THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF MECHANICAL AND MANUFACTURING ENGINEERING

Intelligent Sharing of Occupancy Grids over wireless Adhoc networks on mobile platforms in urban environments

Matthew Cork

3207888

Bachelor of Engineering (Mechatronic Engineering)

Submitted: October 2012

Supervisors: Dr J.E. Guivant
Mr M. Woods

Declaration of Originality

I, Matthew Cork declare that this thesis is my own work. All content from other sources has been acknowledged and cited.

Matthew Cork

3207888

Abstract

Sharing Occupancy Grid (OG) data over a wireless adhoc mobile network is one of the major problems faced in using multiple platforms to map an area.

The approach was split into the inter-process communication and networking manager. The inter-process communication was achieved using a circular buffer, written in shared memory, and was able to efficiently share between multiple processes without data corruption. The circular buffer implements an array which can hold ten different instances of the data structure being shared.

The network manager used a TCP connection, over an adhoc network, to establish communication protocol between the platforms. The network manager had a low overhead which allowed for intelligent communication and proved able to share data across the network.

The final protocol and framework proved to be an intelligent solution to share OG data over a wireless adhoc network.

Acknowledgements

I would like to firstly thank the Mavstar team for the equipment and support to help me in my thesis.
Next I would like to thank my supervisors Dr Jose Guivant and Mr Michael Woods for their support and ideas to help me in completion of my Thesis.
I would also like to thank Matthew P. Grosvenor for his guidance and expertise in networking.
I would also like to thanks Graham Cork and Alex Lester for proof reading my thesis and correcting my grammar.
Lastly I would like to thank my friends and family for their support while writing this Thesis.

Table of Contents

Nomenclature

ACK	Networking Acknowledgement
IPC	Inter-Process Communication
MANET	Mobile Adhoc NETworks
MAV	Micro Aerial Vehicle
MUTEXES	MUTual EXclusion
NACK	Negitive Acknowledgment
NTP	Network Time Protocol
OG	Occupancy Grid
RAM	Random Access Memory
SCP	Secure Copy Protocol
SHM	SHared Memory Communication
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

List of Figures

Chapter 1: Introduction

1.1 - Importance of OG Sharing

Occupancy Grids (OG) are an important part of mapping while operating mobile platforms in an ever changing urban environment. This presents many problems for the platform. One of the major problems with mapping is the need for multiple devices to map simultaneously. This allows for a faster and more accurate map to be created but creates a new set of problems.
The major problem for multiple platforms is trying to find an efficient way to share the OG data. This is important as all platforms need to have the same map to be able to localise while also avoiding each other and plan paths around obstacles.
Another major problem found in mapping is the need to setup infrastructure to connect all the platforms. This can be quite costly (either money or time) and isn't always possible. Adhoc networks were designed to eliminate the infrastructure requirements and allow for communication between the mobile platforms without the need of a centralised wireless access point.

1.2 - Thesis Objectives

The main aim of this thesis is;

- to develop a protocol and framework to intelligent share occupancy grid(OG) data between processes on the platform and,
- create a network manager to intelligent share the OG data over an wireless adhoc network.

This involved implementing a way to share data from programs that created and manipulated the data to other programs that required it and then using the network manager to communicate it over the network.

1.3 - Thesis Overview

Chapter 2 contains a look at different research into OG sharing, sharing of data over adhoc networks and different routing protocols.
Chapter 3 contains a technical report of the network structure layers and the built-in advantages of each of the relevant layers.
Chapter 4 is the implementation that was designed to be tested as the OG sharing protocol and framework.
Chapter 5 outlines the experiments that were undertaken to ensure that the framework was an adequate solution.
Chapter 6 is the results of the experiments undertaken in Chapter 5.
Chapter 7 is the discussion of the results from the experiments.
Chapter 8 outlines the future work that could be taken to increase efficiency of sharing OG data.
Chapter 9 contains the conclusion of thesis.

Chapter 2: Literature Review

The literature for this thesis has been split up into four sections. They are Inter-process communication, Bandwidth Estimation, Routing Protocols and OG sharing.

Computer systems are machines that constantly have interprocessing communication[3]. Ford and Hamacher[3] discussed a hardware implementation for increased efficiency. They proved that this was benefial however it would required a redesign of the hardware to support it.

Graham explored the use of “BINS”to implement inter process communication. Their approach consisted of a set of structures that are inherited by the parent program. The bin had two functions ‘PLACE’and ‘TAKE’that are used by the programs to publish their data.

The main disadvantage of this approach is programs need to be spawned by the same parent to have access the same data. Another disadvantages are the programs are unable to easily select the message they want and the BINS can only store data structures that are programmed for (int, string). One of the most common IPC with built in data corruption protection is the use of a circular buffer. The circular buffer idea to avoid corruption is a common solution for device drivers [5]. This is often used as it can allow for multiple readers to read one person’s data without the chance of corruption happening. The main advantage of this approach is it has been proven to be effective and corruption free way to share data between one writer and multiple readers. The major disadvantage of this approach is that it hasnt been tested for multiple readers and multiple writers

Sumathi and Thanamani[10] discuss an approach for collision avoidance and bandwidth estimation for intelligent sending protocols. They proved the Implicit pipeline backoff method reduced the quantity of data sent while also increasing the throughput of the wireless nodes. It was designed for multiple hop networks, so it handles efficiently the communication. The main disadvantage of this approach is it is written on the link layer and also would need to have unique identifiers added to distinguish between each platform as currently the MAC protocol on the link layer or the IP protocol on the Network layer handles this. Differing Routing algorithms have been research to try and determine the most efficient protocol to implement. The major disadvantage with these approach is they are all written on the link layer. Although this makes them unable to be used the research if able to be used on the application layer for my protocols.

Pham Van's[13] researched the issues of bandwidth-hogging and time-sensitivity that is required when streaming video (high volume data) over an adhoc network by proposing a new proactive architecture. The paper analyses the re-transmission time and explains how the new architecture handles the incoming packets. The procedure for handling the incoming packets includes checking if the round trip time is less than a set threshold of delay. The protocol discards the packet if it is above the threshold or sends it on if it isn't. A discarded packet means that there is a faster path within the structure. The packets that have less delay are selected for retransmission to the next node in the sequence. The loss of packets is handles by an NACK which reduces the quantity of ACK messages on the bandwidth.

The paper presents an architecture that allows the efficient streaming of video in real time. The strength of this approach is that the data is sent over the quickest path and and thus doesn't flood the network. The major disadvantage is, as all nodes know if they are part of the quickest route if the structure changes suddenly their is no redundancy for this data and the data is lost. In a Dynamic MANET the structure is constantly changing and thus packets dont know the fastest path or if the node actually exists.

Putta, Prasad, Ravilla, Nath and Chandra [12] outlined the different algorithms used for routing in Mobile Adhoc Networks (MANETS). The paper cover two different reactive algorithms; Adhoc on Demand Distance Vector (AODV) and Dynamic Source Routing (DSR), and the main proactive protocol Optimised Link State Routing (OLSR). These algorithms were compared based on the following criteria; Packet Delivery Ratio, Mean end-to-end delay and Routing load. Neither reactive protocol proved to be the superior than the other.

The paper shows that if the OG was high bandwidth load it would be more efficient and reliable to use a proactive protocol however for low load data (such as control signals) the reactive protocol would be the better choice. The tests of these protocols however didn't allow for a drastically changing MANET.

Lee, Ra and Kim [7] investigated three other methods of routing; DSR and AODV with Gossiping and AODV with flooding. Gossiping is the process of the node determining whether to resend based on a fixed probability function.

It was shown that DSR was only effective for small networks. Whereas AODV with flooding was proven to reduce the end to end delay of the packets however has a high routing overhead and increase the load on the bandwidth. The AODV + Gossip was shown to be an effective method, however more research is needed to increase the throughput as it currently has low probability of success.

Abdel-Hardy and Ward [1] also tested the difference between the proactive protocol (OLSR) and the reactive protocols; AODV and Dynamic Manet On Demand (DYMO). Their tests included one video stream with five UDP data connections,

one video stream with 50 UDP connections and multiple video streams.
The tested the theory that the current protocols are acceptable for low latency applications, but they fail in situations with high latency applications and resulted in large packet loss. Their results showed that DYMO was the best protocol for smallmedium manets and for large and heavy use manets. AODV was the best for large and light use manets. OLSR was proven not to be suitable for the high latency communication.
OG sharing over adhoc networks can be done using the above protocols and methods. Some research has already gone into trying to share OG over multiple platforms to create a Network OG. A network OG is a OG that is shared across all platforms and each platform updates its own copy then syncs with the other platforms. There are many different ways of representating OGs and many different ways to share them.
Pfingsthorn and Birk [11] researched into the efficiency of sharing of the Occupancy Grid (OG) through the three following methods; Sending the occupancy grid periodically, sending the cells when they are alter or sending the sensor data used to create the OG.Their research showed that the most efficient method minimise bandwidth hogging was the sending the data set used to create the platforms OG. Each node used the same data sets that are sent across the network to generate its own OG map onboard. This method proved to be almost seven times faster than sending the cells as they are modified and over seventy times faster than sending the sections periodically.
The major disadvantage of this method is that all hosts need to have the same data set to have the same occupancy grid. In a dynamic structure used in manets the entire data set (from the beginning of the map) would need to be sent to

ensure everyone has the same map. Failure to send the whole data would cause the platform to have a different data set and would generate a different map. The MAV wouldnt be able to explore out of the network then upload the changes when it got back in range. This wouldnt allow for distributed exploration. A possible solution to this problem could be the data sets get sent but the platforms also had an ability to sync the entire map less frequently. Jang, Choi and Lee[2] explore creating the OG mapping implementation as a cluster of smaller cells. They showed that this approach allowed optimisation to be performed to increase the efficiency of all manipulations of the clusters. It also showed that the cluster were able to manipulated quicker than manipulating the same number of individual cells. Each cluster was made to be independent of the cells around it which would simplify the sending process through the network manager as the sections would already be made for you. The timing would be similar to Pfingsthorn and Birk second testing of changing the alter cells[11].

Chapter 3: Technical Report

Network communication can modelled on five main network layers[4]. These layers are the physical, datalink, network, transport and the application as shown in Figure 1.

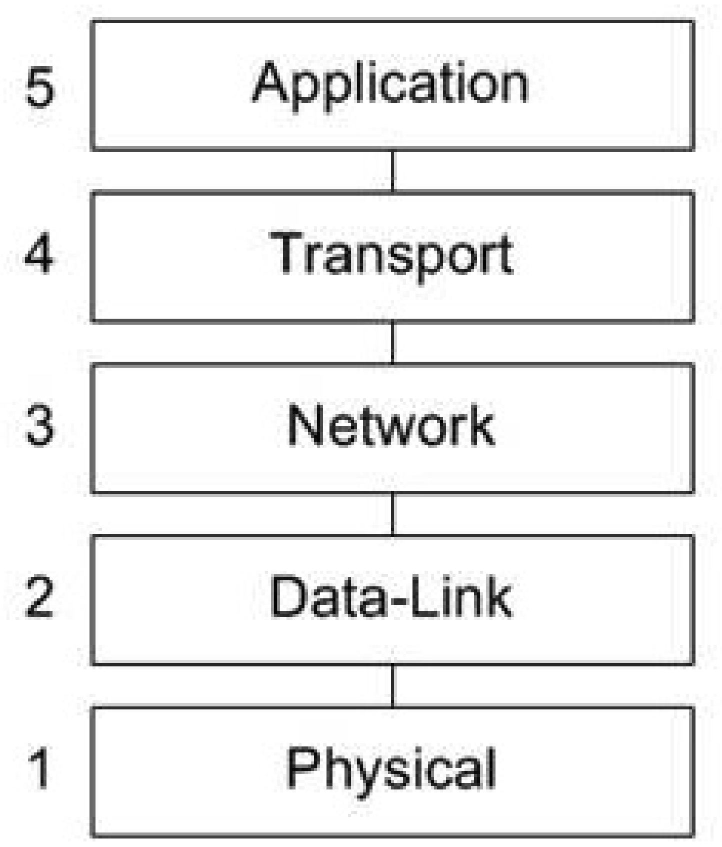


Figure 1: Network Levels Structure [4]

The transmission of data from the application layer is passed down, through the transport, network, datalink and then finally to the physical link.
The physical link is the only link that has a direct connection to another devices.

This base link depends on the method used to connect the devices. This could be via copper, fibre or even wireless. The physical layer sends its data using electrical pulses and waves on the electromagnetic frequency. This is received by other wireless devices for communication.
The link layer sits above the physical layer. This layer is mostly implemented in the Network Interface Card (NIC). The NIC is the onboard hardware that handles the data passed from the physical layer. The NIC uses the Message Authentication Code (MAC) protocol which gives a unique MAC address for each NIC device. It is responsible for the:

- Flow control,
- Error Detection and Correction,
- Controlling the Duplexity.

[6] The link layer has a buffer that is of a limited size. This means that the NIC hardware needs to control the flow from the host to avoid causing the loss of data through data overflow. This is done through two main methods; software and hardware controlled flow. The NIC is in charge of the hardware flow control and will ensure that the data is less than the buffer. The second option is done through software. The only part of the link layer that is implemented in software is the driver which allows the program to interact with the Operating System (OS). This driver controls the the software flow and allows the user to change the setting as needed[6].
The error detection and correction is accomplished by using various parity methods. The parity method is given a parity value which can check to see if there is

corruption in the packet. By using this method if a corruption has happened the program will be able to detect where the corruption is and recover from it. These approaches however will not be able to fix major corruption of the data but will be able to detect it. The NIC other main task is to establish whether the user requires a full duplex channel or only a half duplex. If the user only requires one way transmission they can control the duplexity of the channel through the driver. The two types of communication in the link layer are point to point and broadcasting. The point to point communication requires a single transmitter and a single receiver. The broadcasting communication allows for multiple senders and receivers which causes a problem with over saturation of the bandwidth from any user. The three main methods used to ensure that bandwidth is shared are:

- Channel Partitioning (FDM and TDM)
- Random Access
- Turns based sending.

[6] The two methods used in channel partitioning are the Frequency Division Method (FDM) and the Time Division Method (TDM). Both these methods require division of resources to allow for multiple senders and receivers. The FDM divides up the bandwidth frequency with the number of concurrent programs assigning each program a certain range of the frequency. This then allows the program to use its frequency as it sees fit. The TDM divides the time into frames and assigns each subsection to a different program. Each program is only allowed to transmit during its allocated time.

The random access principle starts by transmitting on the full bandwidth. When

it detects a collision it will wait a randomly determined time then retransmit all the data. This approach is simple and effective for a small quantity of programs.
The last category is the turn based sending algorithms. There are many different turn based algorithms that are able to be used however the most common and effective are the ALOHA and SLOTTED ALOHA[6]. The SLOTTED ALOHA method divides up the link layer frame into segments based on the bits of each frame divide by the full bandwidth on channel. The programs only transmit on the start of their period. This requires the synchronisation of all programs, to ensure that two programs don't transmit at the same time. The ALOHA starts by transmitting on the full bandwidth. When it detects the collision it will determine the wait time based on a random probability. This approach doesnt require any of the programs to be synchronised and still is a simple and effective method of collision avoidance. The link layer has many effective means to allow communication between devices as well as error checking/detection built in.
The next layer is the network layer. This layer is implemented entirely in the software and is controlled by the OS. The network layer uses the IP protocol. Due to the dynamic nature of this layer the IP address is often issued by a Dynamic Host Control Protocol (DHCP) Server. For any fixed system, the IP address is usually statically defined. This minimises the network overhead for connection and allows for a quicker establishment of communication between the network layers of different devices. The two main functions of this layer is either forwarding or routing.
Network layer forwarding is the ability for the network layer to determine the correct input to the correct output. This layer is able to sustain many connections simultaneously. When an input is received from one connection that needs to go

to another, the network layer forwarding function is responsible for ensuring that the correct input is aligned with the correct output.
The second function of the network layer is the controlling of the routing between hosts. Each network layer device has a forwarding table that shows the route to all hosts it has encountered. As the network receives data from new hosts this table is automatically updated by the OS. This data is used by the forwarding function to find the correct output to the host required.
The fourth layer is the transport layer. This layer requires the ports of both the destination and source hosts. This layer uses two different protocols User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).
The UDP protocol is a connectionless transmission. Due to the lack of connection state, UDP overhead is small. However as the connection cant be verified, congestion control on this level is almost impossible and this makes the protocol unreliable for data that requires a pattern.
TCP is the most common connection orientated protocol. It is established using a three way handshake. This handshake ensures the reliability of the data transfer. The protocol has three main methods to ensure reliability These are

- StopnWait (SW)
- Go-Back-N (GBN)
- Selective Repeat (SR)

[6] The TCP protocol can detect and handle congestion in the bandwidth. It does this by throttling the data allowed to be sent by the application layer. The disadvantage of this layer is the limited buffer. To overcome this problem the transport

layer provides feedback to the application layer. This requires the application layer to ensure that the data sent is less than the buffer size of the transport layer. The TCP layer allows for a full duplex connection (i.e both sending and receiving).
The SW reliability method is the slowest of the three. When it sends a packet, it waits for acknowledgment that the packet has been received. The GBN method will send a certain number of packets. It will keep sending the packets in the window until it has received the lowest packet number's acknowledgment. The window will then move to the next unacknowledged packet and sending all packets greater than this one regardless whether they have been sent or acknowledged before. The SR method also requires a window that keeps track of the the frames sent. As the sender receives an acknowledgement, it marks that packet as being received however the windows will only move to the last unacknowledged packet. The packets will only be resent when a timeout period for that packet is reached. This is the quickest method as it only resends packets that timeout.
The last layer is the application layer. This layer is the program that is using the network functions. The are no common protocols for this layer. The application layer was chosen for the intelligent sharing of the occupancy grid. This allows for all the safeguards of the lower layers to be used and allows the program to be written in higher level code with more intelligent functionality.

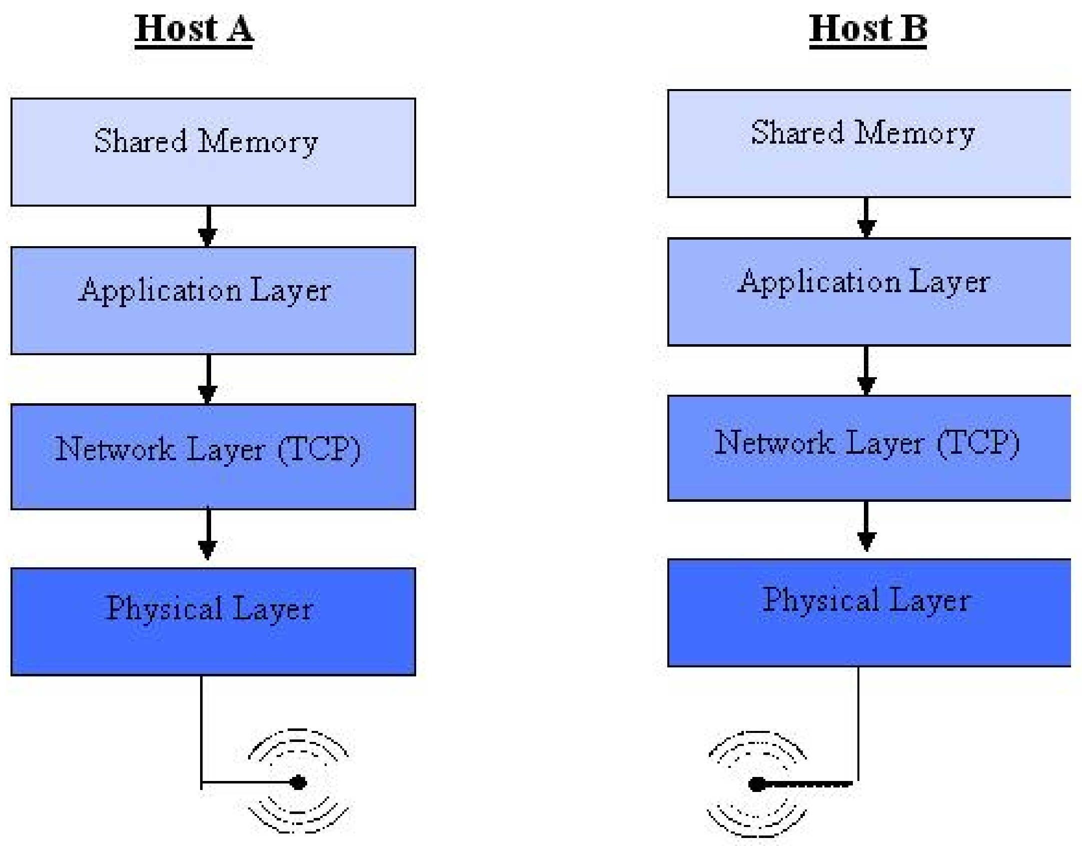


Figure 2: Relevant Network Levels Structure

Figure 2 shows the relevant layers for this thesis.

Most networking is based on two main types of connection; server-client connection and node based connection. The server client method is the first main type of connection. Figure 3 shows the graphical representation of the approach. This type involves a centralised server that has all the information. The clients then connect to this server and request the information from it. It is usually used when their are few sources of information but many clients. This approach allows for multiple clients to get the same data easily as they know exactly where the source is. This approach fails when a platform needs to be both a client and a source.

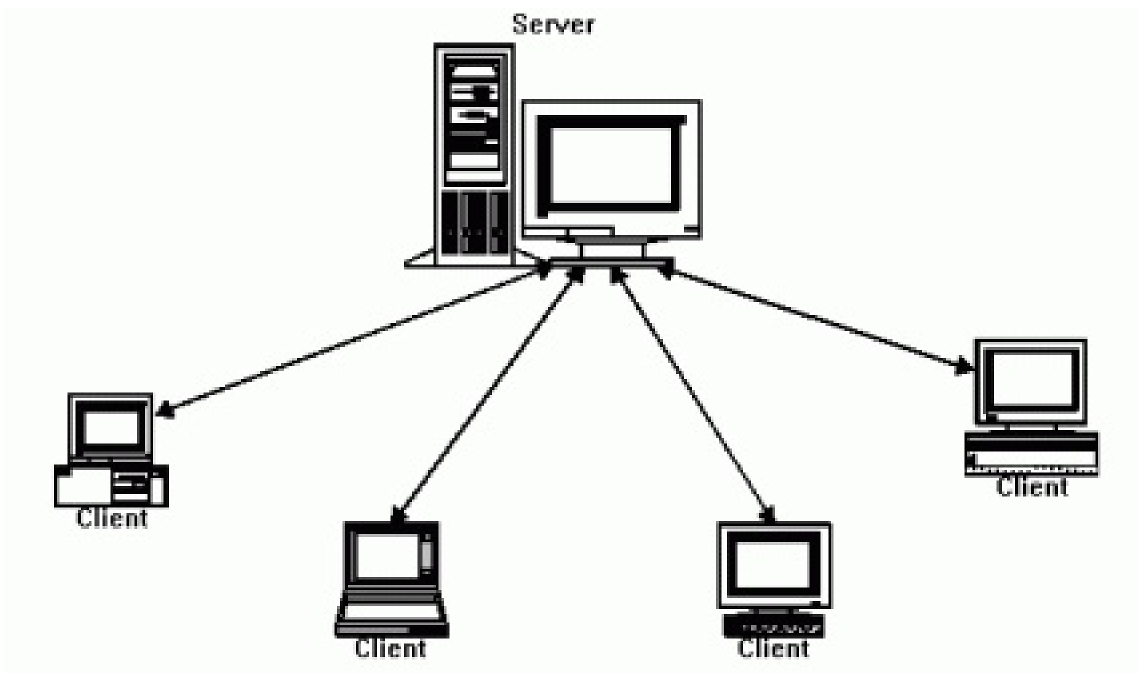


Figure 3: Server-Client Connection

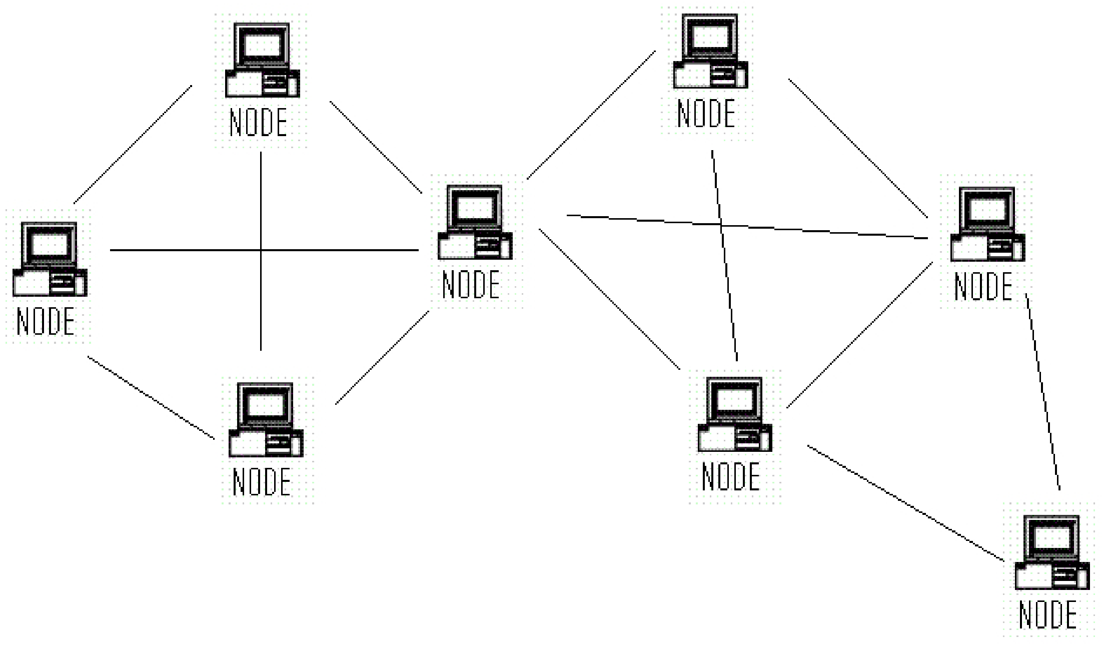


Figure 4: Node-Node Connection

The second main connection is a node to node based communication style as show in Figure 4. Node communication is a manager that performs as both a server and client in one program. It requires having a constant listening port that is used to receive connection requests. Once a request is received then the program establishes connection on another port for communication between the platforms. This approach allows for two way communication between the platforms. The lower network layers handle the connection to ensure that both nodes dont send at the same time. This connection is used when both platforms need to communicate information to each other. Node to node connection also performs better than server-client when the number of sources and number of client are similar. The node connection doesn't always need to be both sending and receiving data but is always capable of the functionality.

Chapter 4: The Implementation

The implementation was separated into two major logical parts to simplify the system and to allow for each section to be developed and tested individually. The modular nature of this approach allowed for both parts to be designed to increase the efficiency while keeping separate the different functionalities. Another advantage of the modular approach was it allowed the shared memory communication to be used as the major inter-process communication (IPC) for the system.

Shared Memory Protocol

The shared memory protocol used in this thesis was specialist design to allow separate programs to share data sets between them. One of the main design criteria for the shared memory communication protocol (SHM Protocol) was to allow multiple sources to publish their data while ensuring that corruption of data was minimised or eliminated. The other main design criteria was the SHM Protocol had to allow multiple receivers to read the latest data and minimise the chance of that data being read being corrupted by another set of data being over-written while the original is being read. The SHM Protocol starts by creating a file in the platform's local RAM with a particular filename (eg "OGData"). This "file"is then treated like a normal file allowing for the normal read and write functions to be used. For other people to read/write the data posted they also need to know the "file"name. These libraries have built in optimisations that make writing to and reading from memory more efficient than developing specific memory read and write functions for the shared memory. The structure of this "file"is shown in

Figure 5

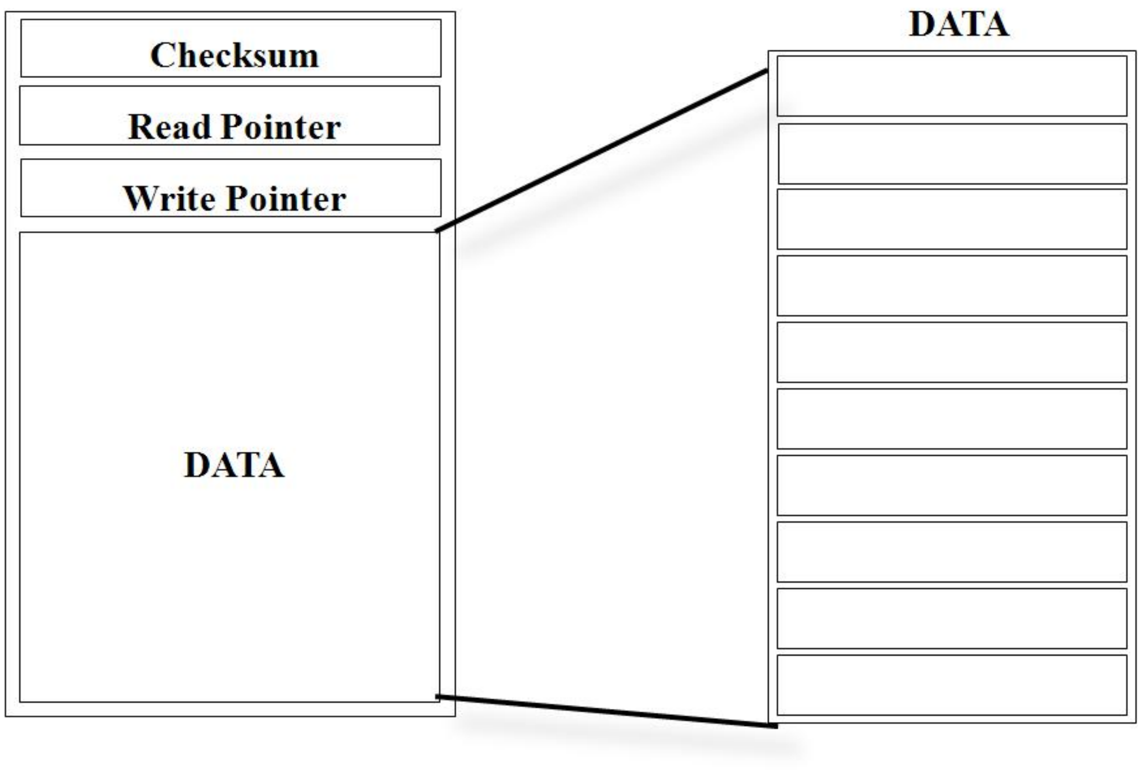


Figure 5: Shared Memory Structure

The file starts with the checksum. This checksum is used when the a program initially connects to the shared memory file(shm file). The program initially checks to see if the checksum is set to the magic number “230912”. If this checksum is set that means the file has been initialised by another program. If this value isn't set then the program is responsible to initialise the shared memory and set the initial variables.

The next part of the file is the read and write counter. Both counters are updated when write is called.

The last and biggest part of the structure is the “DATA”segment. This holds the actual data being communicated between the different programs.
One of the major problems with multiple sources reading from, or writing to, the same data set is the increased chance of corruption. One of the main solutions to this problem is using mutexes or synchronisation locks. Mutexes and synchronisation locks are mutually exclusive locks which prevents multiple people accessing the same file at the same time. This approach is unable to be used for our purpose as we want multiple people to read the data at the same time, while only one person writes at the same time. These approaches also require hardware support to be implemented properly. Due to the above reasons the shared memory protocol needed a different approach to minimise corruption of data.

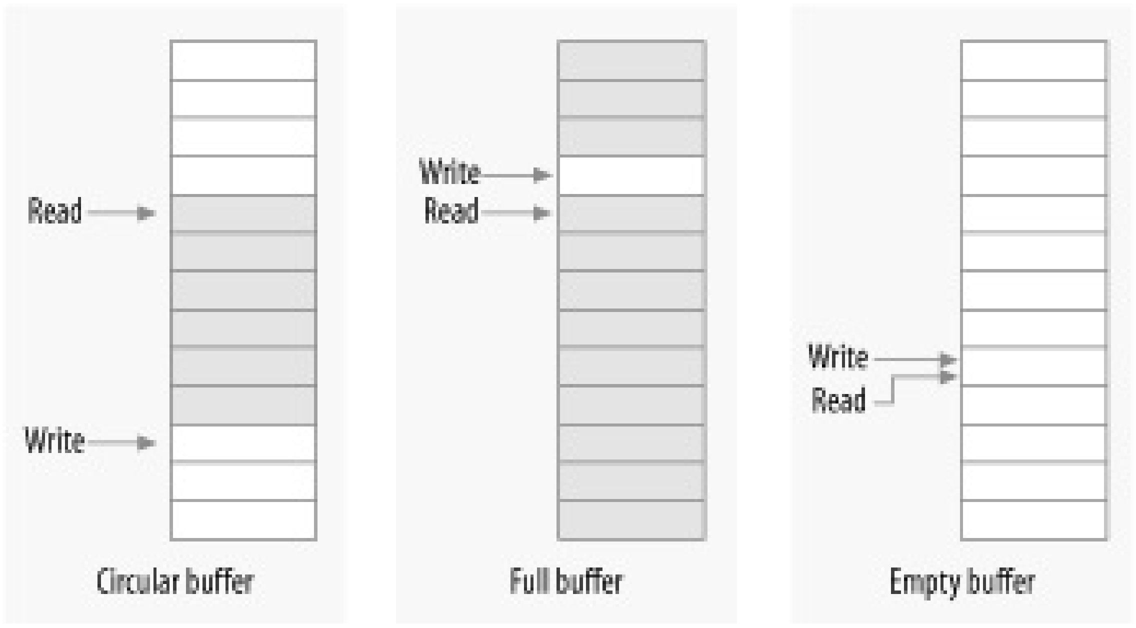


Figure 6: Circular Buffer for Device Drivers[5]

The solution chosen for the shared memory protocol is using a circular buffer to store the data. This is similar to the circular buffer used in device drivers[5] which is shown in Figure 6 however it has been adjusted to allow for multiple writers instead of only a single writer. The buffer implements an array which can hold ten different instances of the data structure being shared. The right hand side of Figure 5 shows the structure of the data segment of the shared memory. The counters included in the shared memory library (mentioned before) keep track of the next available write element of the array and the current read element. When data is written, the library updates the write counter to the next element of the array while the read counter is updated to the element that has just been written. When read is called, the program requesting data is given a pointer the the last element that was written. When data is written and the read pointer is updated, the old pointer is valid for the next nine writes and only upon the tenth write will the data be overwritten and the chance of corruption would happen. With the buffer size of ten the chance of the data being overwritten and corrupted it greatly reduced. The bigger the buffer is the smaller the chance of corruption, however the greater the buffer the more RAM is taken up in the platforms RAM. Each time read is called the update pointer will be given.
The shared memory protocol is an efficient way to communicate between programs on the platform. It allows multiple sources to write to the common data set without corruption, while also allowing multiple sources to read the latest data. This protocol allows the OG to be shared to the network manager and then sent over the network to another platform's OG program. The protocol also ensures other programs such as path planning programs can also get the OG for its use. The shared memory protocol is also used as the major IPC communication on the

platform for all programs. This is done by changing the name of the file.

Network

The network manager was written on the application layer to allow for communication between the various platforms. The program was written as a node style instead of the server-client style as all nodes need to send and receive periodically. This system is superior to the server-client style as once the connection is established the data can be sent both ways across the channel without the need for a second connection. The node is based on the following pseudocode:

```
start listening on port (6000)
connect to host
loop forever{
         broadcast "alive" message
        check for received data
        if newRecvData {
               sync with local OG(using shared memory)
        }
        check to see if new data to send
        if newData {
              send New Data
        }
}
```

Figure 7: Pseudo Code: Network Node Implementation

The protocol was designed to have a initial listening port that all nodes try to connect to each other on. This allows all nodes to only ever need to try one port number to establish a connection. When a connection is detected, the listening port passes the connection data to randomly assigned port number. This other port accepts the connection and establishes the two way pipeline to the other node. This approach make the protocol more dynamic and can mean greater number of connections for each node. After the connection is established the program then checks to see if new data has been sent over the adhoc network. The TCP connection allows the user to specify a few different options to determine if there is data. These methods are blocking, polling[8] and non-blocking[9].

Blocking is a method which only returns from the system call when data has been received. This is very useful if data is constantly expected and you can either guarantee the data will be there or if you only want to continue when the data is there. The main disadvantages of this approach is that it will lock up the program and stop it from completing any other tasks until it receives data.

Polling is a way to check the socket connection to see if data has been stored in the buffer since it was last read. This can be a good method when you are expecting data intermittently and constantly opening and closing the connection. However the major disadvantage is that the system call will block if the connection is still open but no data is being sent over it. It is mainly used when the connection is constantly established and disconnected.

For our approach the only solution that would work is using the non-blocking approach. This approach checks the connection and either returns the data being sent or returns an error code. This major disadvantage with this approach is that the node needs to do all the error checking and error-handling itself. The node program checks the return value of the "receive"system call and handles it appropriately. If data is found, it is then written to the shared memory for the OG implementation to deal with. If the connection has been disconnected from the remote end, then the program would disconnect the socket and wait for another connection. If the connection is still established but no data has been sent through then the nodes skips the writing to shared memory step.

Chapter 5: Experiments

Experiment 1 - Shared Memory

The shared memory implementation is a way to minimise the chance of corruption of the shared data using a circular buffer. The corruption is caused by one program over writing the data currently being read. This means the program reading gets a combination of old data and new data which is the corruption. The shared memory experiments are separated into two stages for testing.

Experiment 1.1 - Benchmarking Shared Memory

There is currently no benchmark for the size of data shared through the shared memory to determine at what size the corruption is reliable. The first experiment was designed to test structures of different sizes with no multiple data slots for the circular buffer (i.e circular buffer size was one). The purpose of the experiment was to show the size at which corruption first occurred and to develop a benchmark in which data corruption reliably occurred. Using the benchmark allows for future comparisons to be made.
The experiment consist of two programs accessing the shared memory. The first program was responsible for writing an array of single integers into the shared memory.

```
set integer initial value
loop forever{
        set all elements of array to integer
        write to share memory
}
```

Figure 8: Pseudo Code: Write program for Shared Memory Testing

Two instances of this program were used to write different values to the shared memory.
The second program was reading the value from shared memory and determining if the data had been corrupted.

```
loop forever{
        read from shared memory
        check array for corruption
}
```

Figure 9: Pseudo Code: Read program for Shared Memory Testing

Corruption was be determined by checking all the values of the array and if the integers arent all the same the data was corrupted. The method to obtain the benchmark is outlined below.

Method

1. Change Array Size
2. Run Both Write Programs and Read Program
3. Monitor Output to check if corruption

4. Repeat Steps 1-3 until corruption reliably appears

Experiment 1.2 - Circular Buffer Size

The second experiment relied on the benchmark obtained in the first experiment to compare the chance of corruption by varying the sizes of the circular buffer. This purpose of the experiment was to prove that the circular buffer could minimise or eliminate the corruption of data and to obtain a reliable circular buffer size for all inter-process communications.

The experiment started off using an array, the size determined in experiment 1.1 as the benchmarking value. The testing procedure was similar to the method used in experiment 1.1. The circular buffer size was be increased and the tests re-run until the corruption was minimised or eliminated completely.

Method

1. Set Array Size to benchmark size
2. Change Circular Buffer Size
3. Run Both Write Programs and Read Program
4. Monitor Output to check if corruption
5. Repeat Steps 2-4 until corruption disappears

Experiment 1.3 - Circular Buffer Size with Multiple Writers

The third experiment for the shared memory was to confirm the safe recommended value with multiple writers, under heavy computer load and to ensure that the circular buffer values were being updated correctly.
The safe recommended value, for the circular buffer, was chosen to be above the value obtained in experiment 1.2. For this experiment a circular buffer of size 5 was chosen. This value should proved to be corruption free as it is higher than the obtained value. This value was then tested at high load on the computer which had multiple programs running at the same time. This allowed the experiment to show if increase CPU load and multiple programs writing to the shared memory resulted in corruption at the safe value.
A status function was written to allow the read program to read the current values of the read, write counters and the checksum and allowed it to be checked on screen for accuracy.
The corruption checking function iterated through the array size and checked that all values we the same.

Method

1. Set Array Size to benchmark size
2. Set Circular Buffer Size to "safe" size
3. Run Read Program and 10 write programs
4. Monitor Output to check if corrupted
5. Monitor Circular Buffer Status to ensure Updating happens

Experiment 2 - Network Node Timing

Experiment 2 was designed to test the end to end timing of the entire approach. This includes the network manager and the shared memory implementations. Timings were be able to compared with other network sharing protocols such as SCP. All platforms had their clocks synced before the testing using the built in linux NTP servers.

Different sizes OG's were also shared over the network, to check to see if there was an optimal size of data being sent. The OG data was time-stamped by the initial program then written to the shared memory to be shared memory. This change was propagated through the network to the next node. The next node checked the time stamp and display the timing it took for the packet to reach it.

Each data set consisted of twenty changes and the timings were averaged for that data set to minimise errors created by interference in the bandwidth. The experiment was repeated with different sizes of OG to determine the ideal size of sending data and allowing comparisons to be made against the same sizes being sent through another sharing program.

.

Chapter 6: Results

Experiment 1 - Shared Memory

Experiment 1.1 - Benchmarking Shared Memory

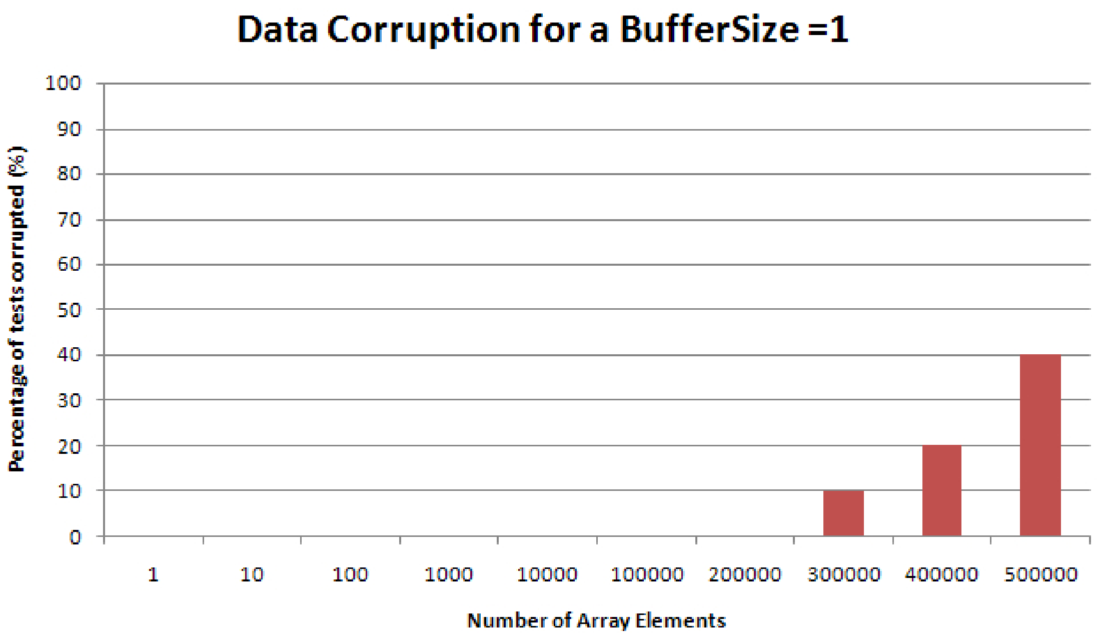


Figure 10: Results of Experiment 1.1 - Benchmark

From Figure 10, It can be seen that corruption began when the the array size had 300000 elements in size. It also shows in the graph that at 500000 the data corruption was only 40% of tests. Arrays with number of elements greater than 500000 caused segmentation faults for the program (memory out of bounds). The benchmark determined by the test thus is array size of 500000. This was then used in experiment 1.2.

Experiment 1.2 - Circular Buffer Size

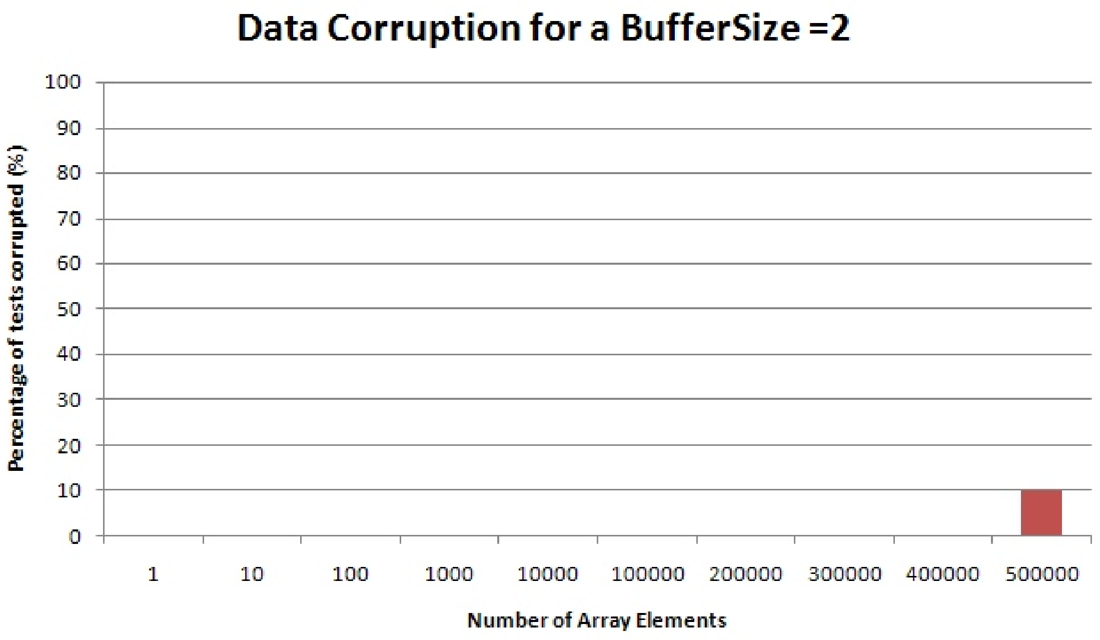


Figure 11: Results of Experiment 1.2 - Buffer Size = 2

The result from Figure 11 show that the implement of just a circular buffer of two reduced chance of the corruption to 10%.

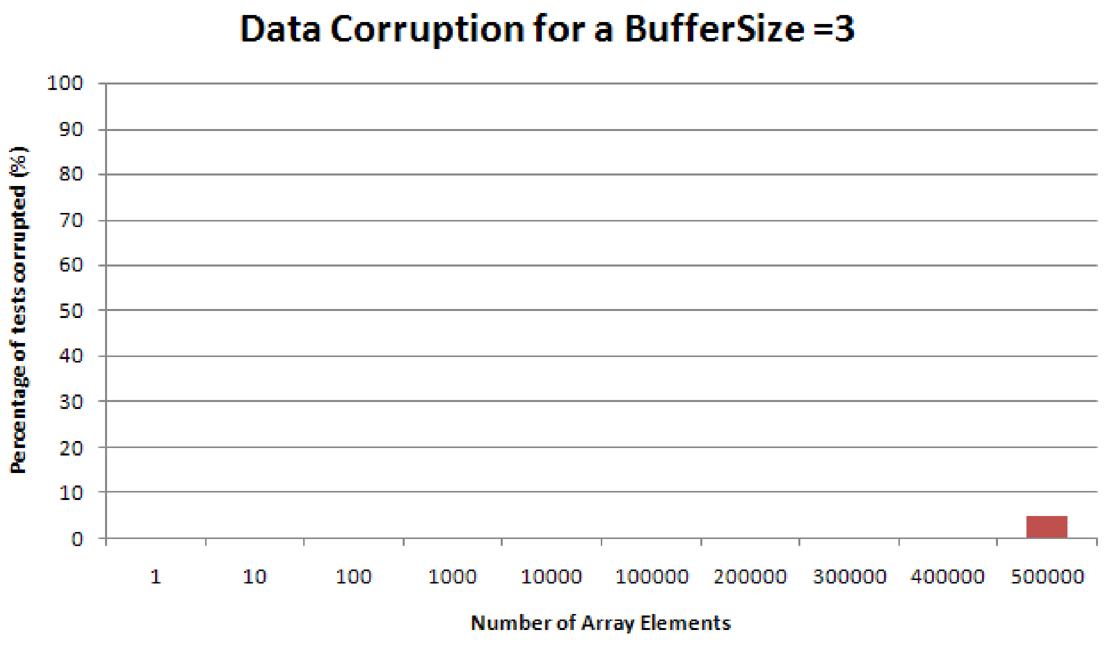


Figure 12: Results of Experiment 1.2 - Buffer Size = 3

The result from Figure 12 showed that a buffer size of 3 further reduced the chance of corruption. This test showed that only 5% of the tests showed data corruption.

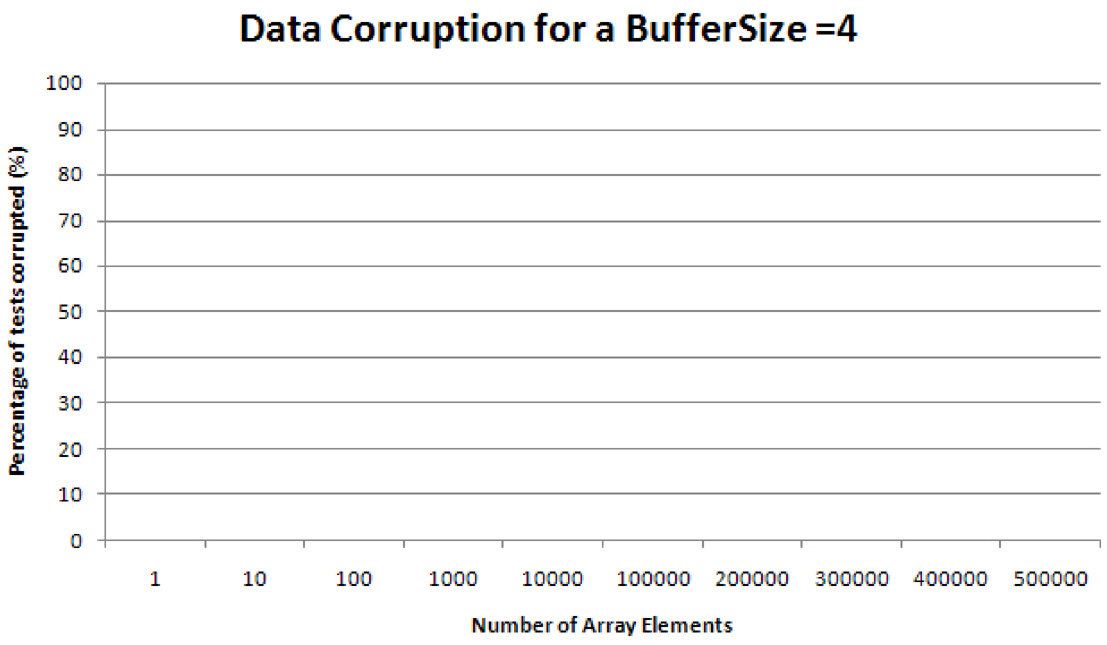


Figure 13: Results of Experiment 1.2 - Buffer Size = 4

Figure 13 results were obtained with a circular buffer of size 4. The graph shows that data corruption didn't happen for any of the test that were run.

Experiment 1.3 - Circular Buffer Size with Multiple Writers

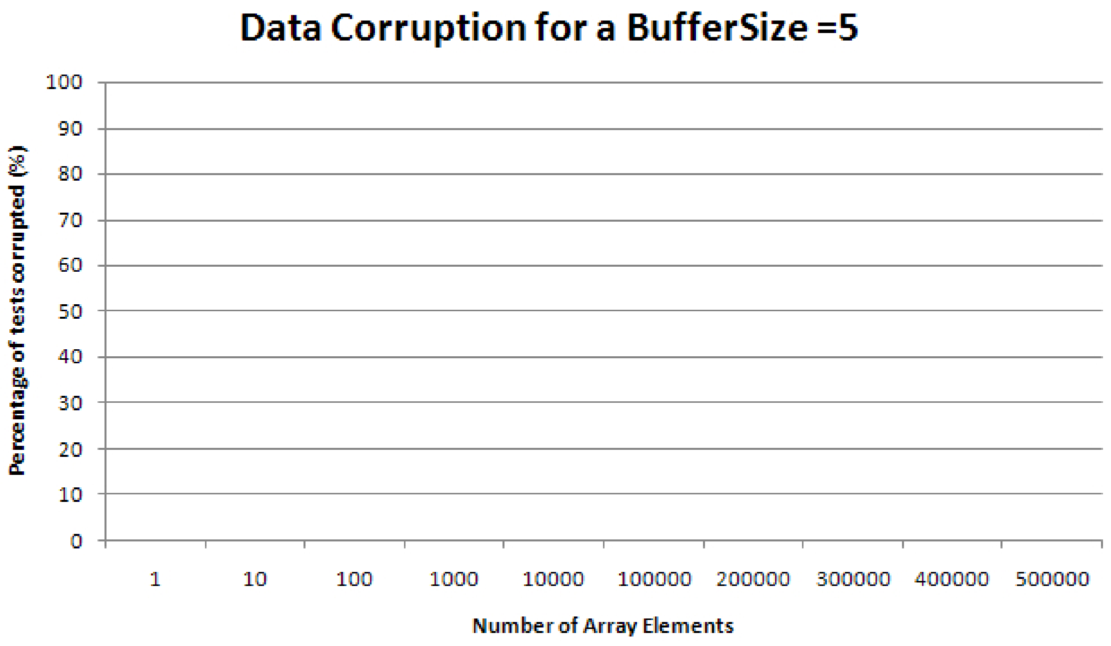
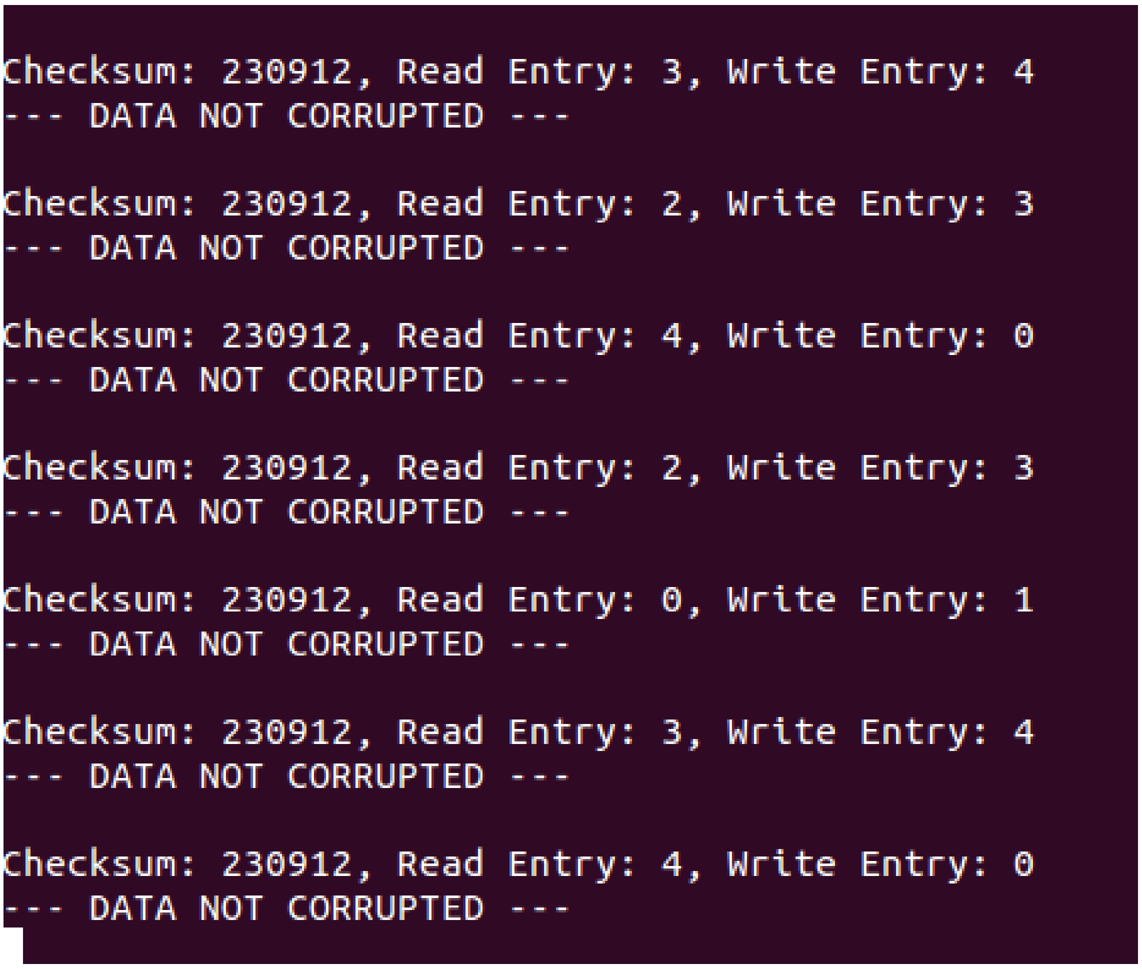


Figure 14: Results of Experiment 1.3 - Buffer Size = 5

The results of this test confirmed that for sizes greater than 4, the data still remains corruption free with more than two writing to the shared memory.



```
Checksum: 230912, Read Entry: 3, Write Entry: 4
--- DATA NOT CORRUPTED ---

Checksum: 230912, Read Entry: 2, Write Entry: 3
--- DATA NOT CORRUPTED ---

Checksum: 230912, Read Entry: 4, Write Entry: 0
--- DATA NOT CORRUPTED ---

Checksum: 230912, Read Entry: 2, Write Entry: 3
--- DATA NOT CORRUPTED ---

Checksum: 230912, Read Entry: 0, Write Entry: 1
--- DATA NOT CORRUPTED ---

Checksum: 230912, Read Entry: 3, Write Entry: 4
--- DATA NOT CORRUPTED ---

Checksum: 230912, Read Entry: 4, Write Entry: 0
--- DATA NOT CORRUPTED ---
```

Figure 15: Results of Experiment 1.3 - Terminal Output

Figure 15 shows the output from the status function for multiple reads. It shows that the internal values were updated correctly.

Experiment 2 - Network Node Timing

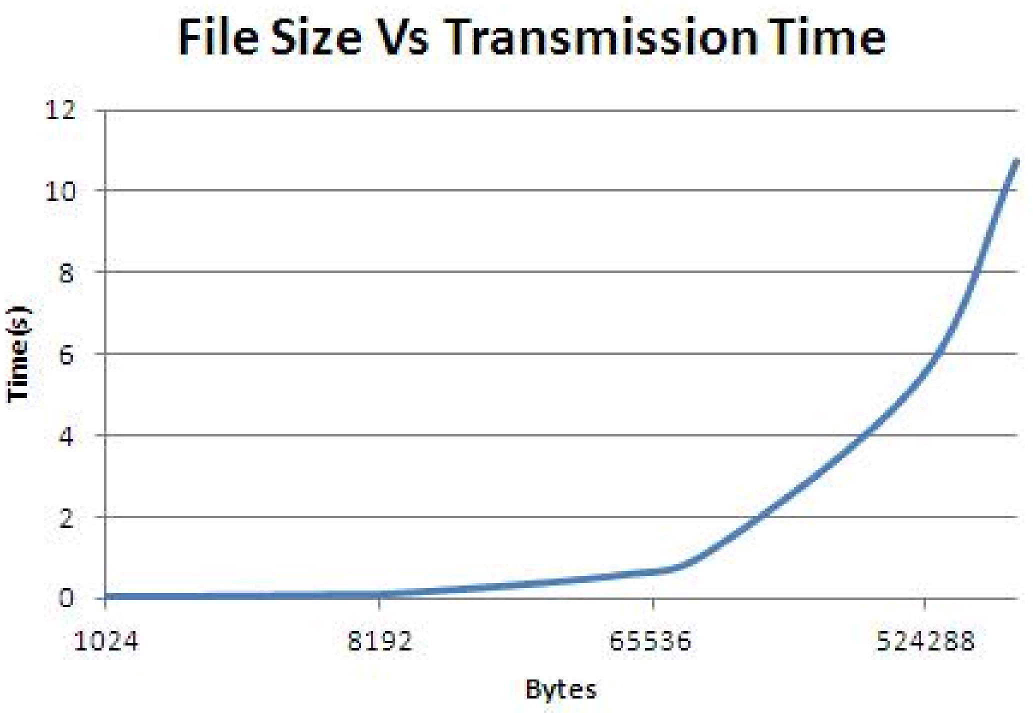


Figure 16: Results of Experiment 2 - Full Graph

Figure 16 shows that the increase in size had a exponential increase in timing. This was consistent with all sizes between 5KB and 10MB.

Sizes greater than 10MB, showed that the timing increase tended towards a linear trend.

The final timing of 50MB was removed to show greater resolution for the other timings. The timing for 10MB was around 107 seconds.

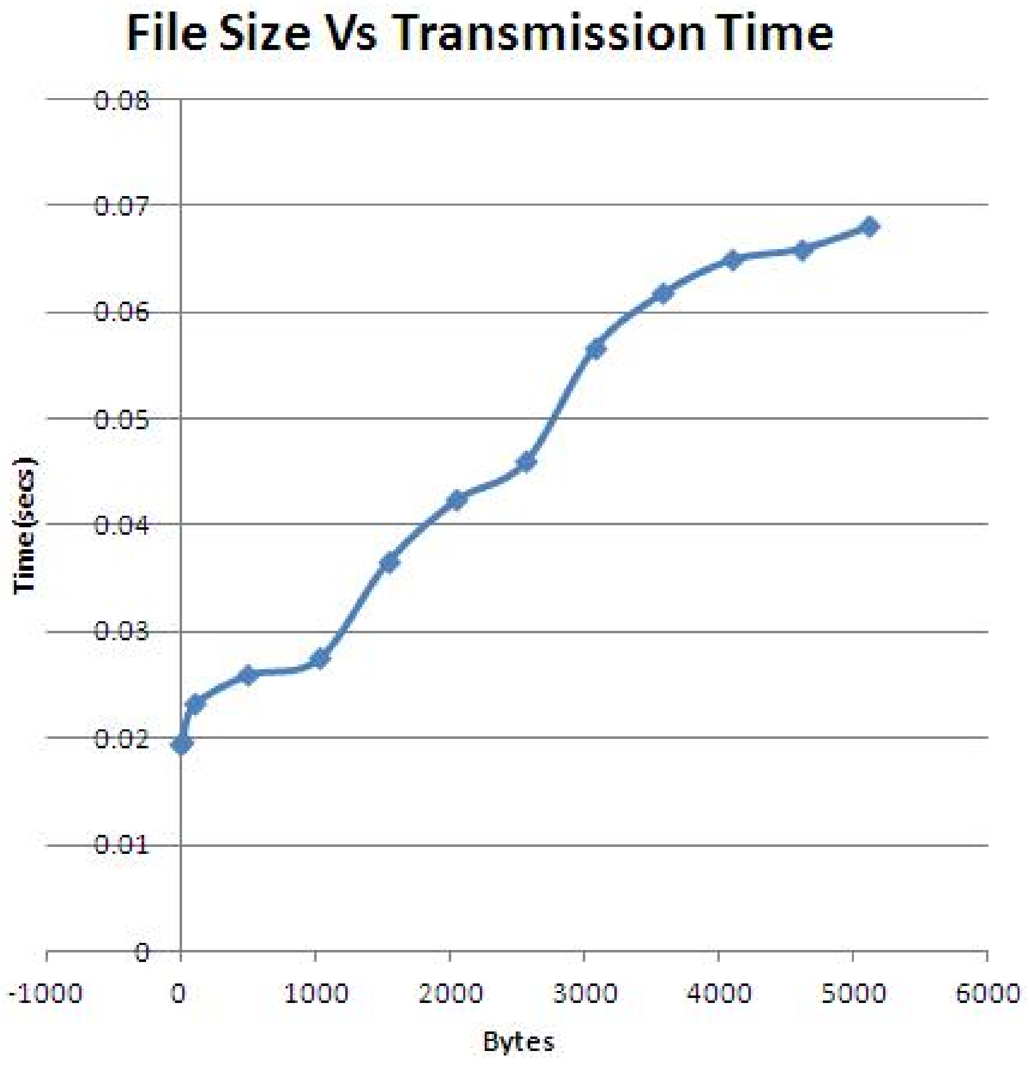


Figure 17: Results of Experiment 2 - Lower Values

Figure 17 is the section of Figure 16 between 0 and 5KB and has produced an interesting graph. Figure 17 shows that graph is a stepping function. Comparison of SCP was unable to be done due to SCP significant overhead in creating the secure authentication overhead.

Chapter 7: Discussion

The implementation showed the result that both the shared memory and the network manager performed the task required of them, however there were some interesting points in both which will be discussed below.

7.1 - Shared Memory

From Figure 10 it can see that the corruption doesnt happen for smaller sizes. This is due to the computer being able to read the shared memory faster than the program can overwrite it. The testing was performed on a machine of similar specs to the actual platform. Another test was performed on a computer that had greater specs than the platform and that test showed that there was no shared memory corruption. The graph also showed that only 40% of tests were corrupted. This would be mainly caused by the reading happening immediately before another write happens. It would be possible to time the platforms exactly to eliminate corruption however that would require a custom Operating System controlling all platform timings(as well as all other OS functions). The greatest number of elements that didnt cause a segmentation fault was 500000. This was due to an inbuilt restriction on the OS. The total memory usage increased when the circular buffer increased in size so it can be said that memory limit was not reached and thus the maximum size was set by the OS.
Figure 11 and Figure 12 showed that the corruption was minimised as the circular buffer size was increased. This shows that the circular buffer implementation was able to eliminate some of the corruption by giving the reading programs more

time before the data is overwritten. Figure 8 also shows that data corruption still exists with a circular buffer of size 3. The rate at which the circular buffer decreases wasnt a proportional amount. This means it is impossible to calculate, in advance, the size of the buffer that would allow for corruption free sharing for any particular array size.

Figure 13 shows that corruption didnt occur at all in the data with a circular buffer of size 4. All these values are affect by how much the computer is being used for other tasks. To allow for the difference between different computers, and the load on the computer, Experiment 1.3 was tested with a circular buffer of size 5 while having the computer perform multiple instances of the program. Figure 14 also proves that under a heavier load (multiple programs) the corruption was still eliminated.

In Experiment 1.3 Figure 15 shows that the values inside the shared memory are updating based on each write. The numbers missing in the middle show that all programs are using the same set of pointers and that two people wrote between the reads. It also shows that the read counter is always one previous to the write counter. The experiment was important to ensure that the shared memory was working properly and efficiently and that all programs were using the same set of counters and not either reinitialising each time a program starts or using their own set of counters.

These experimental results show that the data being shared through the shared memory is intelligently and efficiently being communicated between programs and each other or the network manager.

7.2 - Network Node Timing

Experiment 2 was conducted by averaging twenty times for the same file size. This was done to help reduce errors created by interference in the wireless range. In most of the tests outliers were found and removed from the average. To establish an outlier, the average was taken and any value more than 20% from the average was considered an outlier. The average was then recalculated without this value. This gave a more accurate value for normal timing for each file size. The graphs in the results are without outliers.

Figure 16 contains the values from 1KB up to 5MB. This was necessary as below 1KB was already represented on Figure 17 and only graphing up to 5MB gave a greater resolution for the other values. Above 10MB the graph tended to be more linear than exponential.

However above 10 MB the values were also much less reliable.

For a file size of 50MB more testing was required as more than half of the timings were over the threshold of an outlier. Most values were around 75% above or below average with some being over 200%. With the second set of testing the results were consistent with the first set of testing. This error was too great to include it in the graphed results.

Figure 16 shows the overall shape of the curve. As the file size increased the timing was increasing exponentially. This was expected as the data is increasing with a base of two and each increase will compound by a factor of two.

Figure 17 shows the timings between 1KB and 5KB. These timing show a very unique and complicated pattern. As can be seen from the Figure files sizes under 1KB suggested that the timing was going to plateau. This plateauing indicates a

limit in the networking packet size. The shape of the first part of the graph also shows that there is a significant overhead for small packets as the difference in the timings was less the closer the file size approached to 1KB.
From 1KB onwards, the graph showed a nonstandard oscillating wave around a 45 degree angle. The curve wasn't exactly the same for each period as the first period was larger than the other two. This was an unexpected result as the shape for all other segments were exponential. More research would need to be done to determine if the results were influenced by some interference.
The network manager also could be modified to change how intelligently the data is shared. The current method only incorporated connecting and sharing of the OG data.

Chapter 8: Future Work

Although this approach has been shown to be viable option to share the OG data there are ways that the efficiency could be increased. One of these ways is to increase the efficiency of network manager. This could be writing it on the link layer instead of the application layer.

This approach would eliminate the need for all layers on the networking structure to deal with this data. The link layer would also be able to pass on the data to the next node while simultaneously passing the data up the network stack resulting in decrease of the propagation time for communication. This would also allow for simplified upper layers of the network stack as all the sharing protocol would be handle by the link layer.

The main disadvantage of this approach is the protocol would need to include all the error checking and provide unique identification for each node (a possible implementation could be the MAC address currently used).

Another way that was mention in the discussions was to store non-critical data until the data size was of the optimal size for sending. This will ensure that the bandwidth is used to the greatest efficiency while also ensuring data reaches the other platforms.

The networking approach could also be tested using proper network simulation programs to determine the exact efficiency of the protocol.

Chapter 9: Conclusion

The results for the shared memory implementation showed that using a circular buffer, data corruption was eliminated using a buffer size of 5 or greater. The shared memory implementation was also shown to be an effective and intelligent protocol to communication between programs manipulating the OG and the network manager.

The network manager proved to be an intelligent way to communicate data across the wireless adhoc network. Although it wasn't the most efficient program, with further research and development it could be made both more efficient and incorporate different intelligent protocols.

Both of these implementations when combined, provide a framework protocol for sharing OG data between platforms over a wireless adhoc network. The protocol is able to be modified to incorporate different methods for intelligent sharing data.

References

[1] Marwa Abdel-Hady and Rabab Ward. A framework for evaluating video transmission over wireless ad hoc networks. *PACRIM*, pages 78–81, 2007.

[2] Tae-Yong Choi Byoung-Gi Jang and Ju-Jang Lee. Adaptive occupancy grid mapping with clusters. *Artificial Life Robotics*, pages 162–165, 2006.

[3] W.S. Ford and V.C. Hamacher. Hardware support for inter-process communication and processor sharing. pages 113–118.

[4] Richard Hay. 5 layer network structure. http://www.tamos.net/ rhay/overhead/ip-packet-overhead.htm, May 2004.

[5] Alessandro Rubini Jonathan Corbet, Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.

[6] J. Kurose and K. Ross. *Computer Networking - A Top-Down Approach Featuring the Internet*. Addition Wesley, 5th edition, 2010.

[7] Ahyoung Lee, Ilkyeun Ra, and HwaSung Kim. Performance study of ad hoc routing protocols with gossip-based approach.

[8] Ubuntu Canonical Ltd. Unix poll() man page. http://manpages.ubuntu.com/manpages/karmic/man2/poll.2.html, 2010.

[9] Ubuntu Canonical Ltd. Unix recv() man page. http://manpages.ubuntu.com/manpages/precise/man2/recv.2.html, 2010.

[10] N.Sumathi and Dr. Antony Selvadoss Thanamani. Pipelined backoff scheme for bandwidth measurement in qos enabled routing towards scalability for

manets. *A2CWiC*, pages 1–6, 2010.

[11] Max Pfingsthorn and Andreas Birk. Efficiently communicating map updates with the pose graph. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2519 – 2524, 2008.

[12] Dr Chandra S.R Putta, Dr K. Bhanu Prasad, Dilli Ravilla, Murali Nath R.S, and M.L. Ravi Chandra. Performance of ad hoc network routing protocols in ieee 802.11. *Conference on Computer and Communication Technology*, pages 371–376, 2010.

[13] Tien Pham Van. Proactive adhoc nodes for real-time video. *IEEE*, pages 1–5, 2006.