile sink when it receives responses to its beaconing.

Once the GPS logs and the RSSI logs had been gathered they were processed by a python script written to parse the logs and measure the distance from the responder node at a particular time and the corresponding RSSI value. The results of which can be seen in Figure 5.2. In addition to measuring the RSSI, the throughput of the system at a given distance was also measured.

Selection Method

As can be seen on the graph in Figure 5.2 the throughput of the connection between the mobile sink and the responder is loosely correlated with an increase in the RSSI. More precisely, one can see that the maximum data rates are achieved when the RSSI values become higher than 15 dBm. Another feature that can be seen in the graph is that the RSSI values increase exponentially as the sink approaches the responder. These two features allow two conclusions to be drawn. First if a connection is to be effective the average RSSI value for a responder node should be at least 15 dBm. Second, the trend in the variation of the the RSSI values can be used to determine whether the distance between two nodes is increasing or decreasing given several readings over a period of time.

The conclusions drawn from the RSSI and throughput data lead to the following requirements for a node selection system. The measurement of the RSSI must

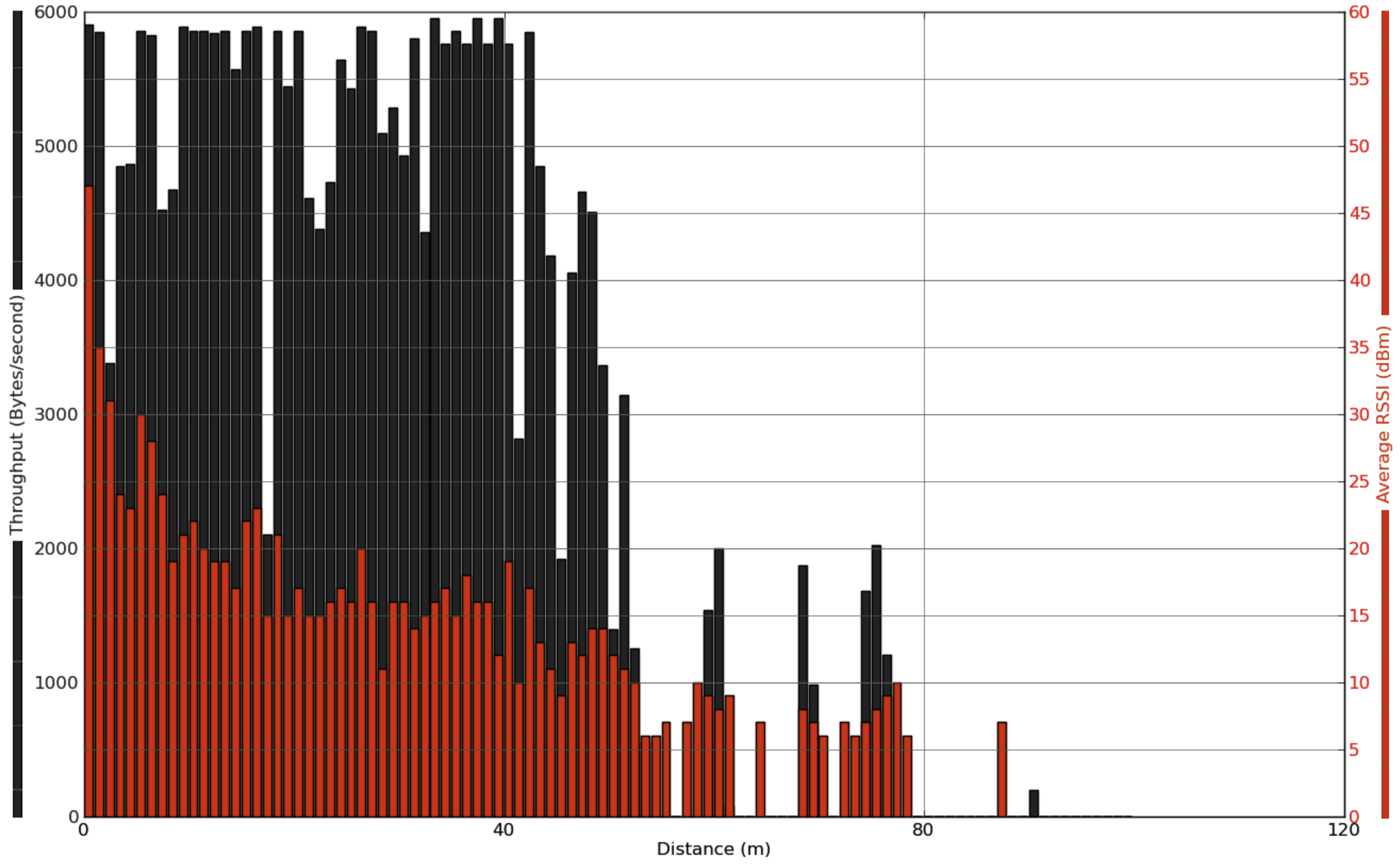


Figure 5.2: The RSSI and throughput values at different distances from the responder node

happen over a period of time to be able to determine if the RSSI values are trending towards higher values or whether they are trending towards lower values. A node which consistently reports RSSI values of less than 15 dBm should be ignored as it is too far away to initiate a useful connection. With these requirements it is possible to outline a basic system for the selection of nodes based on their RSSI values.

In order to effectively explain the node selection method it is useful to keep the picture in Figure 5.3 in mind. The method aims to maximise the amount of time the sink and the responder have to transfer data while the connection has an RSSI which is above the cutoff threshold. This implies initiating the connection at position A in Figure 5.3. To do this the method polls the RSSI values of a connection with a responder over a certain period of time, the period of time taken to poll the RSSI values should be proportional to the speed of the mobile sink in an attempt to ensure that the RSSI values cover a range of distances. A weighted average of the RSSI values is then taken to ensure that they are above the threshold[1] - 15dBm. Once it has been determined by the weighted RSSI average that a given node is eligible for connection, the slope of the linear regression of the collected RSSI values is calculated, if it is a positive slope the final rank of the node will be 200-(weighted average RSSI). If, on the other hand the slope is a negative slope the final rank of the node will simply be the weighted average RSSI. This means that if a sink and responder are in a situation corresponding to position A in Figure 5.3 the responder will have a rank of close to 185 - the maximum rank. Whereas if a sink and responder are in a situation corresponding to position C in Figure 5.3 the responder will have a rank close to 15. This ensures that

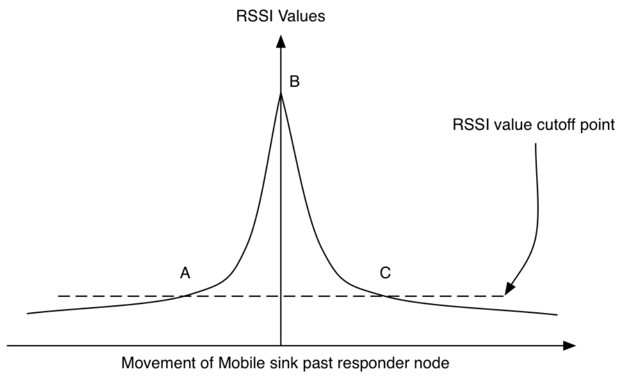


Figure 5.3: The RSSI values of the connection between a sink and a responder as the sink goes past the responder. Position A is the most desireable point to start a connection, B is the place where a selectiong method based only the RSSI value would start the connection and C is the least desirable place to start a connection.

[1]The average is weighted in favour of more recent RSSI values in an effort to ensure that stale RSSI values do not impact the average too much

the desirability of a node is accurately represented by the ranking system. Once all of the possible responders have been ranked, the sink will then choose the responder with the highest rank.

Chapter 6

Conclusions

Summary

This project aimed to build a mobile sink which will integrate with the WildSensing wireless sensor network to gather the data collected by the sensors. In particular it aimed to produce a fast, reliable, connection-oriented transfer method which could be used to transfer data from the network onto the mobile sink. It also aimed to investigate the link between RSSI values and the connection throughput in order to propose a way of selecting nodes of the WildSensing network which will generate the highest overall throughput.

All of the aims of this project were achieved. A mobile sink framework was created, three different transfer methods were tested on the framework in order to determine which was the fastest and a method was proposed to select which storage nodes should be selected to transfer data from. The final conclusion of the investigation of the transfer methods was that the Sliding Window method would be the best method to use as it had the highest throughput on the physical systems and had the ability to become even faster given an embedded system with more RAM memory which would allow for longer bursts.

The investigation of the correlation of the RSSI values to the data throughput yielded information that the data throughput reaches its maximum speed above an RSSI value of 15 dBm. It also revealed that the RSSI values rise exponentially the closer two nodes get to each other meaning that changes in RSSI levels will determine whether two nodes are getting closer together or are moving apart. These two findings led to the proposition of a selection method which polls potential responder nodes over a period of time while the mobile sink is moving and uses the level and variation in RSSI values to rank the potential responder nodes in order of desirability. Testing of the proposed system however, was left for future research.

Learning Outcomes

Before building this mobile sink for the WildSensing project I had never worked on embedded systems before, and it is completely different to conventional programming. It demands a much more in depth knowledge of the system being developed for on the software and hardware layers. Also, due to the space restrictions on the embedded system, I needed to be constantly aware of exactly how much space my code was using given that if I used too much the code would either not compile or cause the system to behave erratically.

When a system behaved erratically finding out what was causing the problem was incredibly difficult. Given that for the majority of the project I did not have access to the Cooja simulator, all exploration of the system had to happen through printf statements which may or may not break the time-sensitive operation observed.

Because of the novel development environment and the difficulties exploring the system on which I was working, it took a very long time to ensure predictable performance. Causing me to spend a lot more time than expected building the systems presented in the project. Which in turn limited my ability to develop more advanced transfer and node selection methods than the ones presented.

On finishing this project, I feel that I have gained an insight into the intricate nature of programming for embedded systems as well as a full understanding of wireless network communication systems. I have also gained a deep understanding of how the Contiki operating system works and more specifically how the uIP stack works. In all, I have greatly enjoyed the challenge of developing software for embedded systems and I look forward to seeing what future research will provide in the field of wireless sensor networks.

Future Work

The topic of gathering data from wireless sensor networks is an extremely interesting one which is seeing more and more attention as the subject of wireless sensor networks becomes more mature. Given more time to work on this project it would have been extremely interesting to delve deeper into the different mechanisms for delivering data to mobile sinks and extend the currently implemented system to include a more advanced version of the proposed node selection method and allow the data collection to extend past single-hop transfers into multi-hop transfers which allows the mobile sink to gather data from nodes which it cannot directly communicate with.

It would also have been very interesting to implement other transfer methods such as the selective acknowledgement scheme described in [3] or adding a packet preloading system such as the one in the Double Buffered method to the Sliding Window method in order to reduce the time taken to reply to burst acknowledgements. Another option to improve the data collection process would be to implement the mobile sink on a more powerful machine and only use the radio of the embedded system for data communication meaning that the mobile sink would be able to handle incoming data at a much faster rate and less time would be wasted while the mobile sink processes the data coming in from the responder node it is connected to.

The node selection method could, with more research be adapted to handle complex paths more precisely and rank nodes more effectively than they are currently being ranked. Also, a more complex node selection system would be able to incorporate the amounts of data stored in each responder node in order to make decisions based on projected future node visibility as well as the amount of data needed to be fetched from each node.

Bibliography

[1] I Akyildiz, W Su, Y Sankarasubramaniam, and E Cayirci. Wireless sensor networks: a survey. *Computer networks*, Jan 2002.

[2] H Balakrishnan, S Seshan, and E Amir. Improving tcp/ip performance over wireless networks. *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 2–11, Jan 1995.

[3] Hari Balakrishnan, Venkata Padmanabhan, Srinivasan Seshan, and Randy Katz. A comparison of mechanisms for improving tcp performance over wireless links. *IEEE/ACM Transactions on Networking (TON)*, 5(6), Dec 1997.

[4] T Bova and T Krivoruchka. Reliable udp protocol. Technical report, Network Working Group - Cisco Systems, Feb 1999.

[5] M Buettner, G Yee, and E Anderson. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. *Conference On Embedded Networked Sensor Systems*, Jan 2006.

[6] V Bychkovsky, B Hull, and A Miu. A measurement study of vehicular internet access using in situ wi-fi networks. *Proceedings of MobiCom '06*, Jan 2006.

[7] A Cerpa, J Wong, and L Kuang. Statistical model of lossy links in wireless sensor networks. *... in Sensor Networks*, Jan 2005.

[8] I Chakeres and E Belding-Royer. The utility of hello messages for determining link connectivity. *... Symposium on Wireless Personal Networks*, Jan 2002.

[9] I Chatzigiannakis, A Kinalis, and S Nikoletseas .... Fast and energy efficient sensor data collection by multiple mobile sinks. *Proceedings of the 5th ACM international workshop on Mobility management and wireless access*, Jan 2007.

[10] I Chatzigiannakis, A Kinalis, and S Nikoletseas. Efficient data propagation strategies in wireless sensor networks using a single mobile sink. *Computer Communications*, Jan 2008.

[11] K Chin, J Judge, and A Williams. Implementation experience with manet routing protocols. *ACM SIGCOMM Computer Communication Review*, 32:49–59, Jan 2002.

[12] Contiki. 6lowpan implementation - http://www.sics.se/ adam/contiki/docs-uipv6/a01109.html, May 2010.

[13] Contiki. Sicslowmac implementation - http://www.sics.se/ adam/contiki/docs-uipv6/a01103.html, May 2010.

[14] Dunkels. Contiki - a lightweight and flexible operating system for tiny networked sensors. *Local Computer Networks, 2004. 29th Annual IEEE International Conference on Local Computer Networks*, pages 455 – 462, 2004.

[15] A Dunkels, J Alonso, T Voigt, H Ritter, and J Schiller. Connecting wireless sensornets with tcp/ip networks. *Wired/Wireless Internet Conference*, (2004):143–152, 2004.

[16] Vladimir Dyo, Stephen A Ellwood, David W Macdonald, Andrew Markham Cecilia Mascolo, Bence Pa sztor, Niki Trigoni, and Ricklef Wohlers. Poster abstract: Wildlife and environmental monitoring using rfid and wsn technology. *SenSys*, November:1–2, Sep 2009.

[17] N Finne, J Eriksson, Z He, O Schmidt, and J Abeillé. Ip-based sensor networking with contiki.

[18] D Ganesan, B Krishnamachari, A Woo, and D Culler. Complex behavior at scale: An experimental study of low-power wireless sensor networks. *Citeseer*, Jan 2002.

[19] C Gomez and C Bormann. 6lowpan working group e. kim internet-draft etri intended status: Informational d. kaspar expires: September 9, 2010 simula research laboratory, Jan 2010.

[20] B Greenstein, A Pesterev, C Mar, E Kohler, and J Judy. Collecting high-rate data over low-rate sensor network radios. *University of California*, 2006.

[21] Y Gu, D Bozdag, R Brewer, and E Ekici. Data harvesting with mobile elements in wireless sensor networks. *Computer networks*, Jan 2006.

[22] P Gupta and P Kumar. The capacity of wireless networks. *mit.edu*, 1999.

[23] E Hamida and G Chelius. Analytical evaluation of virtual infrastructures for data dissemination in wireless sensor networks with mobile sink. *Proceedings of the First ACM workshop on Sensor Networks*, Jan 2007.

[24] J Hill, R Szewczyk, A Woo, S Hollar, and D Culler. System architecture directions for networked sensors. *ACM Sigplan Notices*, Jan 2000.

[25] J Ioannidis, D Duchamp, and G Maguire . . . . Ip-based protocols for mobile internetworking. *Proceedings of the conference on Communications architecture & protocols*, pages 235–245, Jan 1991.

[26] H Karl and A Willig. *Protocols and architectures for wireless sensor networks.* Wiley, Jan 2007.

[27] I Khemapech, I Duncan, and A Miller. A survey of wireless sensor networks technology. *Proc. of The 6th Annual PostGraduate ...*, Jan 2005.

[28] D Kotz, C Newport, and C Elliott. The mistaken axioms of wireless-network research. *Citeseer*, Jan 2003.

[29] F Lewis. Wireless sensor networks. *Smart environments: technologies*, Jan 2004.

[30] Loyz. *MTM-CM2000-MSP Manual ver1*, Aug 2007.

[31] David J.C MacKay. *Information Theory, Inference, and Learning Algorithms*, volume 1. Cambridge University Press, Apr 2005.

[32] K Madria. . . . Mobile data and transaction management. *Information Sciences*, Jan 2002.

[33] Mohammad Mozumdar, Luciano Lavagno, and Laura Vanzago. A comparison of software platforms for wireless sensor networks: Mantis, tinyos, and zigbee. *Transactions on Embedded Computing Systems (TECS)*, 8(2), Jan 2009.

[34] F Österlind and A Dunkels. Approaching the maximum 802.15. 4 multi-hop throughput. *HotEmnets*, Jan 2008.

[35] J Parsons and P Parsons. *The mobile radio propagation channel.* Wiley, Jan 2001.

[36] Polastre. *Tmote Sky Datasheet*, Jan 2007.

[37] Y Sankarasubramaniam and Ö Akan. Esrt: event-to-sink reliable transport in wireless sensor networks. *Proceedings of the 4th international symposium on Mobile ad hoc networking & computing*, Jan 2003.

[38] S Schaust and I Ritter. Mobile sinks in ad-hoc sensor networks. Master's thesis, University of Hannover, Jan 2005.

[39] F Stann and J Heidemann. Rmst: Reliable data transport in sensor networks. *Proceedings of the First International Workshop on Sensor Networks*, Jan 2003.

[40] M Sugano, T Kawazoe, Y Ohta, and M Murata. Indoor localization system using rssi measurement of wireless sensor network based on zigbee standard. *From Proceeding (538) . . .*, Jan 2006.

[41] J Sun and J Sauvola. Mobility and mobility management: A conceptual framework. *Proceedings of the 10th IEEE International . . .*, Jan 2002.

[42] B Walke. *Mobile radio networks: networking, protocols, and traffic performance.* Wiley, Jan 2002.

[43] X Wang and I Akilydiz. A survey on sensor networks. *IEEE Communication Magazine*, Jan 2002.

[44] Jon Wilson. *SENSOR TECHNOLOGY HANDBOOK: Chapter 22*. Dec 2004.

[45] J Zhao and R Govindan. Understanding packet delivery performance in dense wireless sensor networks. *Conference On Embedded Networked Sensor Systems*, Jan 2003.

[46] G Zhou, T He, and S Krishnamurthy. Models and solutions for radio irregularity in wireless sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 2:221 – 262, Jan 2006.

Appendix A: System Behaviour Flow Charts

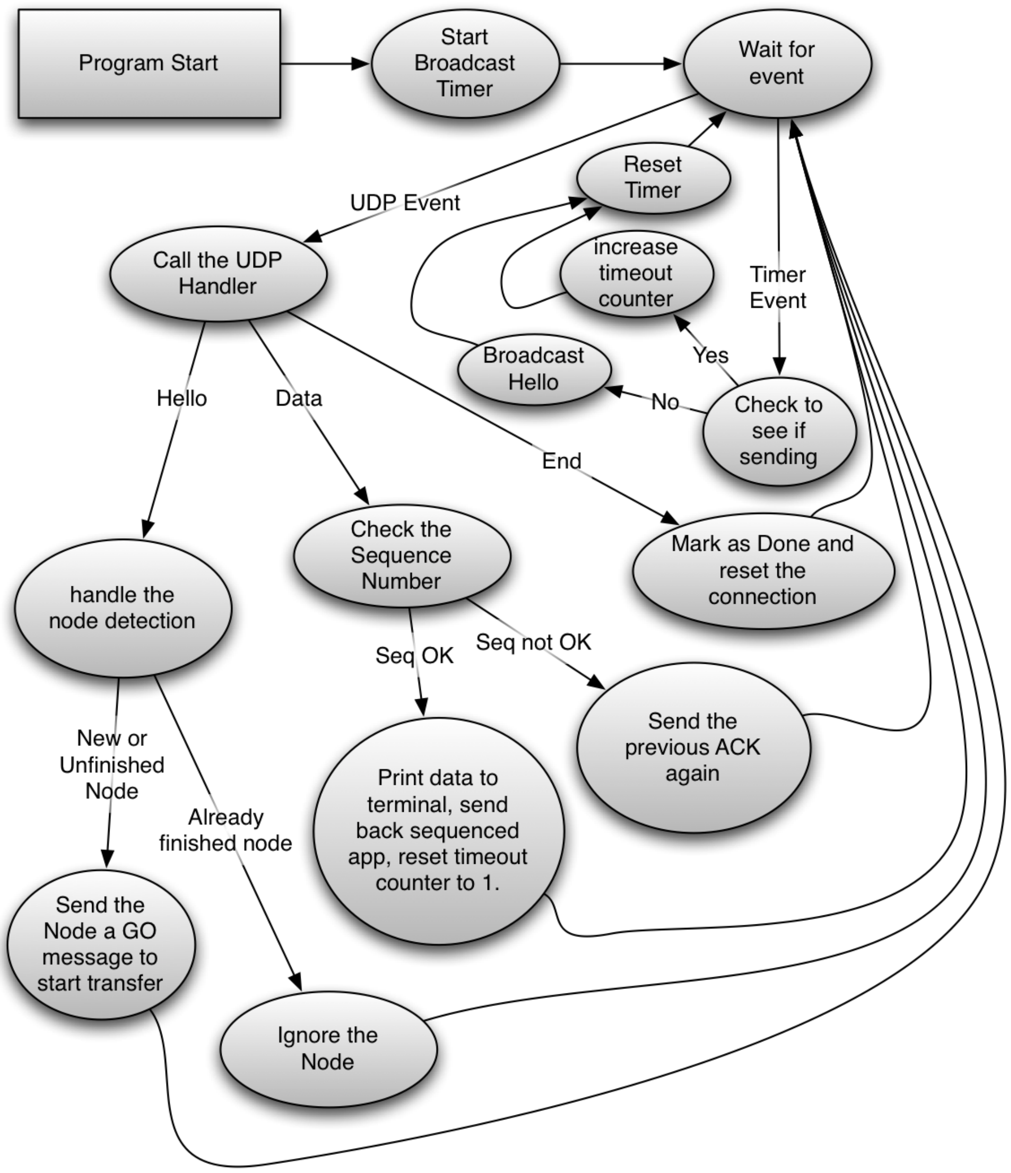


Figure 1: The Single Acknowledgement Method Sink Behaviour

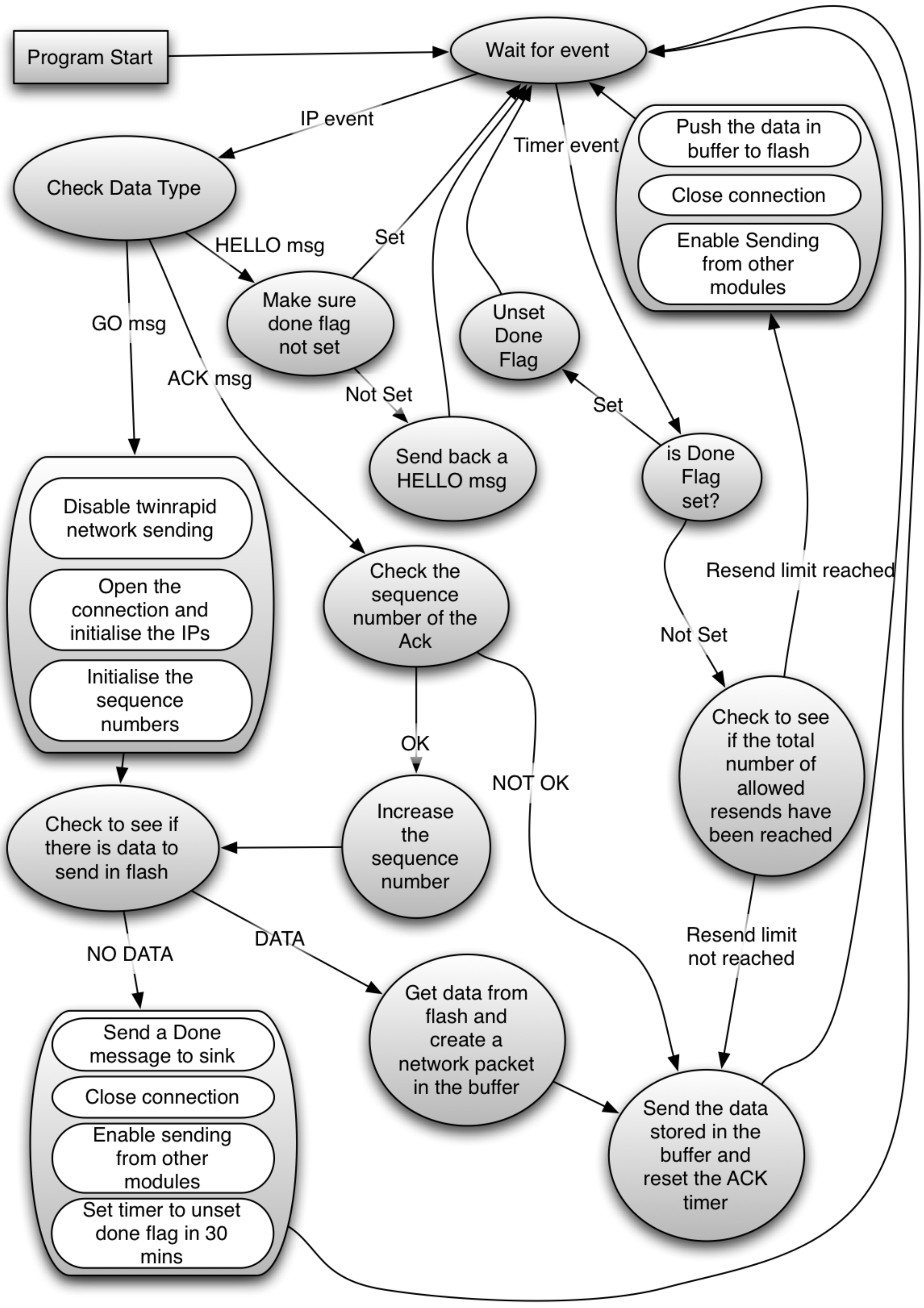

Program Start
Wait for event
IP event
Timer event
Check Data Type
HELLO msg
Set
Make sure done flag not set
Not Set
Send back a HELLO msg
GO msg
ACK msg
Unset Done Flag
Set
is Done Flag set?
Not Set
Push the data in buffer to flash
Close connection
Enable Sending from other modules
Disable twinrapid network sending
Open the connection and initialise the IPs
Initialise the sequence numbers
Check the sequence number of the Ack
OK
NOT OK
Increase the sequence number
Check to see if there is data to send in flash
NO DATA
DATA
Get data from flash and create a network packet in the buffer
Send a Done message to sink
Close connection
Enable sending from other modules
Set timer to unset done flag in 30 mins
Check to see if the total number of allowed resends have been reached
Resend limit reached
Resend limit not reached
Send the data stored in the buffer and reset the ACK timer

Figure 2: The Single Acknowledgement Method Responder Behaviour

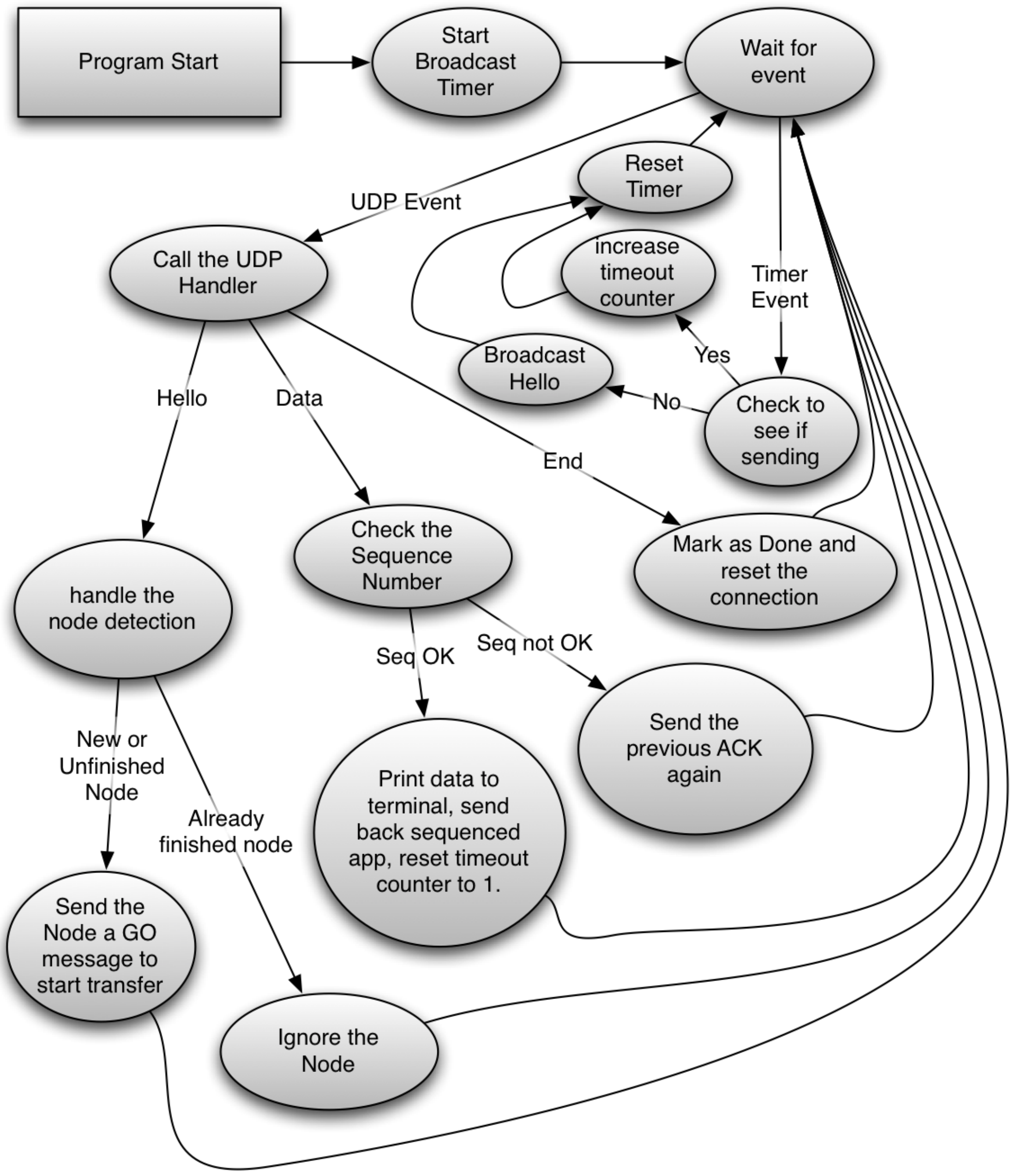


Figure 3: The Double Buffered Method Sink Behaviour

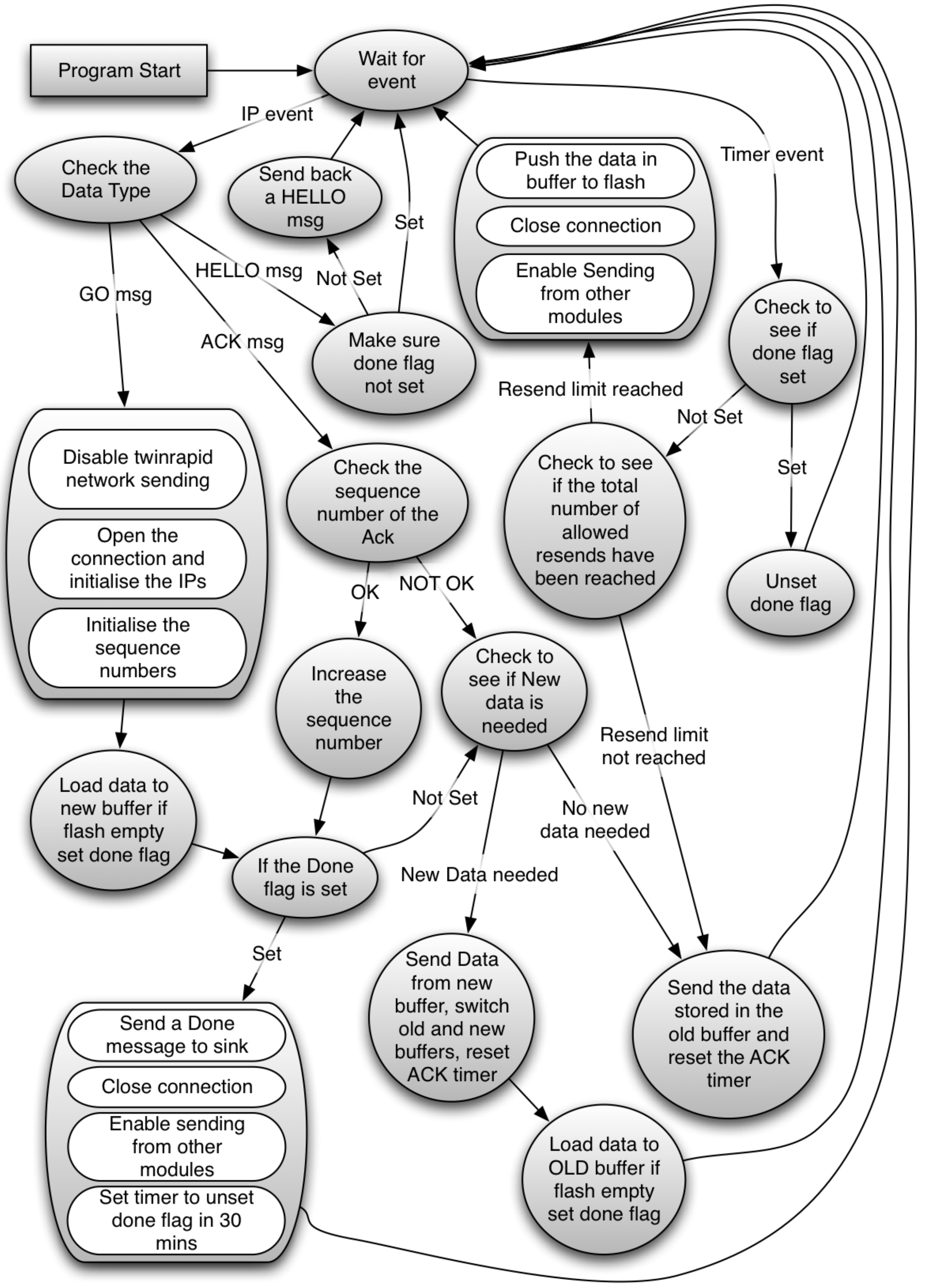

Program Start
Wait for event
IP event
Timer event
Check the Data Type
Send back a HELLO msg
Set
Push the data in buffer to flash
Close connection
Enable Sending from other modules
GO msg
HELLO msg
Not Set
ACK msg
Make sure done flag not set
Check to see if done flag set
Resend limit reached
Not Set
Set
Disable twinrapid network sending
Open the connection and initialise the IPs
Initialise the sequence numbers
Check the sequence number of the Ack
Check to see if the total number of allowed resends have been reached
OK
NOT OK
Unset done flag
Increase the sequence number
Check to see if New data is needed
Resend limit not reached
Load data to new buffer if flash empty set done flag
Not Set
No new data needed
If the Done flag is set
New Data needed
Set
Send Data from new buffer, switch old and new buffers, reset ACK timer
Send the data stored in the old buffer and reset the ACK timer
Send a Done message to sink
Close connection
Enable sending from other modules
Set timer to unset done flag in 30 mins
Load data to OLD buffer if flash empty set done flag

Figure 4: The Double Buffered Method Responder Behaviour

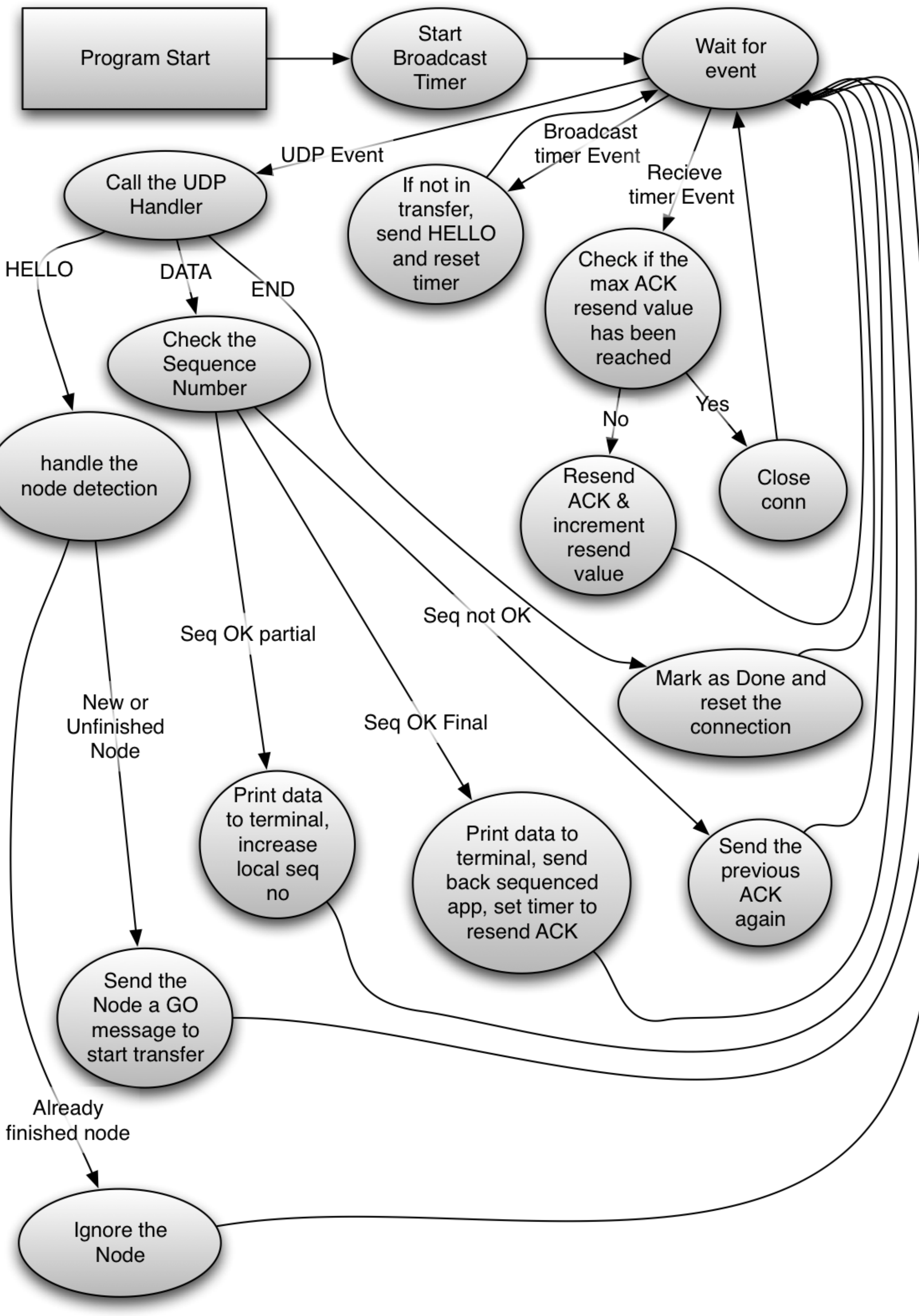

Program Start
Start Broadcast Timer
Wait for event
UDP Event
Broadcast timer Event
Recieve timer Event
Call the UDP Handler
If not in transfer, send HELLO and reset timer
Check if the max ACK resend value has been reached
HELLO
DATA
END
Check the Sequence Number
No
Yes
handle the node detection
Resend ACK & increment resend value
Close conn
Seq OK partial
Seq not OK
Mark as Done and reset the connection
New or Unfinished Node
Seq OK Final
Print data to terminal, increase local seq no
Print data to terminal, send back sequenced app, set timer to resend ACK
Send the previous ACK again
Send the Node a GO message to start transfer
Already finished node
Ignore the Node

Figure 5: The Sliding Window Method Sink Behaviour

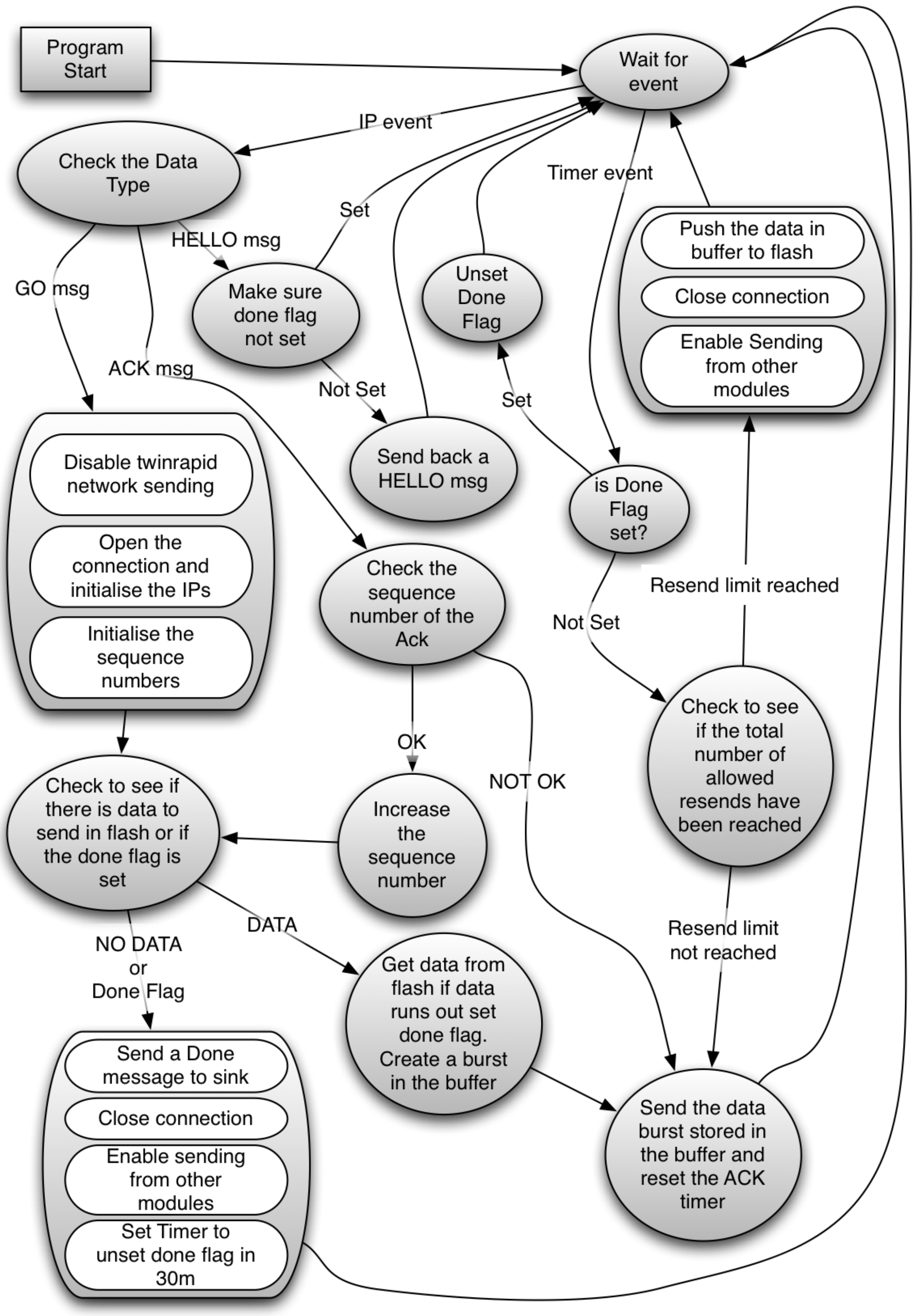

Program Start
Wait for event
IP event
Check the Data Type
Set
Timer event
HELLO msg
GO msg
Make sure done flag not set
Unset Done Flag
Push the data in buffer to flash
Close connection
Enable Sending from other modules
ACK msg
Not Set
Set
Disable twinrapid network sending
Open the connection and initialise the IPs
Initialise the sequence numbers
Send back a HELLO msg
is Done Flag set?
Check the sequence number of the Ack
Resend limit reached
Not Set
OK
NOT OK
Check to see if the total number of allowed resends have been reached
Check to see if there is data to send in flash or if the done flag is set
Increase the sequence number
DATA
NO DATA or Done Flag
Resend limit not reached
Get data from flash if data runs out set done flag. Create a burst in the buffer
Send a Done message to sink
Close connection
Enable sending from other modules
Set Timer to unset done flag in 30m
Send the data burst stored in the buffer and reset the ACK timer

Figure 6: The Sliding Window Method Responder Behaviour

Appendix B: Python Scripts

Converting GPS and RSSI Logs into RSSI and Throughput at a Given Distance

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
sinkFile = open("./rssi.txt", "r")
gpsFile = open("./gps.txt", "r")
sinkData = []
dataPerSecond = {}
class data_msg(object):
        def __init__(self, rssi=-55, time=0):
                self.rssi = rssi
                self.time = time
x=0

#Parse the data in the log file from the mobile sink
#this will give us a bunch of RSSI values associated to
    times
while sinkFile:
        line = sinkFile.readline()
        s = line.split(',',2)
        n = len(s)
        if n==1:
                break
        if 'rssi' in s[1]:
                rs = s[1].split(':')
                print int(rs[1])

                t = s[0].split()
        times = t[1].split(':')
        secTime = float(times[2])
        secTime += float(times[1])*60

        data_packet = data_msg(50+int(rs[1]),secTime)
        sinkData.append(data_packet)

#Parse the GPS data, this will give us a bunch of
#lattitudes and longitudes connect to time values
while gpsFile:
    line = gpsFile.readline()
    s = line.split(',')
    n = len(s)
    if n==1:
        break
    if s[1] == '$GPGGA':
        t = s[0].split()
        times = t[1].split(':')
        secTime = float(times[2])
        secTime += float(times[1])*60
        s[3] = float(s[3])
        s[5] = float(s[5])
        if s[4] == 'S':
            s[3] = -s[3]
        if s[6] == 'W':
            s[5] = -s[5]
        dataPerSecond[secTime] = [(s[3],s[5]),float(s[9])]

#Calculate all of the distances by comparing each
#lattitude and longitude to the starting coordinates.
#All of the times are also updated to be an offset
#from the first time recorded
tList = dataPerSecond.keys()
tList.sort()
firstTime = tList[0]
data = dataPerSecond[tList[0]]
firstPos = data[0]
firstErr = data[1]
del dataPerSecond[tList[0]]
dataPerSecond[0] = [0, data[1]]
del tList[0]
R = 6371
for t in tList:
    newKey = t - firstTime
```

```
        data = dataPerSecond[t]
        del dataPerSecond[t]
        pos = data[0]

        dist = np.arccos(np.sin(firstPos[0]) * np.sin(pos
            [0]) + np.cos(firstPos[0]) * np.cos(pos[0]) * np.
            cos(firstPos[1]-pos[1])) * R
        dist = dist/4
        dataPerSecond[newKey] = [dist,data[1]]


timeList = [0]
dataList = [192]
timeRSSIList = []
levelList = []

#This creates two arrays, one of time values
#the other of RSSI where the times are
#aligned with the times from the gps
for elements in sinkData:
        elements.time = elements.time-firstTime
        timeRSSIList.append(elements.time)
        levelList.append(elements.rssi)
origtime = sinkData[0].time
prevtime = 0

#This creates two arrays, one for time
#and one for data rate. The times are
#aligned with the gps times. The data rate
#goes in increments of 192, which is the
#amount of data transferred at every RSSI
#measurement
for elements in sinkData:
        if elements.time>len(timeList):
                dataList.append(0)
                timeList.append(floor(elements.time))
        else:
                dataList[len(dataList)-1] += 192
largestDist = 0

#This creates two lists, one for distances
#and one for times
for time, dat in dataPerSecond.iteritems():
        if dat[0]>largestDist:
                largestDist=dat[0]
for time, dist in dataPerSecond.iteritems():
        dataPerSecond[time] = (largestDist-dist[0],dist[1]+
            firstErr)
tempList = dataPerSecond.items()
tempList.sort()
times, data = zip(*tempList)
distances, error = zip(*data)
error = list(error)
distances = list(distances)
times = list(times)

#these three loops round off all the
#times to the nearest second
for i in range(0,len(times)):
        times[i] = round(times[i])
for i in range(0,len(timeList)):
        timeList[i] = round(timeList[i])
for i in range(0,len(timeRSSIList)):
        timeRSSIList[i] = round(timeRSSIList[i])

avgRSSI = []
for tim in times:
        avgRSSI.append(0)

meanQuant = 0
timePos = 0

#This calculates the average RSSI per second
#from all the RSSI values recorded each sec
for x in range(0,len(times)):
        if timePos < len(timeRSSIList)-1:
                while timeRSSIList[timePos] == times[x]:
                        meanQuant = meanQuant + 1
                        avgRSSI[x] = avgRSSI[x] + (levelList
                            [timePos])
                        timePos = timePos +1
                        if timePos == len(timeRSSIList):
                                break
        if meanQuant > 0:
                avgRSSI[x] = avgRSSI[x]/meanQuant
        meanQuant = 0

RSSIatDist = []
for x in range(0, 100):
        RSSIatDist.append(0)
for x in range(0, len(distances)):
        distances[x] = round(distances[x])
```

```

#This now uses the avgRSSI per second and
#calculates the average RSSI per metre
meanQuant = 0
for x in range(0,100):
        for i in range(0, len(distances)):
                if x == distances[i]:
                        print meanQuant
                        RSSIatDist[x] = RSSIatDist[x] +
                           avgRSSI[i]
                        if avgRSSI[i]>0:
                                meanQuant = meanQuant+1
        if meanQuant>0:
                RSSIatDist[x] = RSSIatDist[x]/meanQuant
        meanQuant = 0

avgData = []
for tim in times:
        avgData.append(0)

#This calculates the average data rate
#per second
meanQuant = 0
timePos = 0
for x in range(0,len(times)):
        if timePos < len(timeList)-1:
                while timeList[timePos] == times[x]:
                        meanQuant = meanQuant + 1
                        avgData[x] = avgData[x] + (dataList[
                           timePos])
                        timePos = timePos +1
                        if timePos == len(timeList):
                                break
        if meanQuant > 0:
                avgData[x] = avgData[x]/meanQuant
        meanQuant = 0

DataatDist = []
for x in range(0, 100):
        DataatDist.append(0)
for x in range(0, len(distances)):
        distances[x] = round(distances[x])

#this takes the data rate per second and
#translates it into data rate at dist x
meanQuant = 0
```

```
for x in range(0,100):
        for i in range(0, len(distances)):
                if x == distances[i]:
                        DataatDist[x] = DataatDist[x] + avgData[i]
                        if avgData[i]>0:
                                meanQuant = meanQuant+1
        if meanQuant>0:
                DataatDist[x] = DataatDist[x]/meanQuant
        meanQuant = 0

#plot the data rate
fig = plt.figure(figsize=(24,18))
ax1 = fig.add_subplot(111)
ax1.xaxis.grid(color='gray', linestyle='solid')
ax1.yaxis.grid(color='gray', linestyle='solid')
p1 = ax1.bar(range(0,100), DataatDist,color='#2B2B2B' )
#ax1.set_ylim(ymin=-42.0)
#ax1.set_ylim(ymax=-32.0)
ax1.set_xlabel('Distance␣(m)')
# Make the y-axis label and tick labels match the line color.
ax1.set_ylabel('Throughput␣(Bytes/second)', color='#2B2B2B')
for tl in ax1.get_yticklabels():
    tl.set_color('#2B2B2B')

#plot the RSSI
ax2 = ax1.twinx()
majorLocator = MultipleLocator(40)
ax2.xaxis.set_major_locator(majorLocator)
minorLocator = MultipleLocator(5)
ax2.yaxis.set_major_locator(minorLocator)
ax2.yaxis.grid(color='gray', linestyle='solid')
ax2.bar(range(0,100), RSSIatDist,color='#EE3A00' )
#ax2.plot([1,50,200],[1,2,400], '--r')
ax2.set_ylim(ymin=0.0)
ax2.set_ylim(ymax=60.0)
ax2.set_ylabel('Average␣RSSI␣(dBm)', color='#EE3A00')
for tl in ax2.get_yticklabels():
    tl.set_color('r')
plt.show()
```

Converting Cooja Logs into Throughput

```
import sys
import matplotlib.pyplot as plt
from pylab import *
sinkFile = open("./radio.txt", "r")
#refFile = open("./reference_data.txt", "r")
sinkData = []
#refData = []
class data_msg(object):
        def __init__(self, type=0, priority=0, hopcount=0,
            delay=0, seq_no=0, data=[], time=0):
                self.type = type
                self.priority = priority
                self.hopcount = hopcount
                self.delay = delay
                self.seq_no = seq_no
                self.data = data
                self.time = time

#parse the sink file to get data this
#is actuall the radio log
while sinkFile:
        line = sinkFile.readline()
        s = line.split('\t',4)
        n = len(s)
        if n==1:
                break
        if len(s[3])>90:
                        x = x+1
                        secTime=int(s[0])/1000
                        data_packet = data_msg(0,0,0,0,0,[],
                            floor(secTime))
                        sinkData.append(data_packet)
timeList = [64]
levelList = []
origtime = sinkData[0].time
prevtime = 0

#increase the rate by 64 for each packet sent
#in a given second. (64 = 192/3) because there
#are 3 fractured packets sent by Cooja for every
#192 packet send.
for elements in sinkData:
        elements.time = elements.time-origtime
        if prevtime<elements.time:
```

```
                    timeList.append(64)
                    prevtime = elements.time
        else:
                    timeList[len(timeList)-1] += 64

#print the graph
fig = plt.figure(figsize=(5,3))
fig.subplots_adjust(bottom=0.17)
fig.subplots_adjust(left=0.15)
ax1 = fig.add_subplot(111)
ax1.xaxis.grid(color='gray', linestyle='solid')
ax1.yaxis.grid(color='gray', linestyle='solid')
ax1.plot(range(0,int(len(timeList))),timeList, '-r')
ax1.set_xlabel('time␣(s)')
# Make the y-axis label and tick labels match the line color
    .
ax1.set_ylabel('speed␣(bytes/sec)', color='#2B2B2B')
for tl in ax1.get_yticklabels():
    tl.set_color('#2B2B2B')

plt.show()
```

Converting Printed Data into Throughput

```
import sys
import matplotlib.pyplot as plt
from pylab import *
sinkFile = open("./sink.txt", "r")
sinkData = []
class data_msg(object):
        def __init__(self, type=0, priority=0, hopcount=0,
            delay=0, seq_no=0, data=[], time=0):
                self.type = type
                self.priority = priority
                self.hopcount = hopcount
                self.delay = delay
                self.seq_no = seq_no
                self.data = data
                self.time = time
x=0

#parse log file to get out information
while sinkFile:
        line = sinkFile.readline()
        s = line.split(',',2)
        n = len(s)
        if n==1:
                break
        if len(s[1])>90:
                        x = x+1
                        t = s[0].split()
                        times = t[1].split(':')
                        secTime = float(times[2])
                        secTime += float(times[1])*60
                        secTime += float(times[0])*3600
                        data_packet = data_msg(0,0,0,0,0,[],
                           secTime)
                        for i in range(18, 98):
                                if i%2 == 0:
                                        data_packet.data.
                                           append(int(s[1][i
                                           :i+2],16))
                        sinkData.append(data_packet)
timeList = [48]
levelList = []
origtime = sinkData[0].time

#Each item in list represents 1 WildSensing
```

```
#data packet so increment the data per sec
#by 48 each time
prevtime = 0
for elements in sinkData:
        elements.time = elements.time-origtime
        if elements.time>len(timeList):
                timeList.append(48)
                prevtime = elements.time
        else:
                timeList[len(timeList)-1] += 48

#Print the graph
fig = plt.figure(figsize=(5,3))
fig.subplots_adjust(bottom=0.17)
fig.subplots_adjust(left=0.15)
ax1 = fig.add_subplot(111)
ax1.xaxis.grid(color='gray', linestyle='solid')
ax1.yaxis.grid(color='gray', linestyle='solid')
ax1.plot(range(0,int(len(timeList))),timeList, '-r')
ax1.set_xlabel('time␣(s)')
# Make the y-axis label and tick labels match the line color
    .
ax1.set_ylabel('speed␣(bytes/sec)', color='#2B2B2B')
for tl in ax1.get_yticklabels():
    tl.set_color('#2B2B2B')

plt.show()
```

List of Figures

1.1 A diagram of the WildSensing Wireless Sensor Network 4

2.1 The general behaviour of the Mobile Sink 9
2.2 The general behaviour of a responder node 10
2.3 The Responder Module in the WildSensing System 11
2.4 The Networking stack used by the project 13

3.1 An example of the Single Acknowledgement transfer method with a connection fail after the first successfully sent packet. 17
3.2 An example of the behaviour of the Double Buffered transfer method with connection interruption at the second acknowledgement. 18
3.3 The behaviour of the sink when a full burst is received. (a) is when all the sequence numbers match and (b) is when the local sequence number is higher than the arriving sequence numbers. 19
3.4 The behaviour of the sink when receiving a partial burst. Left is the situation where no data arrives, middle is an example when a packet from the burst is missing and right is the situation when the end of the burst fails to arrive. 21
3.5 The behaviour of the responder in the different acknowledgement situations. Left: Timeout. Middle: Partial Acknowledgement. Right: Full Acknowledgement. 22

4.1 Throughput, resend and timeout counts in the virtual environment . 28
4.2 The throughput and the number of timeouts in the real world tests . 29
4.3 The effect of various different operations on the throughput of the nodes 30
4.4 The data rates of the different methods tested 32
4.5 The throughputs of the different methods at different distances 33

5.1 A situation where a smart node selection method is needed to select node A over B,C & D 35
5.2 The RSSI and throughput values at different distances from the responder node 37
5.3 The RSSI values of the connection between a sink and a responder as the sink goes past the responder. Position A is the most desireable point to start a connection, B is the place where a selectiong method based only the RSSI value would start the connection and C is the least desirable place to start a connection. 38

1 The Single Acknowledgement Method Sink Behaviour 48
2 The Single Acknowledgement Method Responder Behaviour 49
3 The Double Buffered Method Sink Behaviour 50
4 The Double Buffered Method Responder Behaviour 51
5 The Sliding Window Method Sink Behaviour 52
6 The Sliding Window Method Responder Behaviour 53