Simulation and Animation of a Mobile Ad Hoc Network

By
Nicholas Payne
May 2005

Nicholas.Payne@ncl.ac.uk

G602 Software Engineering

School of Computing Science
University of Newcastle Upon Tyne

Supervisors: Professor Isi Mitrani, Dr Neil Speirs

Abstract

A program which simulates a protocol family suitable in a Mobile Ad Hoc Network (MANET) is provided, with the ability to simulate and view an animation using a number of different scenarios defined by the end user. The aim is to achieve as near to complete network coverage as possible (that is, as close to 100%) while keeping the underlying protocol simple and message distribution overheads low. The implementation is somewhat simplified to ensure certain conditions are kept consistent and to allow meaningful comparisons to be made when using different parameters governing the program.

Declaration

I declare that this dissertation represents my own work except where otherwise stated.

Acknowledgements

Professor Isi Mitrani – for allowing me to take the project on in the first place, and giving me the help when I most needed it

Dr Neil Speirs – for being an extremely helpful and understanding tutor for the past three years

Christopher & Diana Payne – for being extremely helpful and understanding parents for the past twenty two years

Contents

List of Figures

Chapter 1 - Introduction

This chapter introduces the broad problem investigated in the project, background information, and the aims objectives of the project.

1.1 Overview

In the technological society of today the need for rapid and reliable communication is becoming ever more prominent. Whilst the combination of today's powerful computers and the Internet largely satisfy this demand these machines are still limited to being stationary devices relying on an underlying fixed infrastructure. With the advent of wireless technology and the increasing power of portable devices however, comes an exciting new field of investigation, that of mobile ad hoc networks. These are networks that are formed when necessary and disbanded when no longer needed. Crucially, they require no fixed infrastructure to communicate allowing for rapid on demand data exchange, suitable particularly for commercial meetings or disaster recovery situations or indeed any situation where a permanent infrastructure is not necessary and a waste of resources. Typical devices in such networks are mobile, battery operated, small units such as sub laptops or PDAs. These networks are widely referred to as ad hoc networks, and this paper expands upon these by implementing the idea of high mobility of the devices involved, forming a Mobile Ad Hoc Network.

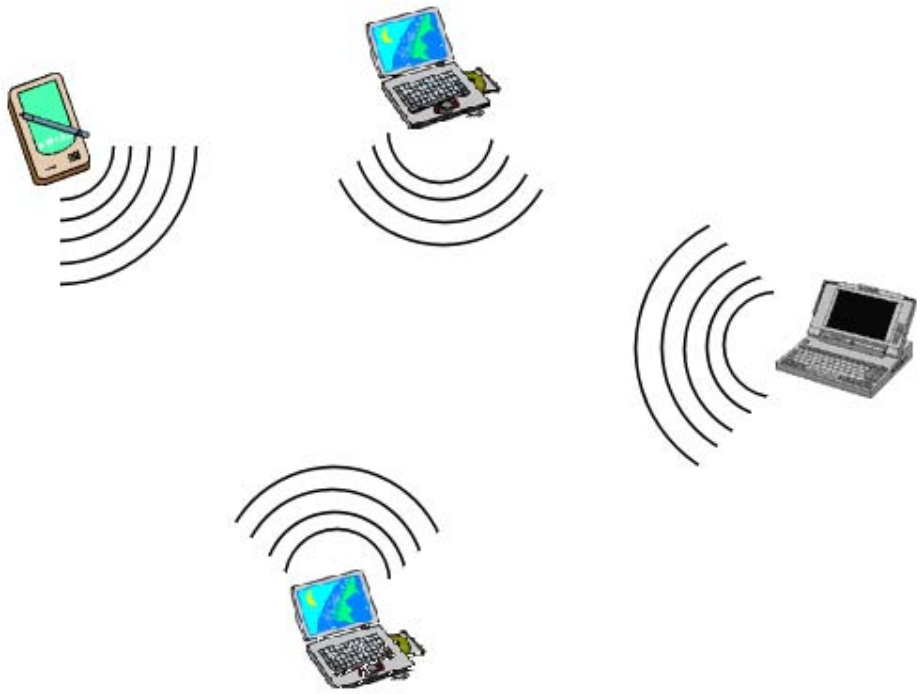


Figure 1.1 – a Mobile Ad Hoc Network

Whilst the underlying technology is not itself new, indeed it was first originated with the DARPA packet radio network during the 1970's project [5], with today's hugely advanced technology such networks are becoming a more viable alternative to the traditional fixed networks found in almost any networking situation around the world – mobile units, whilst still very limited compared to their desktop counterparts, are at a stage where mobile networking is a realistic possibility. However, a key factor in their success naturally depends on the efficiency of propagating a given message across the entire network of mobile devices (often referred to as *nodes* in this report) – this is essentially what the project aims to simulate. Of course, the term 'efficiency' is determined by a number of factors, primarily speed and reliability. Messages being lost in a Mobile Ad Hoc Network (hereafter referred to as a MANET) are a real concern due to the erratic and unpredictable topology of the network so appropriate measures must be implemented to deal with these. Whilst one propagation protocol might offer faster speeds than another, if it only works in certain conditions and performs dramatically worse when messages are lost then the other may offer better overall efficiency (dependant, of course, on how it deals with dropped messages).

This problem offers an exciting potential for dramatically changing standard network topologies, or at the very least expanding them into previously unexplored directions. Ever since networks were first popularised they have always consisted of an entirely static foundation; relatively large devices with fixed power sources communicating over physical links. Granted, with the increase of laptops and wireless networking this general assumption appears incorrect – however they must still connect to a fixed network of some kind. The beauty of a MANET is that it can be built 'on-the-fly', with every node serving to not only receive but pass on messages (essentially acting as routers in a network), eliminating the overheads of a fixed network when not in use and allowing for a huge increase in flexibility.

1.2 The State of the Art

An emerging popular example of MANET technology is the advent of Bluetooth, which is already finding its way into every day life, particularly in the world of mobile phones and Personal Digital Assistants. This, whilst a relatable illustration does not entirely encapsulate the typical model of a MANET as Bluetooth does not necessarily involve all devices working toward a common goal as is usually associated with MANETs (and indeed found in this project), although it could be argued that the recent rather quaint development of the BEDD software, which allows users of Bluetooth enabled phones to setup a profile and automatically download any 'matching' profiles within the receiver's radius, could be seen as just that [1].

The other current standard is the IEEE 802.11 [8] platform which allows network interface cards to communicate either with fixed access points (as in a normal network) or in an independent configuration whereby stations communicate directly with each other within a limited range – a so called "ad hoc" network.

The limitations of the above standards are that they are only single-hop technologies, meaning that devices must be within a small radius to communicate with one another. The development of multi-hop networks, whereby devices can act

as routers – not only originating but redirecting messages – is currently the subject of a great deal of focus in the world of mobile communications.

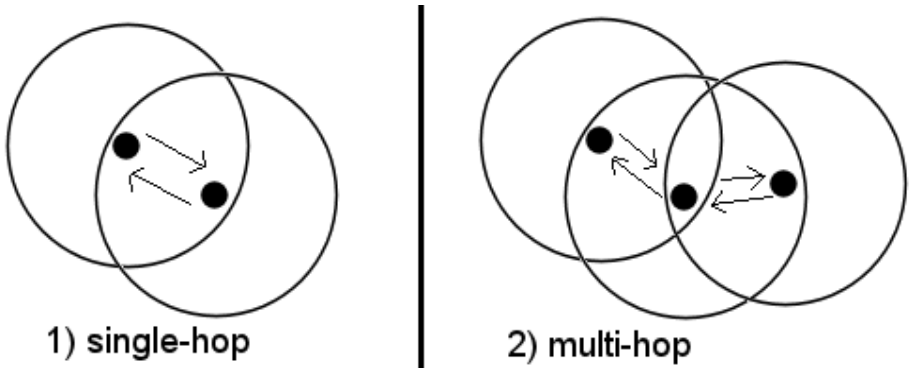


Figure 1.2 – single and multi-hop routing protocols

1.3 Project Aims

The hypothesis behind the project is to provide a simple range of protocols capable of delivering a message with as close to total network coverage (that is, every node in the network should ideally receive the message) as possible without compromising the simplicity of the protocols. The need to keep the simplicity low was to reflect the fact that the mobile devices of today (such as PDAs) have limited memory and power and that the bandwidth available in a network comprised of these devices is limited. A program was to be developed that simulated and (optionally) animated a simple ad hoc network in progress using these simple protocols. These objectives were largely achieved, although some areas could do with expansion (see 8.3 – Future Work).

The project was chosen due to offering the chance to develop an increased understanding of networking topologies whilst at the same time studying something not previously covered. It also offers the chance to develop programming and animation skills and explore the subject of performance evaluation in an attempt to learn how to evaluate and predict system behaviours. It provides a chance to research a truly cutting edge and exciting technology and one that will play a large role in the world of computers and networking in the future. In fact the scope is not simply limited to computing, and could be widely used in military and emergency operations. Indeed it is particularly applicable both; for example a military operation abroad is unlikely to have the time or resources to establish a fixed network to aid communications (at least on a smaller scale), which is where the speed of constructing (and deconstructing), and the ephemeral nature of a MANET would be highly suitable. In terms of an emergency the speed and lack of physical overheads would be crucial and saves a lot of time in establishing a network.

The development of the program was loosely based on the waterfall development model; such that once a particular stage was completed it was not returned to in any great detail. The advantages of such a model are that it allows a

focused amount of time to be dedicated to each stage and allows progress to be seen in large, conclusive stages and the linear nature of it provides a real sense of focus. Of course, the disadvantages are that once a stage is designated as being complete, it cannot be returned to for adaptation later on if need be. Of course, this would be more applicable in a larger scale development where large numbers of people would be working on the same project – with just one developer stages can be returned to and adapted without much fallout as nobody else is affected and therefore the whole 'team' knows what, how, and why a change has been made. Despite this flexibility, it would be preferable to follow the model as closely as possible, as it encourages good and thorough practices that will be needed in projects that do contain multiple people. As such, the four key stages outlined are:

- Research
- Design
- Implementation
- Testing

Chapter 2 introduces the background of MANETs and their accompanying technology and also sets this paper and its aims against existing work in the field. Chapters 3 and 4 comment on the design and implementation respectively of the program developed during the project, whilst Chapter 5 provides instructions on how to use it. Chapter 6 describes program testing, and Chapter 7 discusses experimental results generated by running the program once completed. Chapter 8 provides an evaluation and summary of the project on the whole.

Chapter 2 – Background

This chapter introduces information about Mobile Ad Hoc Networking technology and outlines the scope of this thesis, with regards to the level of implementation in the program to be developed for it.

2.1 Overview

This paper does not attempt to expand upon or implement any real world applications of current appropriate MANET technologies but instead attempts to provide an abstract view of the problems and nature of such a network. As such the project itself can still be related to other works in the field. According to the taxonomy defined in *The Handbook of Wireless Ad Hoc Networks* [6] the scale of this abstracted network fits in to that of a LAN, as it remains within metres rather than kilometres or greater.

Current work in the area of ad hoc technology is advancing rapidly and is at a level far more sophisticated than that simulated in this project. Given that it is a field at the forefront of the development effort, comprehensively implementing such a relatively young and changing technology would be beyond the scope of this project.

2.2 Mobile Ad Hoc Networks

"*An autonomous collection of mobile users that communicate over relatively bandwidth constrained wireless links***"** [23]

2.2.1 MANET Description

Whilst the project will not simulate in depth the low level functions of a MANET, it is important to have a strong background on the subject. This helps examine throughout the project just how faithful the simulation and eventual end product actually is. It is not appropriate to simply take advantage of the abstract nature of the project and assume that any discrepancies are because of this fact. With that in mind, an outline of what exactly a MANET is and the state of the technology follows. In the context of this project, a MANET can be characterized as having certain attributes as follows, which whilst not necessarily totally accurate in every application of such a network, are valid in the type of MANET examined in this project (for example nodes in this project are considered to be of very limited power and memory).

- Unpredictable topology: the fact that the network is mobile means its topology can never be defined in a static context; a 'snapshot' of the network at any given moment in time will not necessarily be valid at any time before or after.

- Limited power: nodes have no fixed power source and thus will only be able to function for a certain amount of time before needing to be recharged in someway, i.e. charging or changing the battery.
- Limited bandwidth: Even the most advanced wireless technology today is still inferior when compared a state of the art physical counterpart. Given the point above, the wireless technology on board the nodes in a MANET is simplified even further which restricts the communication available within the network.

Nodes within a MANET consist of both transmitters and receivers due to the fact they accept and pass on data within the network. However, the bandwidth available in such a network (indeed in any wireless network) is dramatically lower than available through a physical cable, therefore it makes sense to keep the message content as simple as possible, particularly as this cuts down on the processing each device needs to perform.

The state of the technology today is at such a level where the concepts of ad hoc networks are being expanded into areas where they could be of real use. One application of the technology that came up time and time and again during research was that of the emergency services, which depend on speed. A common scenario given is that of police, ambulance and fire services all turning up at an accident scene and instantly being able to communicate seamlessly with one another using a MANET. The transparent nature and relative lack of set-up time (particularly when compared to establishing a static network) would provide a crucial communication aid to the services.

2.2.2 Design Factors

The crux of the problem comes in the form of the mobile devices themselves. Whilst they do offer numerous advantages to desktop systems, they have many inherent flaws, particularly when concerned with the paradigm of networking. The addition of mobility is something of a double-edged sword; whilst it increases network flexibility, it has an adverse affect on network stability. Within a static network certain aspects are assumed that somewhat simplify the topology, notably that the devices in the network are located in the same place – at least whilst they are communicating with the network. They also have the added advantage a great deal more memory and, in terms of energy, almost unlimited power. In terms of processing power they too have a huge advantage over mobile units. These factors combine together to make the problem more interesting as they provide an opportunity to experiment with many different variables in an attempt to find the ‘optimum’ network setup.

By the time the project was completed, it was hoped that a range of different propagation protocols will be implemented in a realistic, though basic, simulation of a MANET. Various aspects of the network were to be alterable by the user, such as the number of nodes in the network, node broadcast range, speed, power, message reliability, and so on. The other distinct element to the project is of course that of animation. This was to provide a visual bridge between the mechanics of the program and an overview of the topology in question, and give the user more gratifying, instant

feedback. Furthermore, it would allow the user to see how the network is communicating as it will offer an abstracted display of the simulation, showing nodes moving and messages being sent across the network. It was intended that the animation side of the project would be just as heavily focused upon as the simulation, as, even though the primary aim of the project is not necessarily user satisfaction, it would be better to make the project something that could truly, and intuitively, be interacted with. The simulation side of the project should certainly be robust enough to analyse effectively the performance of the algorithm and generate credible results as these would be thoroughly analyzed and will be used when making conclusions about suitable protocols.

2.3 Background Research

Much of the initial research required for this project was carried out by investigating existing MANET protocols on the Internet as it is a fairly new technology and not many authoritative books exist on the matter. A paper was kindly provided by the Newcastle University School of Computing Science, 'High Coverage Broadcasting For Mobile Ad-hoc Networks' [7], which also helped to formulate ideas for the project (any references to the paper have been clearly marked as such) and indeed the family of protocols examined and simulated in this project is modelled on those expressed in that paper.

It is important to stress that the real focus of the project should not be lost in the complexities of a real-world MANET – the key is accurate simulation and animation of an abstracted network. Nevertheless, it is important to gain an important understanding into the model that is to be represented, however abstract that representation may be. To the end of simulation, research was carried out into the realm of performance evaluation, as this is a crucial technique when trying to analyse effectively the results generated by the simulation aspect of the program. The concept of performance evaluation is to examine exactly how a given system performs, and by simulating a system we hope to imitate as best as possible a real world counterpart.

As the project focuses on merely simulating a MANET rather than actually creating one (the logistics of which would be well outside the scope of this paper), the technologies used in the actual program are nothing out of the ordinary; in fact the project is entirely coded in Java. The actual technology simulated is of course, far more advanced but again much of this can be ignored, or at least scaled down to simplify the simulation. Of course, the basic rules of MANET technology but the low level details are simplified.

In *Simulation Modeling & Analysis* [11], the 'nature of simulation' is defined as "techniques for using computers to imitate, or *simulate*, the operations of various kinds of real-world facilities or processes". It goes on to define a *system* as "the facility or process of interest" and a *model* as "a set of assumptions about how it works ... which usually take the form of mathematical or logical relationships" [12]. These theories will be used as the basis for the simulation aspect of the project.

2.4 Related Work

There are numerous network simulation tools available, some designed for general purpose simulation and others more focused on network simulation. *OMNeT++* [17] is an example of a general purpose simulation environment which is capable of a broad range of simulations. However, it is primarily aimed at network simulation and hence has a range of frameworks for different network communication protocols, including a Mobility Framework [15] capable of simulating wireless and mobile networks. Unfortunately it was not possible to get the program working with the Mobility Framework extensions, though a number of standard simulations of routing are supplied with the program, which give a good indication of its capabilities.

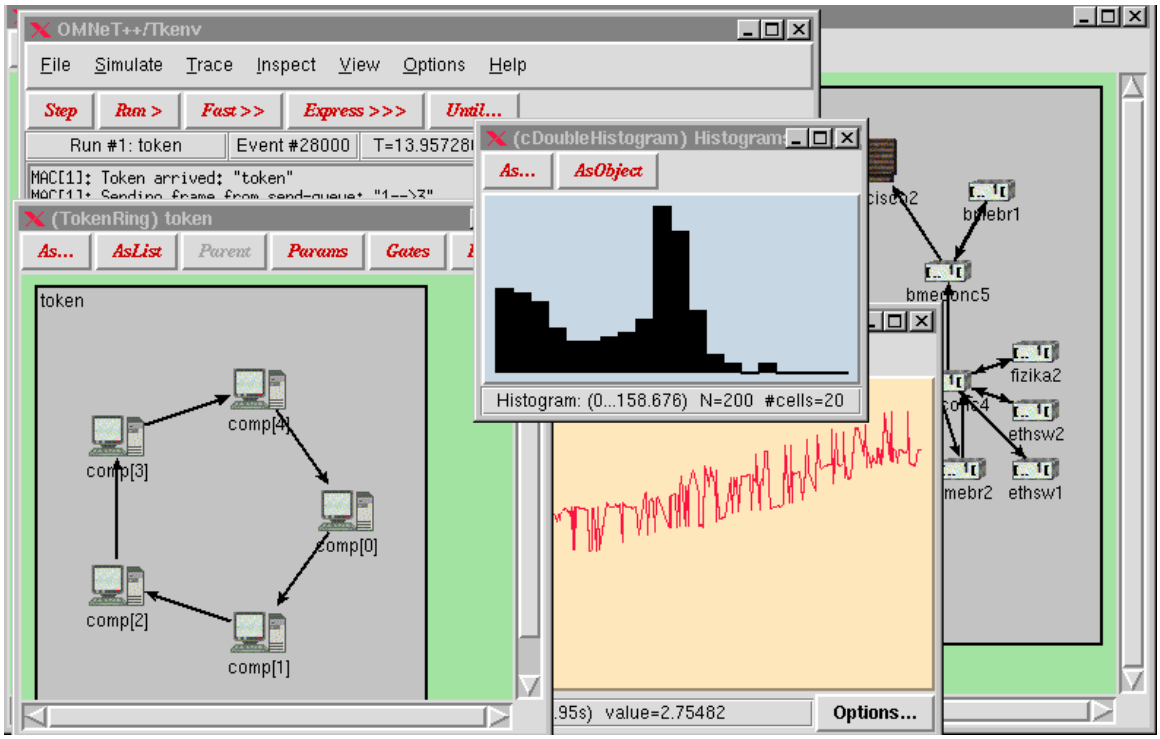


Figure 2.1 – OMNeT++ in action (taken from the OMNeT Community Site)

Another widely used simulator is *The Network Simulator – ns-2* [16], which features an extensive list of simulated communication protocols including Bluetooth and various 802.11 implementations. Initially developed as a variant of the REAL network simulator in 1989, it is used in Ilyas for performance evaluation [21] and cited in *Mobile Ad Hoc Networking* (ed. Basagni, S.) as being the "most popular tool used by both the wireless and wired communities" [2]. Unfortunately it is not as straightforward to use as OMNeT++ and no tests could be performed to evaluate this assertion.

There are of course many other network simulators available, though a draw back to a large number of the more sophisticated ones, such as the *OPNET* modeller [18] (see FIGURE X, right) is that they are commercial products which require purchasing. OPNET does indeed appear to be a staggeringly sophisticated simulation tool, with a dedicated wireless module, and with clients ranging from the US Marine Corps to NASA its credibility is unquestionably of a very high quality.

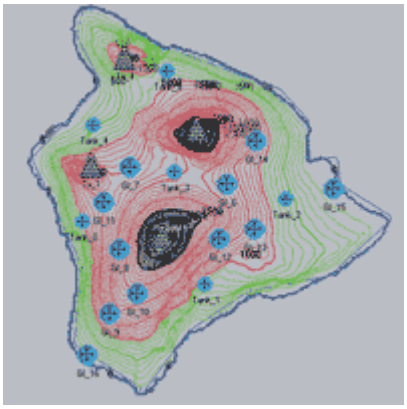


Figure 2.2 – OPNET simulation tool (taken from OPNET website)

2.5 Programming Language Research

Along with researching information on the subject to be modelled, an appropriate language to create it in was also necessary. Experience with predominantly C++ and Java made the choice fairly narrow, and although the option of using VDM-SL was briefly considered for its formal modelling capabilities it was soon discarded due to unfamiliarity and a lack of animation capability. A brief overview of the former two languages follows assessing their merits.

2.5.1 C++

It is widely acknowledged that C++ has long been the industry standard when it comes to programming all manner of applications and games. An extension of C (adding primarily the concept of Object Oriented programming), C++ is used worldwide in a vast number of programs, games, operating systems, and so on. It is fast and very widely supported, with many extensions available to increase ease of development in specific areas (for example networking and graphics). Compared directly with Java, the most obvious advantage is speed. C++ programs are compiled directly into machine code – which is executed by the machine directly and subsequently has access to its memory and peripherals directly. It is therefore possible to write extremely fast specific code for certain machines, the trade off being they will only work for the operating system they are designed for. Whilst compilers are available for the main three operating systems platforms of today (Windows, Mac OS and *nix), it is not possible to simply recompile the source of a given program on any platform using a compiler designed for it, due to the fact that in a typical program many operating system specific functions and APIs are used.

One big attraction to the language is the presence of DirectX – an SDK developed over the past few years which makes utilising advanced hardware features fairly straightforward for the average C++ programmer. This allows for incredibly quick animation, sound and input processing – which was a strong consideration when evaluating the animation requirements of the project.

2.5.2 Java

"*Java technology is a portfolio of products that are based on the power of networks and the idea that the same software should run on many different kinds of systems and devices"* [9]

Syntactically very similar to C++ (though of course with substantial though less immediately apparent differences), Java is becoming increasingly used particularly in the field of web based development. The ethos of platform independence is a very attractive one as it provides an instantly larger target audience with no extra programming effort involved. It also has the advantage of being very well documented with a focused, centralised support provided by its creators, Sun Microsystems. It has an incredibly large array of standard classes and methods which cater for the vast majority of basic programming needs. The ease of use and manipulation of the 'String' class is taken for granted in Java, when in C++ it has only recently been satisfactorily implemented and still severely lacks the range of functionality provided in Java. Similarly, the focus on networkability in Java makes basic network programming a fairly trivial affair, whereas in C++ it is very complex (and not part of the standard library set). It also allows for rapid development of Graphical User Interfaces due again to thorough inbuilt support. Generally speaking, it is a lot quicker and easier to code and distribute a basic application in Java than C++.

However, this simplicity and portability comes at a natural cost; execution speed. Java programs are not compiled into specific machine code; rather they are compiled into .class files, which when run are interpreted by the Java Virtual Machine running on a specific machine. This denies the programmer access to any machine specific abilities, and because of this abstraction between the code and the actual machine the execution of the program is very slow. It also lacks any fast way of doing 2D or 3D graphics, which made the decision over the language that much harder to make.

2.5.3 Language Conclusions

Whilst both languages have their obvious merits, it was decided that the speed of the animation required was not of fundamental importance (running slowly would not adversely affect the experience) where as ease of distribution and simplicity to program were. This led to the choice of Java for the program. The animation speed, whilst a down side to the program, is not critical, is outweighed by the ease and speed of development offered by the Java platform.

Chapter 3 - Design

This chapter introduces an overview of the design concepts and the problem attempted in the project. It also introduces the design specifics of the most important aspects of the program.

3.1 Overview

The project will implement a number of existing multicast protocols as presented in 'High Coverage Broadcasting for Mobile Ad-hoc Networks' [7]. It must be noted here that the premise of the project is to evaluate existing protocols rather than to design new ones, so the aim is to simply replicate a range of protocols as faithfully as possible within the model. It is also necessary to state some assumptions about the level of abstraction to be used in the project. Although initially it was intended for different nodes to have different values (for example, differing power levels, differing broadcast ranges and so on) initial research suggested that this was beyond the scope of the project and detracted from the probabilistic element necessary for evaluating the project. Instead, it is assumed that all nodes share the same inherent properties such as speed and broadcast range – though these can be changed, they must remain the same for all nodes. Furthermore, the nodes are assumed to exist within a given finite space and all interact exactly like one another, i.e. they are part of a homogeneous group.

When it came to designing the actual program for the project, a number of elements needed to be considered. The target audience and target platforms were of primary concern as they both shape the usability and language of the project. Whilst Java may not present the easiest platform to begin running a program, particularly to users familiar with the ease of double clicking a windows executable file, once running it could be made just as familiar as any Win32 application. As mentioned, platform independence was also an important consideration, hence why Java was in the end the language of choice.

Once the development language had been established, creation of a design outline for the program could commence. Given that Java is an Object Oriented programming language there was at least one initial obvious class to create; that of the nodes themselves. It was logical to examine what characteristics define any given node, and then a basic template class could be created based on these. As such, an initial thin template for the class can be drawn as follows:

```
public class Node
{
	// member variables

	int x;
	int y;  // coordinates

	int vx;
	int vy; // velocity

	Boolean msg;  // true = message received

	int broadcastRadius;

	void broadcastMsg();
}
```

An instance of the node class could then be instantiated by calling the constructor with a number of parameters:

```
	Node n = new Node(x, y, false, 10)
```

Where the parameters are the coordinates, message status, and broadcast radius (note that some parameters that would need to be set have been omitted).

This template, whilst leaving out a large amount of actual code, is very similar to the shape of the Node class that is used in the final program (see Appendix: Code Listings to examine the Node source code in detail).

The animation is represented in entirely 2D as there is no benefit to a 3D representation, which would only serve to add computational and development overheads. The terrain in which the nodes move is represented on an entirely static screen with the nodes scaled to fit differing terrain sizes, that is to say the terrain boundaries always appear the same and the nodes are scaled up or down depending on what distance those boundaries represent. The nodes themselves are represented by small points surrounded by a circle representing the broadcast range, and are drawn in different colours representing their current status (message received / not yet received, and so on). The actual ‘look’ of the program is fairly simple, but hopefully the added levels of animation and feedback from it will provide interest.

3.2 The model

The model itself will be modelled as a *discrete time* simulator, defined in Basagni “the modeling technique in which changes to the state of the model can occur only at countable points in time” [3]. This is suitable for the model because each run can be broken down into discrete ‘frames’. At this point it is useful to define the use of the term ‘run’; used in this hereafter in this paper to mean the complete simulation of a network consisting of a specified number of nodes in a specified area size. By ‘complete’ it is meant that the simulation terminates by propagating the message to as many nodes in the network as possible (potentially all of them). At any given frame there can be a number of events which are said to occur instantaneously – for example

a node moving, encountering another, or receiving a message. A template for the execution sequence of the simulation can be outlined as follows:

```
While (running)
      do
            moveNodes()
            checkEncounters()
            broadcast()

            /* animate() if in desired mode */
      od
```

The model implemented involves a number of nodes *n* which are distributed randomly in a given square area. The terrain itself is assumed to be featureless, that is there are no obstacles which get in the way of the nodes apart from the terrain edges. The nodes move according to a random mobility pattern, choosing a random initial angle (within 0-360 degrees), moving for a random amount of time and then pausing for a short period (again, both 'random' within upper and lower limits).

In order to signal their presence to other nodes in the network, each node intermittently transmits a small 'hello' message (though in practice this interval is so small that this message is transmitted once every frame, effectively the same as the message always being present). This message is ignored as being part of the transport layer of the protocol, and thus provides one of a number of assumed simplifications employed in the model:

- 'Hello' messages do not count as network traffic; they merely indicate a node's presence.
- Nodes have infinite power – that is they never run out of energy, hence never lose mobility or broadcast / receive radius.
- When a message is broadcast, it never fails – that is it always reaches any other node in the sender's neighbourhood.
- All node radius sizes must be the same as one another.
- There is only one unique 'message' propagated throughout the network – thus eliminating the need for nodes to store individual message counters or identifiers.
- There is no concept of network interference between nodes communicating over the same area.
- Nodes do not collide with one another – they can be assumed to be small enough as to never hit or else narrowly avoid each other.

A neighbourhood is defined as the number of nodes that a given node *i* can hear at any one time.

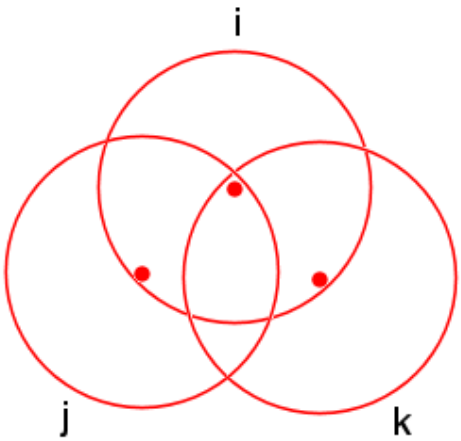


Figure 3.1 – Network Neighbourhood illustration

In the above illustration, node *i*'s neighbourhood consists of {*j*, *k*}. As a result of the restriction on radii sizes posed above, it is clear that in the above each node contains the two others. This goes someway to proving why the radius simplification helps when propagating a message; clearly if for any given node *x* to encounter another node *y*, the reverse encounter also holds. An encounter can be defined simply as any node *i* entering another node *j*'s neighbourhood. Due to the mobile nature of the devices a given node's neighbourhood range from being empty to containing every other node bar itself (although of course the likelihood of this is very slim).

3.2.2 Protocol Specification

The protocol family used in the model deserves some expansion, as it is the crux of how the nodes behave during a particular run. This family is an implementation of the one proposed in [7].

The key factors which govern them are as follows:

- When receiving the message for the first time, a node checks its neighbourhood. If any node other than the sender exists, it transmits the message and increments a counter, *e*.
- At every encounter, if a node has the message, it transmits it, providing $e < l$, where *l* is the limit of the number of times a message can be transmitted by any one node.
- When $e = l$ the node can no longer transmit the message.

It should also be noted that nodes do not keep track of whom they have transmitted the message to – there are no acknowledgements to keep traffic overheads low. Therefore if a node leaves and then re-enters another node *i*'s neighbourhood, it is treated as a new encounter. The same logically applies to nodes that may already have the message which enter *i*'s neighbourhood. Thus the protocol sacrifices potential efficiency to compromise on memory and bandwidth requirements.

As mentioned, there is a given limit *l* which defines how many times any given node *i* may transmit message *m*. The differing protocols considered in this project do not change these rules, merely the limit *l* and consequently the overall behaviour of the network. These protocols are referred to in this paper a family of *l*-propagation protocols.

Take, for example, the minimum encounter limit of 1. Using this protocol the simulation would be expected to perform similar to a standard flooding protocol, the principle difference coming with the fact that nodes do not broadcast the message after a random amount of time, but at their first new encounter. This will still give a very similar performance and network coverage to flooding as node encounters could be said to occur at random points in time anyway. It should be expected that a 1-propagation protocol would perform very poorly in a MANET, as in order to achieve total network coverage every new encounter *e* would have to be between the last node to receive the message and one which has not already received it. As *l* increases it would be expected that total network coverage does so as well, though at some point the limit will become too high and cause unnecessary overheads. These values of *l* are experimented with and discussed in the Experimental Results section (Chapter 7).

3.2.3 Node and World Specification

The scale of the model implemented in the program was implemented as close to a real world scenario as possible. Due to the hypothetical nature of much MANET documentation, finding actual solid values did not prove straightforward. Loose guidance was taken from Basagni [13] and Ilyas [20], which has several references throughout it to particular test values used when simulating different environments. Common sense was also employed, for example when determining the unit of scale to be used – as no MANET using today's technology is likely to expand beyond metres as opposed to kilometres. As such, the available world sizes to run the program are:

- 125x125 metres
- 250x250 metres
- 500x500 metres
- 1000x1000 metres

Node radii can vary from 15 to 50 metres (the default is 30m) – these limits are hard coded into the program. Nodes can move at between 5 and 25m/sec (the default is 15m/sec), and their mobility is governed by a pattern very similar to the 'Random Mobility Model' as described in Basagni [4].

3.3 Program Design

3.3.1 Graphical User Interface

In order to implement user flexibility a Graphical User Interface (GUI) is included with the project, in an effort to make running varying tests a more straightforward approach. Whilst the program is not intended to be a learning device there is no harm in making it more accessible to a wider user base. With the added addition of a GUI and inclusion of animation the finished product will hopefully appeal to people who have not perhaps even heard of a MANET before and held no previous interest in the technology. Simply running a series of text based batch commands and producing an output file of network statistics is unlikely to attract the attentions of all but the keenest user.

Naturally, a GUI would be rather redundant if it did not offer a wide variety of controls or if they had little affect on the program. The intention was to make it as comprehensive as possible so the user can truly interact with the program and have a real affect on the outcomes, results and graphs generated. Obvious choices include:

- number of nodes in the network
- terrain size
- protocol to be used in the program

The GUI takes advantage of Java's Swing components [22] which make for straightforward and rapid development of simple user interfaces without the need for any low level programming. All manner of classes are supplied which cater for every type of interface (buttons, text boxes and so on) used in the program.

3.3.2 Program Outline

Although detail is reserved until later in this paper (see Chapter 4 – Implementation) it is worthwhile to provide an outline of the design of the program in this section. As previously mentioned, a Node class will exist to cater for the representation of the mobile devices simulated in the network. It is logical that this class will also contain the functionality required for sending and receiving messages, as it is at this level in the real world that such communication would occur – that is the nodes themselves govern the communication of messages.

An abstract concept of the 'world' in which the nodes exist is also broadly defined, though not as a specific class. There are two distinct modes of operation in the program, identified as Animation and Batch modes (see Chapter 5 – User Manual, for more details). The program can only operate in one mode at any given time, and the world is represented by whichever mode the program is in. It is useful to use the world abstraction as it is here where the movement and governing of all the nodes as a group takes place. That is, whilst one Node object knows only of its existence, the world contains all of the nodes as a whole. It is in this world that the nodes all move and interact sequentially. The following is a very abstracted view of this world concept:

```
world_tick()
{
	for all nodes
	{
		this_node.move();
		this_node.checkEncounters ( other_nodes );

	}

	updateInterface();

}
```

Thus, once per frame, the world ‘ticks’ (in the sense of time ticking) and moves all the nodes accordingly then makes each node check for any new encounters (there is in fact a lot of logic contained within the encounter checking – explained in more detail in Chapter 4 - Implementation).

3.3.3 Functional Requirements

In terms of functional requirements a few are clear, such as the system must simulate as faithfully as possible the scenario it is given. It also is necessary to set out some important requirements that the eventual system must meet in order to give the project an increased level of coherence; a check list is provided below:

The animation must run smoothly – in order for the animation side of the project to merit implementation, it must provide a satisfactory experience for the user.

The GUI must be logical – whilst the program will not stand or fall based solely on the ease of use, personal standards dictate that it must be accessible for all. A neat and logically designed GUI is crucial to achieving this.

The program must perform as expected – a crucial part of the program will be, naturally, that it performs properly. If the user does something unexpected or some outside influence acts upon the program it must react accordingly and graciously. There must be no errors in the code that affect how the system performs or affect the output of the simulation – as this could easily lead to erroneous conclusions based on inaccurate graph data.

Chapter 4 - Implementation

This chapter introduces how the key ideas presented in the design of the project were implemented in the program.

4.1 Overview

When it came to actually implementing the ideas researched earlier in the project, Java was eventually decided upon as the language of choice. The factors determining this were primarily:

- Portability – the ability to develop and eventually distribute the program on a wide range of Operating Systems was a big draw when choosing Java. The fact that the SDK is also compact and readily available on the Internet meant that development could be moved to an entirely new computer (as did in fact happen midway through development) and resumed without much difficulty at all. Although not particularly relevant in the scope of a small scale project such as this, having the ability to distribute the program to a number of Operating Systems is undeniably a bonus.

- Support – given that Java is developed by one company, there is one universal source for the API specifications and official tutorials – something seriously lacking with the author's experience involving C++ (the only realistic rival when language choice was made).

- Simplicity – although similar to C++ there are several aspects of the Java language which make it a simpler more attractive choice. The added packages such as the Swing components, which made designing and implementing the GUI straightforward, were also an important consideration.

That is not to say, however, that the implementation process was entirely straightforward. One of the main initial causes for concern was the simplicity, or lack thereof, of achieving animation in Java whilst presenting a GUI at the same time, and having the two cooperate, i.e. not slow down the GUI responsiveness by using too much processing power to handle the animation or vice versa. This did indeed prove to be a time consuming part of the implementation process and severely hindered the initial development of the program due to a reluctance to program the underlying code without being able to visualise the proceedings to ensure the nodes were moving correctly and interacting properly. Most of the information gathered suggested that using threads would be the best way to achieve reasonable performance, and whilst eventually animation was achieved using an instance of the Timer class, the animation model is loosely built upon the basic ideas introduced in 'Double Buffering In A Frame' as presented by Frederic Patin [19].

The program was written using the latest Java SDK (1.5 at the time of writing), built using a free Integrated Development Environment (IDE) called JCreator [10]. A number of other IDEs were investigated including BlueJ and NetBeans, and whilst all had their merits, JCreator offered the most immediate familiarity due to the author's

experience with Microsoft Visual C++, as they both share very similar interfaces and provide a similar range of options. The program uses no specific IDE code however; hence it can be edited and compiled under any other IDE with ease.

Once a satisfactory template involving a simple GUI and animation panel was in place, much of the early coding of the program went smoothly and indeed a vaguely recognizable ancestor of the finished product was up and running within a couple of days. However, as progress continued, decisions about what was realistically achievable came to the fore.

4.2 Implementation Details

The following section attempts to provide a high-level overview of the workings of the program; that is, how the simulation model is implemented. However, it is impossible to accurately convey how it works without involving some specific references to the code. Where presented, these are explained as fully as possible.

4.2.1 Program Structure

The program itself comprises a number of Java classes governing the various aspects of it, as follows:

Animation – a large class responsible for creating and managing the Animation mode of the program.

Batch – another large class which governs various simulations of the model in Batch mode (see Chapter 5.4 – Batch Mode).

CONST – a small non-functional class that simply provides a list of globally accessible constants for the program.

JPanelAnim – an extension of the standard JPanel class which simply allows for double buffered rendering. This is the black square area (in Animation mode) where all the nodes are drawn.

Node – a class describing an individual node in the network. This class governs the node movement and communication throughout the network.

SwingWorker – a standard Java class that has not yet been implemented into the standard API packages, hence it has to be compiled as part of the source code. Used to move the time consuming batch mode out of the event dispatching thread, to keep the GUI responsive.

Tabulator – this class is responsible for gathering data to be output in the program. Whilst other classes must call its functionality, this class manages and arranges the data supplied to it.

A discussion of the technicalities of the Node and Tabulator class follows.

4.2.2 The Node Class

The most fundamental class with regards to the simulation of the model, the node class contains a number of essential characteristics of each individual node (such as coordinates, radius, velocity) as well as the functions necessary to check for 'hello' signals, keep track of a given node's neighbourhood, and broadcast the message (if in possession of it) upon a new encounter. Note that the class represents just one node – it is up to the world (either the Animation or Batch class) to create multiple nodes as necessary and then manipulate them (generally update their position and status) each frame.

In order to update its current neighbourhood, a given node *i* periodically calls a function `checkNeighbourhood()` - for simplification, this period is once every frame or loop through the simulation logic. Note that although 'frame' is fairly specific terminology generally regarding animated sequences, it is used here interchangeably as one iteration through the main logic of the program. As the Node class is only representative of an individual node it needs some way of knowing the position and status of the other nodes that exist in the world at that time - hence the world passes this function the current list of all nodes in existence. What follows is a discussion of how the node first checks its neighbourhood and then propagates the message if appropriate.

Armed with knowledge of the other nodes in the network (which is unrealistic in terms of a real model), node *i* iterates through the list and performs a series of tests. It is simplest to outline these in pseudo code:

```
for all nodes
do

      if node ID != my ID and
      if node is broadcasting hello and
      if node's radius overlaps my x, y then

            add this node to current neighbourhood

      fi

od
```

Whilst it is unrealistic for node *i* to 'know' everything about every other node in the network, note that this unfair knowledge is not exploited to gain any unrealistic advantage in the model. Any other node *j* still has to be broadcasting a 'hello' message and be within range so that node *i* can hear it – thus no rules of the model are broken.

Once all nodes have built up their neighbourhood for the current frame, they all set about checking for any new encounters, by calling the Node class function `checkEncounters()`. If a new encounter has occurred this frame, the node broadcasts the message if it has it. This is outlined as follows:

```
for all neighbours
do
      if neighbour is new encounter and
      if i have message and
      if encounters < limit then

            if not broadcast this frame
                  encounters += 1
                  broadcasts += 1
            fi

            neighbour receives message
            neighbour passes message on to other nodes in its
            neighbourhood
      fi
od
```

It is difficult to properly and concisely explain the reasoning behind this function, as getting the nodes to properly broadcast and re-propagate the message to their neighbours proved to be very difficult. As a consequence, the resulting code to achieve this does not look as logical as one might expect. The above pseudo code is very abstract and does not provide the detail necessary to make clear the intricacies of the function, but it provides a general outline. It may seem strange that as opposed to nodes simply 'broadcasting' the message and then other nodes 'hearing' it if in range the node specifically gives the message to its neighbours – the reasons behind this are due to fairly low level implementation designs, which may become clearer in the actual source code listing for the class (see Appendix: Code Listings). Again, though it is not an accurate reflection of a real world situation, it does not violate the rules of the model and still achieves message propagation in the manner expected.

Although it will not be examined in detail, it is worth mentioning that the function which caters for a neighbour passing on the message to others in its neighbours is in fact recursive. It is for this reason that under situations where all the nodes are initially connected (that is each node's neighbourhood overlaps so that there is at least one link between node 0 to node *n*) the program may terminate instantly – node *i* broadcasts the message, node *i+1* hears it hence passes it on node *i+2* and so on until every node has the message. This is in fact correct; we assume (and model) that messages take no time to travel from a sending node to a receiver. In a real world situation where all nodes were initially connected, it would only take as long as the message took to travel through all the nodes (barring errors) to achieve total coverage. This statement is true of the model – it only takes as long as the message does to travel, which happens to be zero time.

4.2.3 The Tabulator Class

This class is responsible for gathering data from any variables desired from the program. It is designed in such a way that it can easily be re-used in any Java program that would benefit from a data gathering management system. Although the abilities of it are perhaps a little superfluous to the needs of the program at hand, the ease of use once it had been created proved incredibly beneficial when gathering a range results from the program. It can be used to pool data at regular intervals, useful when

making graphs charting a given variable (or variables) value against time, or called once, perhaps at the end of a process (in the context of this program, a run) with final statistical data. It is capable of tracking 50 different variables at once although this can easily be extended – however it was more than sufficient for this program.

Essentially, the Tabulator class works by storing data in individual sequences, but it is easier to visualise these in terms of columns in a spreadsheet. The main function `addData(Object, String)` is all that the third party programmer would need to use to start logging data. Assume that we have an integer *n* whose value we wish to track every frame. Given that we wish to track it, logically we have a name associated with that integer as well. Hence in a function called every frame we would simply write:

```
tabulator.addData(n, “messages broadcast”);
```

Note that `tabulator` is merely an instance of the Tabulator class. Returning to the analogy of the spreadsheet layout, this function would first create a column header “messages broadcast” and then add the value of *n* to the next empty row, incrementing the row counter every time `addData()` was called with “messages broadcast” as the column identifier. Expanding this slightly, we could add:

```
tabulator.addData(n, “messages broadcast”);
tabulator.addData(r, “messages recieved”);
```

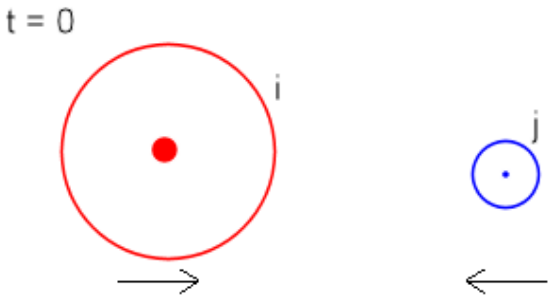
The tabulator would do the same for *r*, adding a new column “message received” and the value of *r* every time it was called. When wishing to output the data gathered, a call to `tabulateData(Array of String, String)` is all that is required. The first parameter is a column filter, whilst the second is the path to output the data to. The column filter is simply taken as an array of String objects marking which of the columns gathered by the earlier calls to `addData` are to be output. Assuming we wanted to output the data gathered in both of our columns, this could be achieved as follows:

```
String [] filter = {“messages broadcast”, “messages received”};
tabulator.tabulateData(filter, “msgs broadcast and
received.txt”);
```

Note that the filter argument determines the order in which the columns are output to the file; hence organisation of the output can be left until this point. The class will search for the requested columns and output them to the requested file. The columns are tab delimited which makes them ideal for direct importing into a spreadsheet program such as Microsoft Excel on Windows (which automatically assumes column separators are tabs) or KSpread on Linux (which has an option to set the delimiter as tabs). Once imported into such a program, data analysis and graph generation is trivial.

4.3 Implementation Problems

One of the most frustrating aspects of the program that was eventually discarded toward the very end of the project was the ability to produce nodes with varying radii (that is to say the radii were randomly distributed amongst the nodes between minimum and maximum values). This was in place for much of the development until the realisation that due to certain simplifications assumed in the model that differing radii cause unpredictable and unrealistic results. The problem comes with how a neighbourhood is defined (recall 3.2 – The Model).Recall that an encounter is defined as having occurred when a node *I* enters another node *j*'s neighbourhood. Assume that node *j* has the message, and because it encounters node *I* will according to the defined protocol transmit it. This presents a problem if node *j*'s radius is smaller than node *i*'s, as illustrated in the following example:



1) At this stage before the encounter there is no problem. Recall that *j* has the message and *I* does not.

2) Now however, *j* has encountered *I* and subsequently transmits the message – however, *I* is out of *j*'s transmit range and the message is wasted.

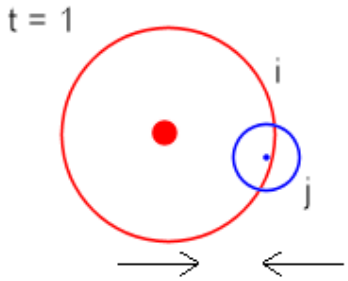


Figure 4.1 – Differing radii problems

This problem is an inherent design problem with the protocols defined, and subsequently the only solution was to insist on all node radii being the same. The actual radius does not need to be fixed at any given size, as long as all nodes adhere to it. It is a shame to lose this extra level of realism, as in a real world situation no two ranges would be identical, but for the sake of producing a simulation that adhered to the simplifications and assumptions established and functioned as expected it was a necessary sacrifice.

4.4 Drawbacks of the Java language

One of the greatest fears and the cause for so much initial delay in getting a skeleton program up and running was the worry about animation in Java. As already mentioned in the section above, getting even simple animation working satisfactorily proved to be quite a stumbling block, and the problems did not stop there. Although not critical, it was desired that the animation of any given run should proceed as smoothly and close to real time speed as possible. However, early initial testing revealed that even rather conservative desired frame rates (initially hoped to be 28fps or more) placed too much demand on the Java Virtual Machine – it could not perform the program logic and animation 28 times per second.

The following graph shows the degradation of performance as the desired frames per second increase. Six tests were carried out on a 600mHz processor with 512Mb RAM, running Microsoft Windows 2000. The run used each time was simply 10 nodes with no anti aliasing, running for 30 seconds. The y column 'animation speed (%)' was calculated by taking the actual elapsed time versus the simulated time achieved by the animation.

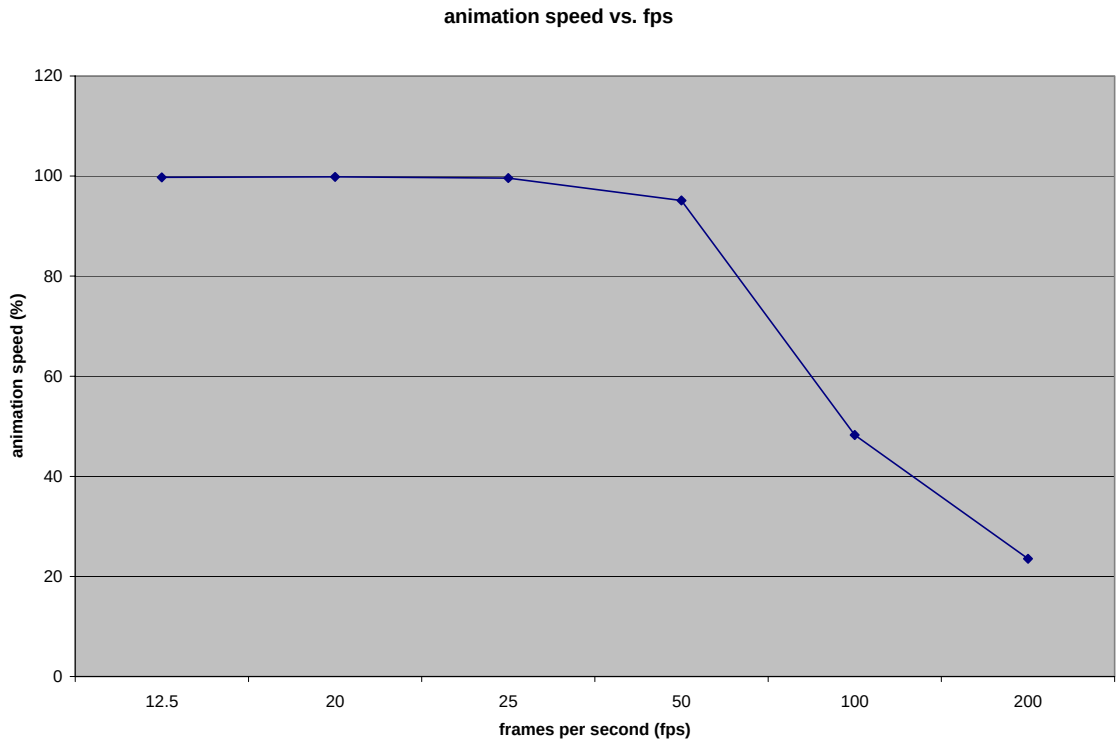


Figure 4.2 – graph showing animation speed vs. fps

Clearly the lower the desired frame rate, the less the animation overheads, hence why up until 25fps the simulated time was equivalent to the real time that had elapsed. As soon as the desired frames per second gets too high, the speed drops off significantly. It is not as straightforward as to say that perhaps 20fps would be a suitable speed due to the fact it ran the test at full speed with some processing power left to spare, as lowering the frames per second also lowers the accuracy of the simulation due to the way the model was implemented.

This is due to the fact that the concept of time and movement is based on the desired frames per second chosen in the program. Take for example, the node velocity, which is given in metres per second and then recalculated as a frame based value, as follows:

```
v = v/fps
```

Clearly a desired speed of 15m/sec results in smoother per frame movement at 50fps than 5fps. The upshot of this is that node movement looks smoother and collision and encounter detection are more accurate at higher frame as the discrete points at which the simulation updates are closer together and more seamless. If the frame rate is too low, the accuracy of the simulation is compromised (as well as the fluidity of the animation), however, if it is too high, the performance is seriously compromised.

By carrying out these tests it was decided that a frame rate of 25fps was the optimum desired value, and this was used for the duration of development. The details of why determining this optimum frames per second was necessary are fairly complex and not particularly relevant to this report, but suffice it to say they are to do with the timing operations used in the program and ensuring that the program runs as smoothly as possible.

This problem with the speed of the animation, combined with the difficulty of getting an animation panel working nearly led to switching development to C++ in order to take advantage of dedicated animation SDKs such as DirectX. However, the advantages outlined previously eventually outweighed this major falling point (indeed, the program was at various stages developed and tested on two entirely different specification Windows PCs, and a Linux machine – with no problems at all).

4.5 Gathering Sample Output

A key aspect of the initial project specification was to ensure that meaningful data could be extracted from the program to provide a variety of output graphs. As such, quite some time was invested into designing a class capable of fairly powerful data pooling. Once in place, this class allowed for related data to be collected fairly trivially from any point in the program before outputting a selection (or all) of the collected data in a columnar fashion to an output file. This data could be gathered on a parameter throughout the duration of a given run in animation mode (for example, network efficiency over time), but its strength was collecting related data in the batch mode of the program. By simply changing the variables of the program to collect the data from it was then trivial to output an array of graphs showing the affect of different parameters used in different runs. Many of these graphs can be seen Chapter 7 - Experimental Results, but even these do not fully demonstrate the power and flexibility of the class (the curious reader can investigate the source code of the Tabulator class, found in the Appendix: Code Listings).

Chapter 5 – User Manual

5.1 Setup

5.1.1 Where to get the Program

The program designed and discussed in this report is available on the CD which accompanies it, found in a sleeve on the inside back cover. If this CD is for any reason not present, the program can be easily obtained on request by emailing *Nicholas.Payne@ncl.ac.uk*. The compiled code is very small (<50Kb).

5.1.2 Version Warning

The first, most important point that must be made is that the program developed for and used in this paper requires the Java 1.5 runtime to be installed on the computer running it. At the time of publication this was not widely used, but nevertheless can be found at http://java.sun.com/j2se/1.5.0/download.jsp. Some important components of the program utilize new features only made available in the 1.5 API, hence it was not possible to rollback development to 1.4 at the cost of a narrower audience. Of course in time the 1.5 API will become the accepted standard (at least) and this will cease to be an issue. It is merely provided as a caution in the meantime.

5.1.3 Installation and Start-up

Installing the program is as simple as unzipping the `MANET sim tool – compiled.zip` file provided on the CD (located in `<cd drive>:\MANET simulation and animation tool\`). Assuming the user has the current Java Virtual Machine installed, running the program is straightforward. Either from the command prompt (in Win32 based operating systems) or from the terminal (in Linux and Mac operating systems), simply change into the directory where the zip file was extracted to and type:

```
java Run
```

This will start the program. If an error occurs, it is advised the user checks they are indeed running the Java 1.5 virtual machine.

5.2 Starting the program

Upon running the program one is faced with the following dialog (though it may look different the content will be the same)

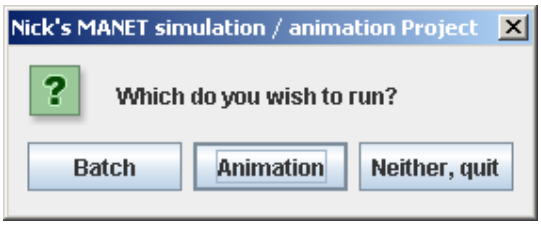


Figure 5.1 – Start-up dialog

Animation mode allows the user to create an animated model of the program and alter several parameters, described in detail below. **Batch** mode allows the user to run a series of scenarios using preset parameters, the details of which are output to an appropriate .txt file after the batch is complete. The **Quit** option will close the program.

5.3 Animation Mode

5.3.1 User Interface

Animation mode allows the user to set up varying parameters of the model and gauge their performance visually. The **area size** combination box allows the user to select the area which the black animation area represents. This can be set between $125m^2$ and $1000m^2$, and although it will not affect the absolute size of the animation area, the nodes drawn within it will be scaled appropriately (hence at $125m^2$ nodes will appear to have larger radii and move faster, whilst at $1000m^2$ the opposite is true).

The **msg threshold** option box indicates the propagation protocol to be used during the animation. Setting it to **1** provides a protocol similar to flooding (as and may result in a fairly short animation period depending on node density. This parameter can be set between **1** and **20**.

The **no. nodes** option box represents the number of nodes present in the animation. The lower bound is **2** whilst the upper bound, due to memory limitations, is **5000**. In practice however neither of these extremities will provide a particularly useful insight into the dynamics of any particular threshold.

5.3.2 Starting the Animation

Once happy with the parameters to be employed in the animation, clicking **Start** will commence the animation. As previously mentioned the nodes will be distributed randomly and evenly within the animation area, with radii and speeds determined by the configuration file (see 5.3.3 - Advanced Animation Options). The animation will run as close to real time speed as possible, however, due to the limitations of Java animation (see 4.4 - Drawbacks of the Java language) and the fact that ellipses are notoriously computationally intensive to draw it may not actually run at this ideal speed. The animation is done such that at lower frame rates, the nodes still move at the correct speed *relative to the simulated elapsed time*, not the actual elapsed time. Hence if the frame rate achieved is slower than preferred, the nodes will move slower and 'simulated' time will move slower. This preserves the integrity of the animation and means that nodes will only ever move a certain distance per frame, whereas in a frame based movement system (whereby the nodes would move relative to how much real time had elapsed) their movement distance per frame could not be guaranteed. To represent this discrepancy between simulated and real time elapsed, the top left of the animation screen will appear similar to the figure on the right. The factors that primarily affect this are the number of nodes present in the animation and the processing power of the machine running it. There is no real need to show the 'actual seconds' elapsed in the animation, it is merely provided as a visual guide to see at what sort of speed the animation is running.

Figure 5.2 on the right represents a typical cross section of the nodes represented during the animation progress. The smaller, bolded circles represent the nodes themselves, whilst the corresponding stroked circles represent their transmit & receive ranges. A red node is one which has not yet received the message, a blue node is one which has received it, and a white node is one which has the message and has retransmitted it the maximum amount of times possible (defined by the threshold parameter).

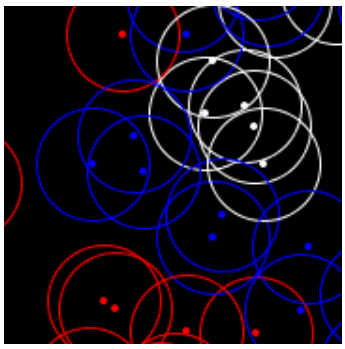


Figure 5.2 – typical animation cross section

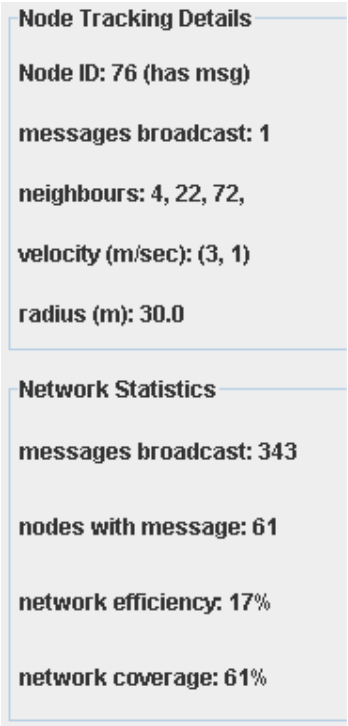
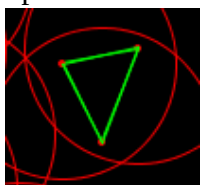


Figure 5.3 – animation feedback

Figure 5.3 on the left shows a section of the area to the right of the animation area. The **Node Tracking Details** panel will initially show the all the parameters as N/A – however, **left clicking** on any node in the animation area will place a rectangular highlight around it and its individual status will appear in this box. The details are fairly self explanatory, aside from **neighbours**, which represents the IDs of the nodes in this one's neighbourhood, and the two values of the **velocity** parameter, which are the node's x and y velocities, respectively. To deselect a node simply **right click** anywhere in the animation area.

The **Network statistics** panel shows, in order: the total amount of messages broadcast in throughout the network, the number of unique nodes who have received the message, the efficiency (number of nodes with message vs. total broadcast), and the network coverage (how many nodes have the message, as a percentage).

Further options available to the user can be found in the menu bar, particularly under the **Configure** menu. The first sub menu, **Graphics**, allows the user to make two graphical changes to the animation. The first, **Anti Aliasing**, switches the rendering mode to anti-aliasing as one would expect. Note that whilst this provides a significant increase in the graphical quality of the nodes, it comes at the price of increased processing overheads. Whether it is on or off, however, has no effect on the workings of the animation. The second option, **Neighbour Lines**, simply draws green lines between nodes, showing their neighbourhood connections. Again, this has no effect on the workings of the simulation, but adds another layer of feedback to the end user.



Note that the **Key** panel in the bottom right of the animation window denotes the colour schemes used in the animation. Recall that the model only attempts to propagate one unique message throughout the network, hence it is straightforward to define nodes as either being in possession of 'the' message (blue nodes) or not (red nodes). Recall also that an encounter limit *l* is also defined as the amount of times a node with the message will broadcast it upon encountering another node. When a node has broadcast the message *l* times (as set by the user) it becomes white. It will still move around in the network as usual but will not propagate the message any further.

Note that the final key indicating mono-directional communications links is redundant under normal program operation due to the identical radius sizes of the nodes. See 5.3.3 - Advanced Animation Options for how to change this.

Once the user has started an animation, it will run until one of two conditions is satisfied:

1. The network coverage reaches 100% (that is, all nodes in the network have the message)
2. Every node that had the message has broadcast it *l* times, and no other node can therefore receive it.

Upon termination, the program will output data collected during the run to an appropriate file in the directory the program is running. In order to make this easy to find and identify, the file is output as follows:

`“n[no. of nodes] t[threshold] – [time taken] secs.txt”`

e.g. a run with 50 nodes, threshold value 6 which took 73 seconds to complete will be output as:

`“n50 t6 – 73 secs.txt”`

The text file is output as four columns containing data about the run. This can be opened with any text editor, though it is output ideally for spreadsheet programs into which the file can be directly imported. If presented with the choice, the ‘column delimiter’ should be set to ‘tabs’. Note that the program can gather up to 20 simulated minutes of data which will, except under the most extreme circumstances, be more than adequate.

At any time during a given run the user is free to pause the animation (the start button, once clicked, will alternate between start and **pause**). This can be useful to examine closely how the network is performing at a given point in time. Note that the whilst the area size, protocol and node number options can ostensibly be changed at this time, they will have no effect until the simulation is reset and started again,

One final note about Animation mode is the presence of non functional menu options. The only two that in fact work are the graphics configuration options as mentioned above. The remaining options were not implemented in time. However, they were to provide no more functionality than is achievable by using the remainder of the interface (merely a parallel method of changing certain parameters).

5.3.3 Advanced Animation Options

Further to the options provided whilst running the animation side of the program, other factors can be altered by hand editing the anim.cfg file, found in the same directory as the other .class files. This file’s primary purpose is to load and save the last options chosen in the animation by the end user (area size, no. nodes etc) but also it also has a couple of extra parameters that are loaded discreetly. The first of these is the RADMINMAX [int1] [int2] parameter. As described in 4.3 – Implementation Problems, at one stage it was hoped that the nodes could have differing radii. This may not have been possible, but by changing these two ‘int’ values one can set a minimum (int1) and maximum (int2) radius size nonetheless. Beware however that nodes with differing radii will cause the program to function

incorrectly for the reasons described in the aforementioned section. The user can however safely set both values to the same to alter the radius size used by all the nodes, as long as it falls within limits imposed by the program (see the configuration file for details).

The other option that can only be altered by hand is the range of the speeds used by the nodes. This was not included in the GUI of the animation panel because it was not deemed a particularly important variable to necessitate changing. However, it is present in case the user wishes to experiment. Again it is governed by minimum and maximum limits imposed by the program, details of which can be found in the configuration file.

The user will also notice in the configuration file the presence of NUMNODES, AREASIZE and THRESHHOLD parameters in the configuration, but there is little value in altering them as they can just as easily be changed in the animation program using the GUI provided. Nevertheless, the option is there if desired.

If at any time the user hand edits the configuration file erroneously, the program is designed to cope and use default values where appropriate. Should this not happen, it is perfectly safe to delete the anim.cfg file – the program will detect its absence and generate a new one, based on default program values.

5.4 Batch Mode

5.4.1 Overview

Batch mode is far less interactive than the animation side of the program and is really only left in for the sake of completeness. It was heavily used during the testing of the program to produce the graphs and results data shown later (see Chapter 7 – Experimental Results). However, it remains in place as it can still be used to produce output files that the end user can examine and tabulate as they see fit. Beware that this mode is recommended only for advanced users who are comfortable with the program and the concepts employed in the model.

Batch mode is a sophisticated part of the program which can perform multiple runs of the model very quickly and then output a list of data gathered in those runs. During each run, one or more ‘variants’ (see 5.4.2 – Modes of operation) are changed, which have a knock on effect on the network performance and consequently the data output gathered in that run. The user can modify a range of parameters which allow for a large number of runs to be performed automatically, enabling rapid output of a wide spread of data.

Upon starting the program in batch mode the user will be presented with a screen similar to the following:

Figure 5.4 – Batch mode

Upon initialization, Batch mode reads in a number of parameters from the corresponding batch.cfg configuration file. These include the mode to operate in, and the variants and constants used in that mode, as shown above. The ‘variants’ are those parameters which will change during the batch execution. The ‘constants’ are those factors which remain the same for the duration of the batch execution. The differing modes are explained in the following section. Note that unlike in Animation mode, the only way to change any parameters in Batch mode is via the batch.cfg file.

5.4.2 Modes of Operation

In each mode there is always at least one variant. A ‘base’ starting value is supplied for that variant, along with a ‘loop’ value and finally an ‘increment’ value. Once started, the program will perform a series of runs starting with the variant at the ‘base’ value and incrementing it each ‘loop’ by the ‘increment’ value. In a mode where there is only one variant, the ‘loop’ value is equivalent to the number of runs performed in the batch. The capitalized words in the following explanations correspond to the parameters that represent them in the batch.cfg file.

Mode 1

variants: no. of nodes
constants: NODESPEED, NODERADIUS, BASETHRESHHOLD, AREASIZE

base value: BASENUMNODES
loop value: NODELOOP
increment value: NODEINCREMENT

Mode 2

variants: encounter threshold
constants: NODESPEED, NODERADIUS, BASENUMNODES, AREASIZE

base value: BASETHRESHHOLD
loop value: THRESHLOOP
increment value: THRESHINCREMENT

Mode 3

Note: this mode is slightly different in that there are no values in batch.cfg that affect the variant – they are all hard coded into the program. This is because the variant, area size, only has four values, so it is assumed all four are desired when performing the batch.

variants: area size
constants: NODESPEED, NODERADIUS, BASENUMNODES, BASETHRESHHOLD

base value: 125x125m
loop value: 4
increment value: variant is doubled each loop

Mode 4

Note: mode 4 is the only one involving two variants. This is because it was determined that these are the two best variants worth of simultaneous comparison

variants: no. of nodes, encounter threshold
constants: NODESPEED, NODERADIUS, AREASIZE

base values: BASENUMNODES, BASETHRESHHOLD
loop values: NODELOOP, THRESHLOOP
increment values: NODEINCREMENT, THRESHINCREMENT

It is perhaps worthwhile to provide an example demonstrating a simple use of Batch mode. Let us assume we wish to execute a batch of runs changing only the encounter threshold from a base value of 5 to a maximum of 15, keeping the number of nodes at 50 with radius values of 30m and speeds of 15m/sec in an area of 500 by 500 metres. In the batch.cfg file set the following parameters as follows (omitted values are of no relevance in this mode):

BATCHMODE **2 -** this sets the variant to be the encounter threshold.

BASETHRESHHOLD **5** – this satisfies our starting value.

TRESHINCREMENT **1** – this can be set to any non negative value, but given the threshold values are always fairly low, a low increment value is also logical.

THRESHLOOP **11** – Given our base and increment values, in order to reach our maximum desired threshold, we must set this as such because we want to test 5 to 15 inclusive.

NUMBASENODES **50** – this value doubles as a constant value in this mode – it may be a 'base' value but it never changes.

NODERADIUS **30** – set the constant node radius

NODESPEED **15** – set the constant node speed

AREASIZE **500** – set the constant area size

SAVEFILE **constant_nodes_50_variant_threshhold.txt** – of course, this can be any string, but it is advisable to make it somewhat relevant to the batch mode so it can be easily identified. It is advisable to make the file type .txt or similar so it can be opened easily. If using Windows and the user has Microsoft Excel 2000 or greater installed, the output can be written directly as a .xls file. This is great a great advantage as the data output is formatted ideally for spreadsheet programs, making graph generation a simple step.

This is a very simple application of the batch mode but demonstrates the flexibility it provides. With a little effort all manner of scenarios can be simulated in large batches. It is also worthy of note that by setting the relevant increment values to zero one can perform a batch of runs with exactly the same parameters, and the data output can then easily be converted to form an average. The modes provided are not intended to be an exhaustive combination of potential scenarios, but during extensive testing they were the most common variations used. Note that should the user **stop** the batch whilst in progress, the output file will contain all the data gathered up until the point the execution was halted. Please note that Batch mode is very computationally expensive and may dominate a lot of system power until it has executed which may cause other programs to become unresponsive, particularly if it is a large batch of runs being executed. A work around to this for Windows 2000 or Windows XP users is to open the task manager, right click on java.exe (under the Processes tab) and set its priority to Low. This will allow the virtual machine to use any idle processor power but allow other programs to function as normal.

Chapter 6 – Program Testing

6.1 Overview

Testing was carried out at various stages throughout the program, particularly when new features were introduced. It was not quite performed as expected and as described in 1.3 – Project aims, in that the 'Waterfall Model' of design was not particularly appropriate when it came to the actual coding of the program. Although the initial design had clearly outlined what needed to be achieved it proved impossible to code discrete chunks of the program without returning to them frequently. However, the two distinct modes of operation in the program were implemented for the most part in this fashion. The animation mode concentrated on first until it was at a suitable level of implementation and then the batch mode was started. This was something of a logical step, as batch mode is essentially very similar to animation, using the same logic to govern the model but without the need to draw anything or slow the simulation to a realistic speed.

That said, there were certain key stages where extensive testing was performed, notably when message propagation was first introduced and then slowly as it was refined. When considered complete, the two main categories that required testing were the actual animation process and Graphical User Interface, and whether the data output did in fact seem correct – which is slightly subjective. To combat this latter problem, sample graphs using the same test categories demonstrated in [7] were generated from the program. Given that the protocols implemented are based on those in that paper the resulting graphs should be similar to those in the paper, as indeed they were (see Figure 7.1 in Chapter 7 – Experimental Results).

6.2 Animation Testing

The animation of the simulation process served as a very helpful tool in determining whether the program was functioning as expected or not. Whilst not the most scientific method of analysis, the advantage is that it is immediately obvious if a message is being propagated incorrectly. Extensive knowledge of the workings of the program, hence knowing what to look for and the expected knock on results of a message being propagated make it very straight forward to spot a node acting anomalously.

Testing of the GUI was performed on a slightly more functional basis. It is fairly trivial to create a set of use cases to ensure each item of the interface performs as expected. These are shown as follows:

Do the 'start' and 'reset' buttons function as expected? – When faced with 'start', the user would expect this to start the animation, as indeed it does. It immediately changes to show 'pause' which will pause the animation, changing the button to 'resume', which restarts the animation. If at any time the user clicks 'reset', the animation is cleared and reset. The other button returns to 'start' and the animation is in the same state as it was when first initialized.

Can the user enter invalid parameters for the 'msg threshold' or 'no. of nodes' – these inputs are JSpinnerBox types which must take a minimum and maximum value on creation, and they automatically do not accept any other input – thus this is automatically taken care of.

Can the user in anyway cause the program to animation program to perform unexpectedly? – aside from advanced editing of the anim.cfg file (see 5.4.4 – Advanced Animation Options) which is provided with appropriate warnings, the animation program is very robust and cannot be broken. The inputs are all verified and the buttons always perform as expected.

6.3 Batch Mode

Testing of batch mode was fairly straightforward as the main verification lies in the outputted data provided by it. It runs the same simulation logic as the animation mode and hence performs identically. It was necessary to ensure that it was indeed changing the correct variants in each mode, a fact confirmed by the intermediate output it provides and the correctness of the outputted data. The interface is extremely limited, consisting only of a 'start' and 'stop' button, and testing of these was consequently very trivial – the start button does indeed start the batch (and has no effect until it is stopped or has terminated), and the stop button terminates the batch mid execution.

Chapter 7 - Experimental Results

7.1 Overview

In order to examine the functionality of the model once it had been fully written and thoroughly tested, a number of different runs were made using a range of differing parameters, in order to evaluate the effect they had on different factors of the network such as:

- Propagation time – the time taken for the message to either reach all the other nodes in the network (assuming 100% coverage) or for the run to be terminated due to no node in possession of the message being able to propagate it further (i.e. all nodes with the message have reached their encounter threshold).
- Network coverage – the amount of individual nodes which, at the end of the run, are in possession of the message, computed as a percentage ((nodes with message / total nodes) * 100).
- Network efficiency – the amount of unique nodes which are in possession of the message versus total amount of messages broadcast, again computed as a percentage. This gives an indication as to the unnecessary overheads that may occur in runs using a higher encounter threshold.

Bearing these in mind, obvious parameters to alter include the total of number of nodes in the network, the area size simulated, node speed and so on. Perhaps the most important experiment to perform is one examining the network coverage of a given scenario – as this is ultimately the aim of any given protocol (though achieving that coverage in a reasonable time is of importance as well).

7.2 Experiment 1: General Network Performance

This first experiment focuses on using a set of constant values as follows examined against the above factors of interest.

```
No. of nodes:           100
Node velocity (m/sec):  15
Node radius (m):        30
Area size (m^2):        500
Encounter threshold:    1, 2, 3... 10 (inclusive)
```

The experiment was carried out 10 times; that is 10 runs were performed with the nodes randomly redistributed at the start of each new run. The following graphs represent averages of these 10 runs.

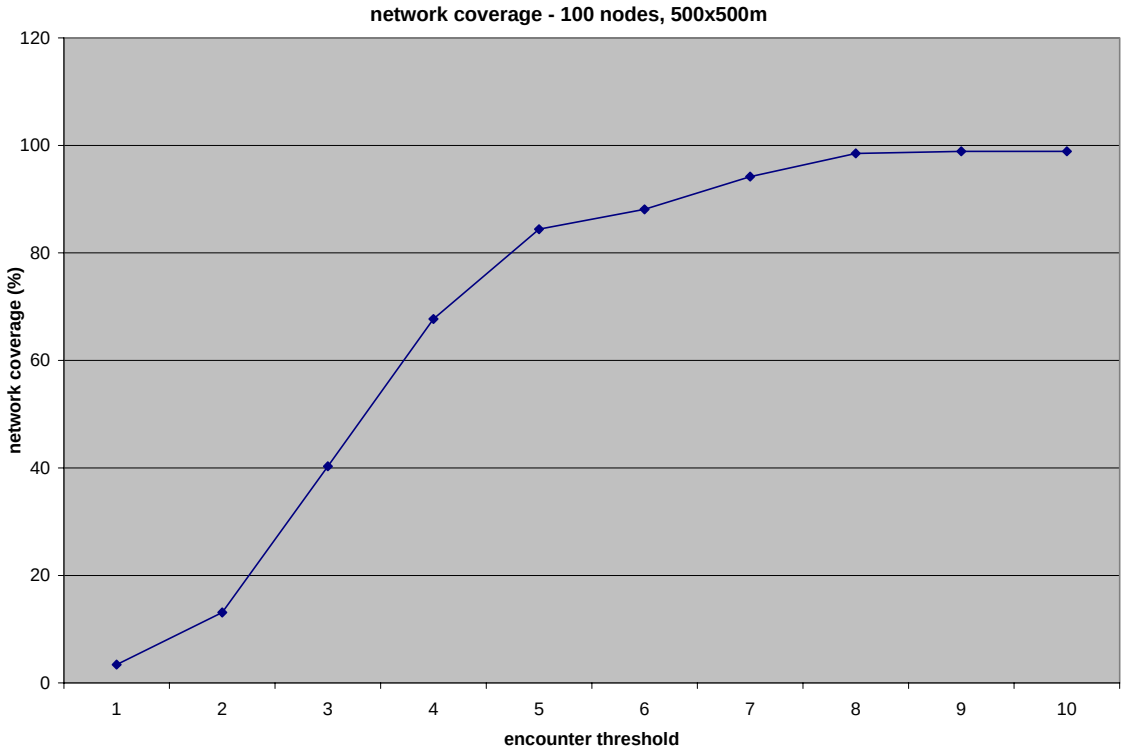


Figure 7.1 – graph showing network coverage

The graph in Figure 7.1 is concerned with the network coverage of the given settings. As one would expect, 1-propagation achieves very poor network coverage, whilst higher thresholds fare far better, achieving near to 100% coverage at 8-propagation. Note that thresholds 9 and 10 did in fact achieve 100% coverage at the end of various runs despite having slightly lower averages (98.8% in both cases).

These results show a not unexpected linear increase in network coverage until 5-propagation when the coverage increase suddenly slows. It is possible to deduce from this that the last 20% of the nodes in the network are more difficult to propagate the message to (as this is where the increase slows from). The reason behind this is due to the fact that these nodes are more likely to be closer the extremities of the simulation area, because for any node *i* to travel from any arbitrary point to any distant extremity the probability is it will experience many encounters. Assume *i* is a node with the message having not yet experienced any encounter (other than receiving the message), starting at point *0, 0* and ending at point *0, max_y* (i.e. travelling the length of the area in a straight line), with a maximum encounter limit *l*. In order to still be able to propagate the message at *0, max_y* it will have had to experienced no more than *l* -1 encounters, which even in a high propagation protocol is very unlikely. This situation can be visualized by the following screenshot, taken using the same constant values used in the above runs (the node speed, area size etc), using 5-propagation at exactly 80% network coverage:

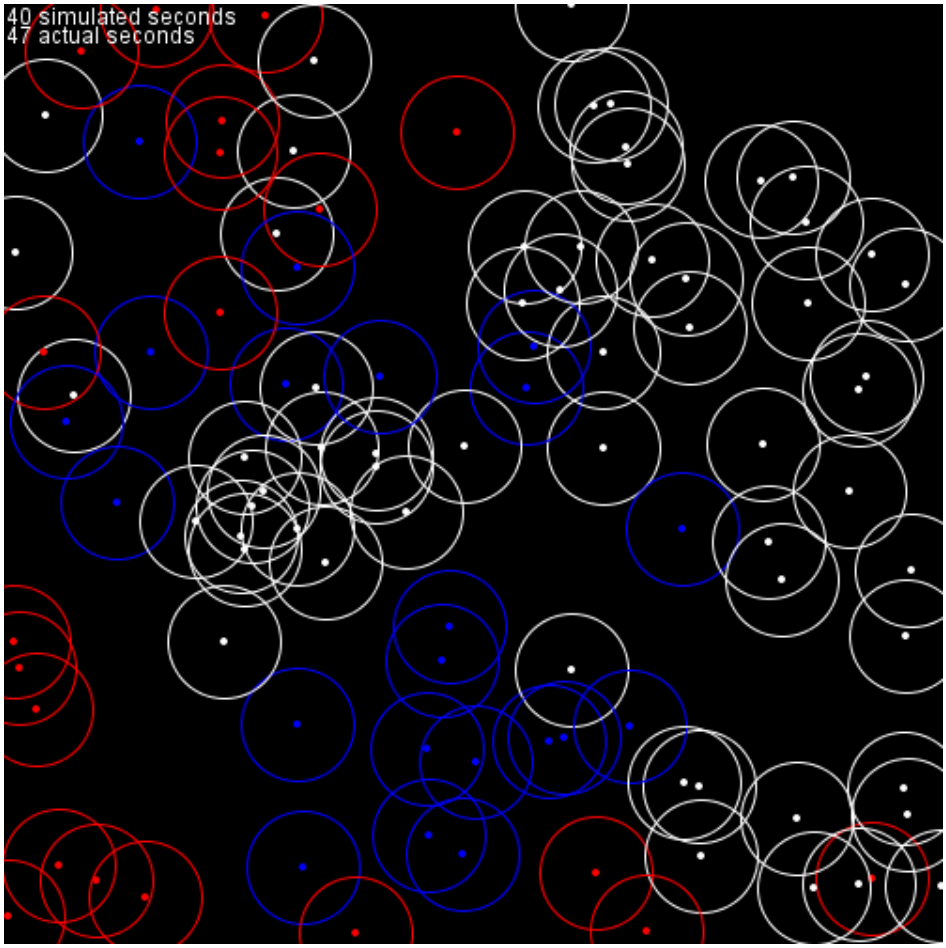


Figure 7.2 – animation screenshot

From this screenshot it is clear that most of the nodes that do have the message have already reached their encounter threshold by the time they reach the extremities of the simulation area. Assume that the extremities are defined as being 50 metres in from each border. In this case there are 20 nodes in the extremities which have the message, and 17 of them (85%) have reached their encounter thresholds, leaving only 3 nodes in the extremities capable of message propagation. Of the 20 nodes in the network who have not received the message, 15 (75%) of them lie in the extremities of the simulation area, thus we can see a large imbalance in the number of nodes who need the message versus those who can propagate it in these extremities at any one time. In a higher propagation protocol, for example 10-propagation, we would assume that this pattern still generally holds, though there will be more nodes in the extremities capable of propagation.

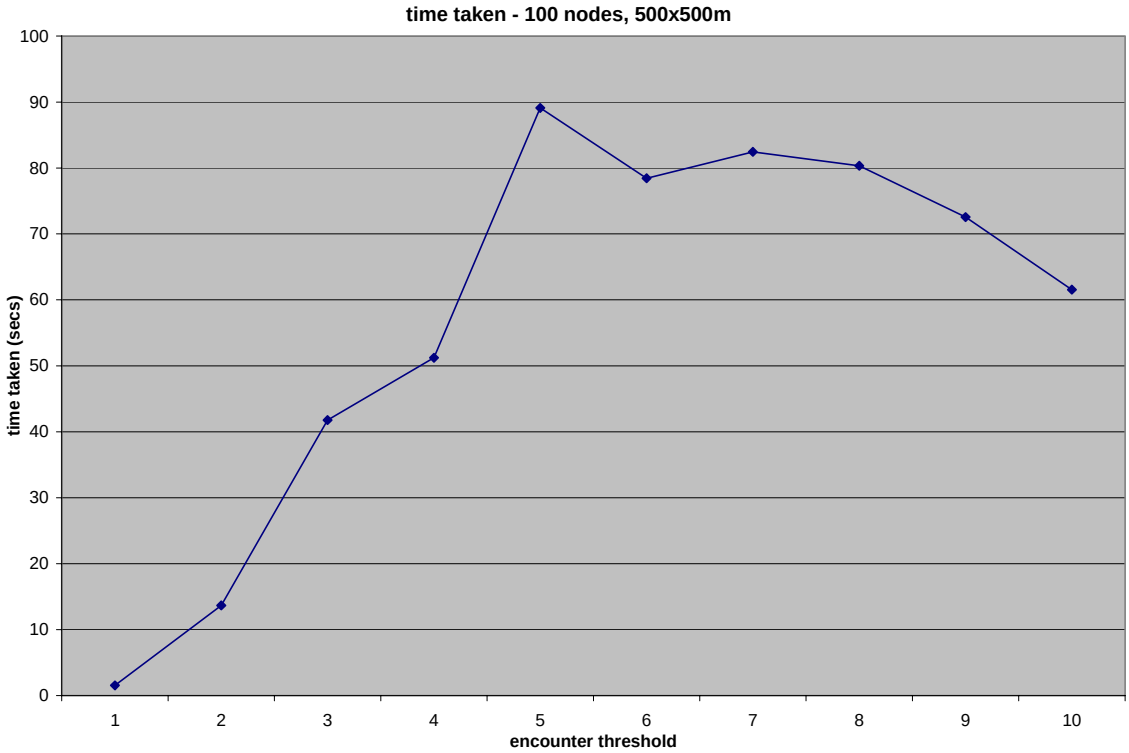


Figure 7.3 – graph showing time taken

Figure 7.3 represents the average time taken per run to achieve the maximum network coverage possible using the different protocols. Note that whilst it may appear that 1-propagation protocol is incredibly quick, one must recall that it is only covering a small fraction of the network (3.4%) before terminating due to the fact no node can propagate the message further. The same is true of the lower number protocols, with time increasing linearly until peaking at 5-propagation and then decreasing linearly until 10-propagation. This can be assumed to be a result of the fact that although 5-propagation achieves a high network coverage (84.4%), toward the end of each run the amount of nodes still able to propagate the message is considerably lower than in protocols with higher encounter thresholds, hence the nodes without the message have to wait longer until they encounter one that can still propagate it. In the 10-propagation protocol there are still many nodes able to propagate the message toward the end of the run, so it terminates quicker despite achieving a similar coverage (98.9%). Note that this is consistent with the issues discussed previously with regards to the extremities of the simulation area.

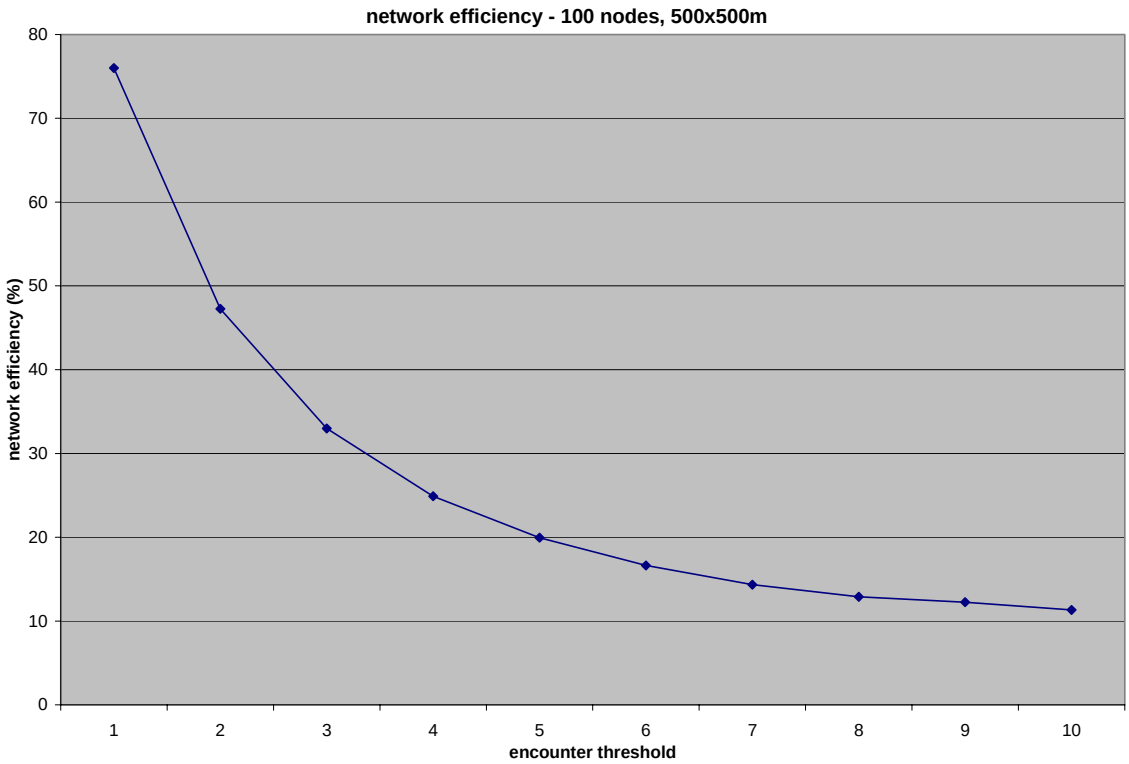


Figure 7.4 - graph showing network efficiency

Figure 7.4 represents the average network efficiency per run for each of the different protocols. It shows an exponential decline in network efficiency, though in a similar fashion to the previous graph the results for the lower thresholds are misleading as they represent an efficiency based on minimal coverage. As the network coverage increases it is to be expected that the efficiency of propagating the message decreases due to the fact that more wasted messages are broadcast (that is messages broadcast to a node that already has the message) – this is an unavoidable and expected trade off against having a higher encounter threshold.

7.3 Experiment 2 – Network Coverage Degradation

The following graph attempts to demonstrate the assumption that the last fraction of nodes of a given run's coverage take longer to propagate than rest. Note that the data presented in it was not taken from the same ten runs as the preceding three graphs, thus it cannot be directly compared with them. It was gathered by modifying the batch mode of the program slightly to produce a running log of the network coverage over time for each threshold 5, 6, …, 10. It is the result of one run per threshold, not an average.

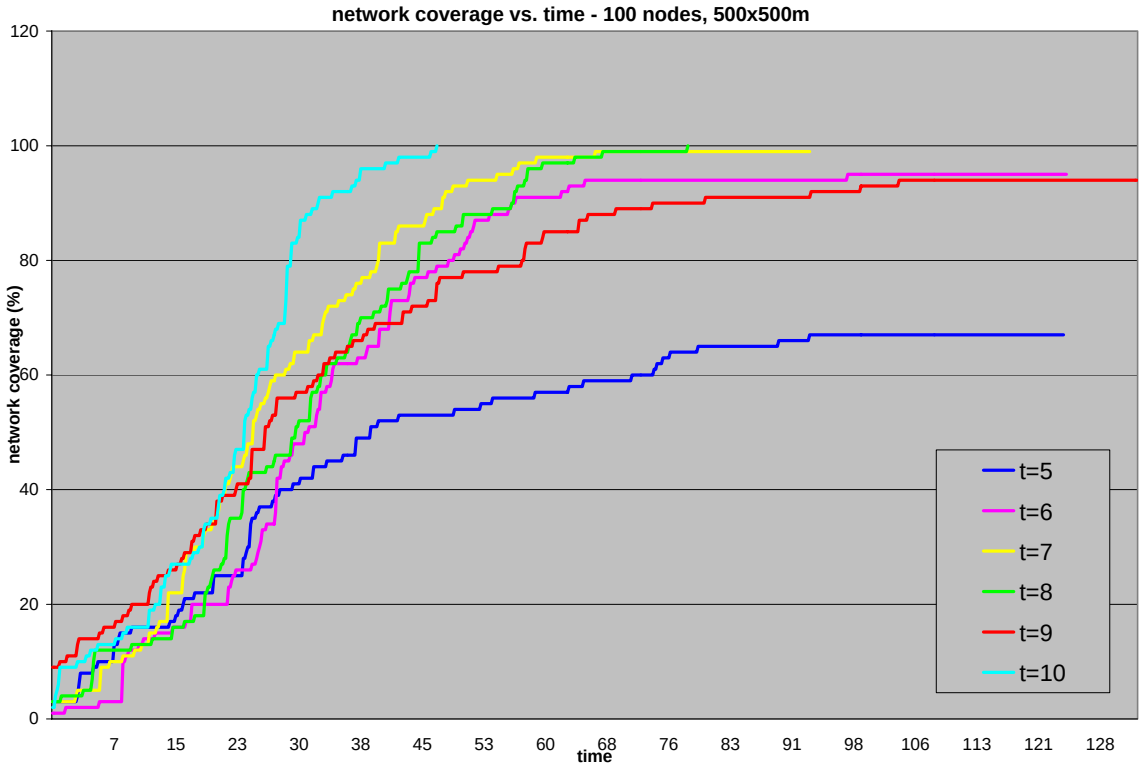


Figure 7.5 – network coverage over time

Thresholds 1 – 4 were omitted from this test as they never achieve anywhere near 100% coverage and thus cannot be tested to see if the assumption is true. Whilst 5-propagation does not achieve near 100% either, it is close enough to bear relevance to the assertion.

The thresholds tested cover the first 40% of the network in approximately the same amount of time in a linear fashion – the fluctuations mainly due to nodes receiving a new message and instantly broadcasting it to many neighbours (this is illustrated by t=6 at around 10 seconds). After this point the performance of 5-propagation rapidly drops off as it enters the last 20% of its final coverage (at 53.6% of 67%). The other protocols still perform similarly as they have not entered this final coverage period. 6-propagation does indeed appear to deteriorate rapidly in its last 20% coverage (at 76% of 95%). The actual percentage does logically decrease as the encounter threshold increases, due to the fact that there are more nodes able to propagate the message for a longer period (hence to a higher network coverage), but

assessing accurately this actual percentage is very difficult due to the random nature of the network topology.

What can be deduced clearly from the graph is that there most certainly is a point at which a given protocol's performance deteriorates – we shall call this the 'drop off' point. Logically this occurs when there are few nodes left able to propagate the message, hence why 5-propagation (which will 'run out' of actively propagating nodes first) reaches this point first. It is interesting to note that using 10-propagation the graph is almost linear, with the drop off point coming very near the total network coverage achieved. This is because the total network coverage is in fact 100% hence there is at least one actively propagating node but probably more – so the network performance does not degrade badly by the time the simulation is complete. If the coverage needed for a termination was greater than 100% (if that were possible) this protocol would indeed experience at some point a drop off similar to the other lower propagation limit thresholds. Based on this we can assume that protocols with higher propagation limits will experience this drop off point ever closer to 100% and then at some point past it, and an infinite-propagation protocol will always experience a completely linear propagation rate, though in such a network the network efficiency will be very low.

7.4 Experiment 3 – Node Density Effect on Network Performance

The following experiment analyses the affect differing numbers of nodes have on a given simulation area. The set of values used are as follows:

```
No. of nodes:           50, 100, 125, 150, 200, 250
Node velocity (m/sec):  15
Node radius (m):        30
Area size (m^2):        500
Encounter threshold:    1, 2, 3... 10 (inclusive)
```

Note that the important factors when considering the relative density of the network are the number of nodes, their radii and the area size. It is only useful to alter one of these at a time, hence in this experiment radius and area size remain constant. The following graph shows the results of these differing densities, each run once (thus the graph does not show average results, merely of one individual run per density).

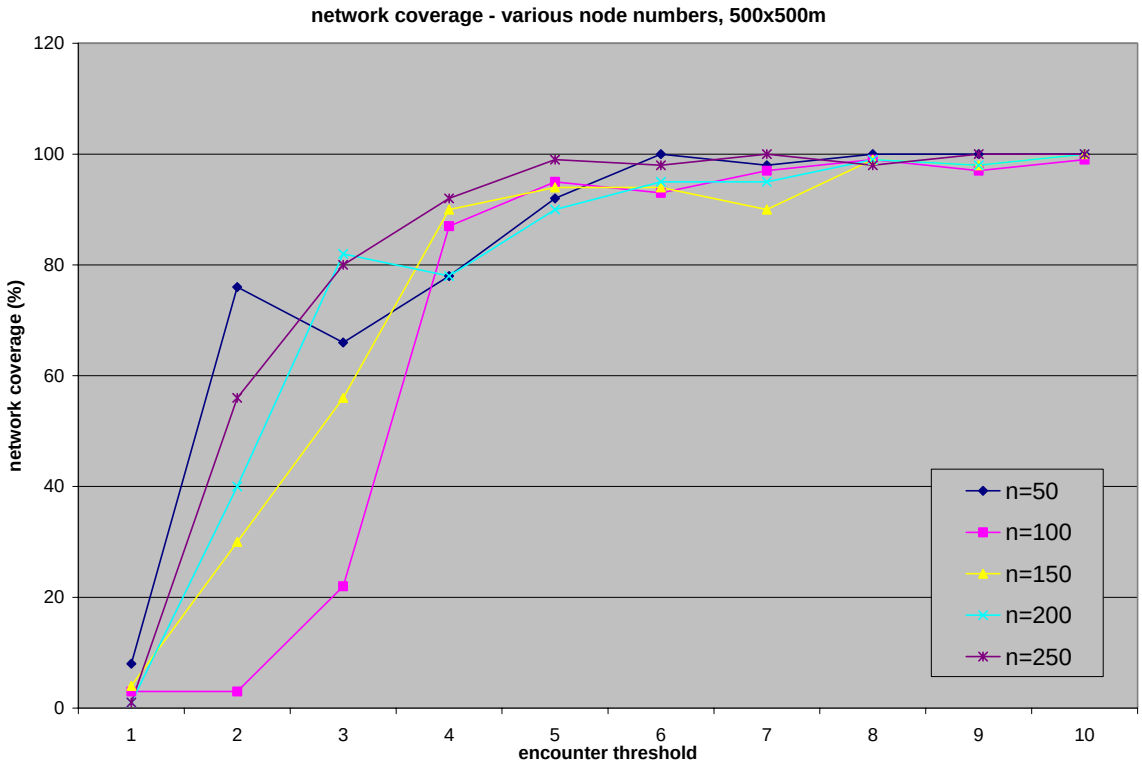


Figure 7.6 - network coverage, various node densities

Perhaps surprisingly, this shows that it is not so much the density of the network (within reason of course) but the encounter threshold that has the dominating effect on network coverage. When reasoned through properly the explanation is actually fairly straightforward: the fact that there are less nodes in the network means there are less encounters on the whole – hence there are less wasted encounters and therefore the network coverage is fairly similar for differing densities. As such, each density is affected by each of these factors to a relative degree; hence they achieve similar network coverage. By the same token we would expect the network efficiency for each density to follow the same general pattern shown in Figure 7.3. This is indeed true:

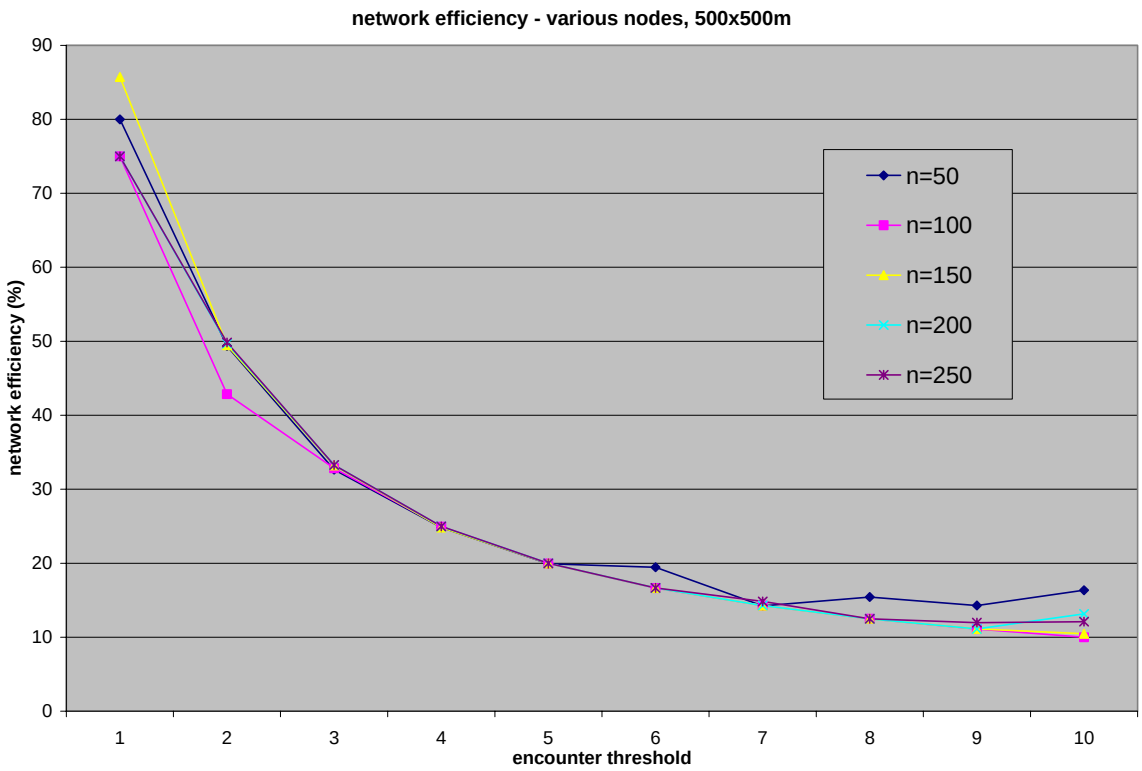


Figure 7.7 - network efficiency, various node densities

Clearly network efficiency is barely affected by the density of the nodes in the network. However, one adverse we would expect in lower densities is an increase in time taken to reach maximum network coverage. This is shown in the following graph.

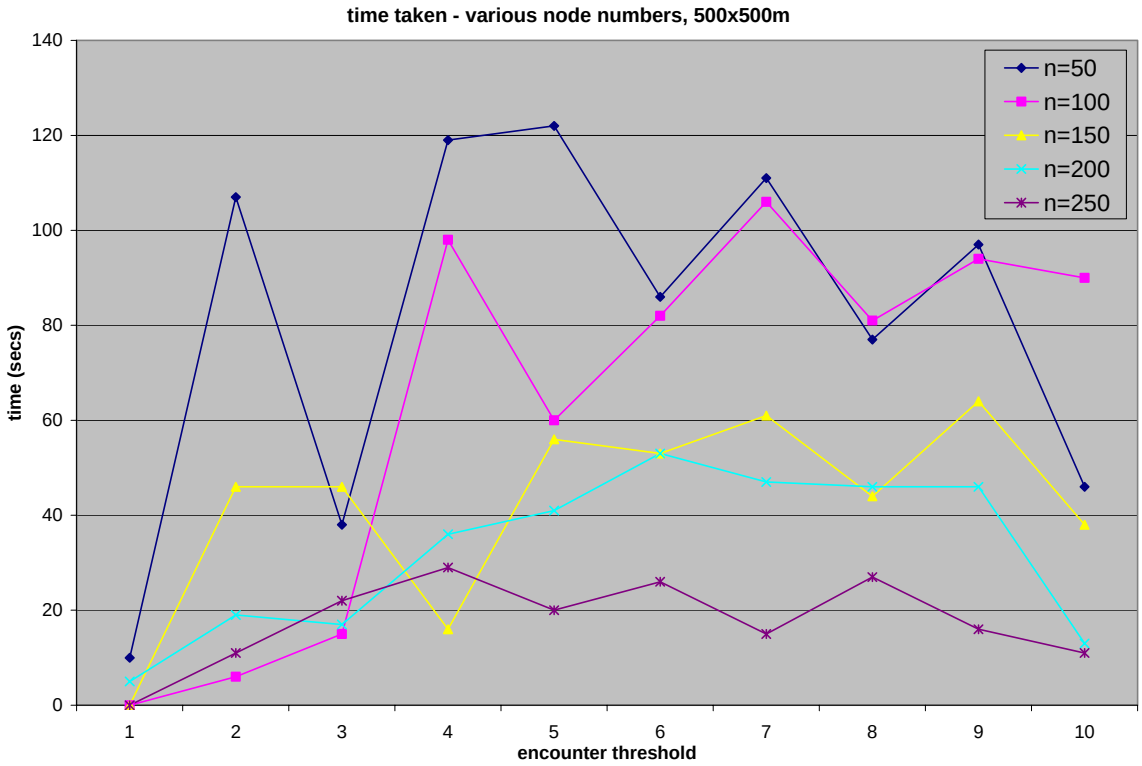


Figure 7.8 – time taken, various node densities

As expected, the lower the node density, generally the longer it takes to reach its maximum network coverage. This is clearly where the major advantage to having more nodes in a given area is found; propagation time.

7.5 Results Summary

The experiments examined provide a range of interesting results which are useful in attempting to gauge appropriate protocols to be used and determining the overriding factors affecting network coverage, efficiency and propagation time. Note that the results are not completely exhaustive – area size, node speeds and node radii were kept constant throughout, but this was to provide a consistent base on which the results presented could be compared.

Chapter 8 – Project Conclusions

8.1 Evaluation

The initial aims set out were on the whole achieved, particularly the implementation of the *l*-propagation family of protocols, though some of the more advanced features were left out due to time limitations and their complex nature. This stemmed from a rather naïve initial view of the problem and a rather shallow initial grasp of the complexity of the project as a whole. However, the principle aim was to implement a family of protocols capable of as thorough network coverage with as little overhead as possible, and this was achieved satisfactorily. The animation side of the program turned out, despite a not inconsiderable amount of difficulty initially, to work very well. When considering the aim was to provide the user with a tangible overview of the network in action, the end product compares favourably. It not only draws abstracted but clearly recognizable nodes, but provides the user with simple colour coordinated feedback on their status, as well as a more detailed breakdown (if desired) and a general breakdown of the network statistics as a whole. Any more onscreen activity could have endangered obfuscating the visibility of proceedings, and any less would not have provided a satisfactory level of feedback to the user.

One aspect of the project that was initially misconstrued was the distinction between ‘simulating’ and ‘animating’ a simple MANET. It was assumed that the two were distinct and separate elements, whereas in fact the end result was a combined animation *of* a simulation. The distinction disappears when one considers that in order to animate a given number of nodes interacting properly, their movement and communications must be governed by a simulated set of rules. The original concept of simulation was converted into a separate mode of operating the program (see Chapter 5 - User Manual) whereby the program performs a series of runs using different parameters and then outputs the data to a file for analysis. This mode works extremely well and allows for rapid data gathering, allowing the end user to perform numerous tests employing new runs to check for themselves that the simulation does in fact work.

The animation mode was implemented as almost as initially expected bar some refined detail that was eventually impossible to implement. It had, for example, been hoped that the animation would show actual messages being broadcast from node to node. As it transpired, this was not possible due to the instantaneous nature of messages being propagated (that is they only exist for one discrete point in time in the node’s broadcast radius). On the contrary however, added interactivity was provided that had not initially been forecast, such as the ability to click on and track any particular node to receive detailed information about it and its neighbourhood. This added step of true interactivity with the user whilst the program is running serves to make the program a more rewarding user experience.

8.2 Learning Outcomes

The original hypothesis of the project was a challenging and interesting proposal that was on the whole accomplished whilst providing a valuable insight into several new areas of interest. The area of MANET technology was an entirely new one to the author which provided an exciting new field of research and also helped developed an enhanced understanding of networks and their topologies as a whole. The development of the accompanying program also provided the chance to develop increased strengths in the Java language, particularly relating to the implementation of the Graphical User Interface and working with simple 2D animation, the latter of which had never been investigated at all before commencing with the project.

8.3 Future Work

The program developed for the project provides a very basic implementation of an *l*-propagation family of protocols; however it has the potential to be expanded in numerous directions. In a real MANET, multi hop routing protocols are increasingly being explored, and many draft proposals for a variety of protocols exist, particularly on the official MANET charter site [14]. Expansion of the simulation model to include one or more of these protocols, whilst a complex and time consuming procedure outside the scope of this project, is certainly possible. The batch mode could also be readily expanded to provide more preset modes and would benefit from the addition of a comprehensive GUI providing the user with a great deal of control without needing to hand edit the configuration file. The additions of a choice of how data is to be collected during the program as well as more flexible data output would also be welcome.

8.4 Summary

This paper has discussed the issue of Mobile Ad Hoc Networks and provided an implementation of a basic family message propagation protocols, with a program provided simulating the model described with the additional bonus of animating it. The current state of the project is open enough to allow many exciting future developments to be implemented in an attempt to simulate ever closer and more realistically a real world Mobile Ad Hoc Network.

References

[1] *BEDD Community*, BEDD 2004, last viewed 04/05/05, http://www.bedd.com/

[2] Boukerche, A. & Bononi, L., 'Simulation and Modeling of Wireless, Mobile, and Ad Hoc Networks' in *Mobile Ad Hoc Networking*, p. 396, ed. S. Basagni, IEEE Press, Piscataway, New Jersey

[3] Boukerche, A. & Bononi, L., 'Simulation and Modeling of Wireless, Mobile, and Ad Hoc Networks' in *Mobile Ad Hoc Networking*, p. 394, ed. S. Basagni, IEEE Press, Piscataway, New Jersey

[4] Boukerche, A. & Bononi, L., 'Simulation and Modeling of Wireless, Mobile, and Ad Hoc Networks' in *Mobile Ad Hoc Networking*, p. 387, ed. S. Basagni, IEEE Press, Piscataway, New Jersey

[5] *Computer history museum – Exhibits – Internet History – 1970's*, Computer History Museum 2004, last viewed 04/05/05, http://www.computerhistory.org/exhibits/internet_history/internet_history_70s.shtml

[6] Conti, M, 'Body, Personal and Local Ad Hoc Wireless Networks' in *The Handbook of Ad Hoc Wireless Networks*, Ch. 1 p. 3, ed. M. Ilyas, CRC Press, Boca Ranton

[7] Cooper, D.E., Ezhilchelvan, P., Mitrani, I., 'High Coverage Broadcasting For Mobile Ad-hoc Networks'

[8] *IEEE 802.11, The Working Group Setting the Standards for Wireless LANs*, the Institute of Electrical and Electronics engineers, Inc., last viewed 04/05/05, http://grouper.ieee.org/groups/802/11/

[9] *Java Technology*, Sun Microsystems, Inc. 1994-2005, last viewed 04/05/05, http://java.sun.com

[10] *JCreator – Java IDE*, Xinox Software 2000-2005, last viewed 04/05/05, http://www.jcreator.com

[11] Law, A.M. (1982), *Simulation Modeling And Analysis*, 2nd ed.

[12] Law, A.M. (1982), *Simulation Modeling And Analysis*, 2nd ed. p. 1

[13] Macker, J.P. & Corson, M.S., 'Mobile Ad Hoc Networks (MANETs): Routing Technology for Dynamic Wireless Networking' in *Mobile Ad Hoc Networking*, p. 257, ed. S. Basagni, IEEE Press, Piscataway, New Jersey

[14] *Mobile Ad-hoc Networks (manet) Charter*, IETF Secretariat, last viewed 04/05/05, http://www.ietf.org/html.charters/manet-charter.html

[15] *Mobility Framework (MF) for OMNeT++*, last viewed 04/05/05, http://mobility-fw.sourceforge.net/hp/index.html

[16] *The Network Simulator – ns-2*, last viewed 04/05/05, http://www.isi.edu/nsnam/ns/

[17] *OMNeT ++ Community Site*, OMNeT ++ Community Site 2005, last viewed 04/05/05, http://www.omnetpp.org/external/whatis.php

[18] *OPNET Modeler – Accelerating Networking R&D*, OPNET Technologies, Inc. 2004, last viewed 04/05/05, http://www.opnet.com/products/modeler/home.html

[19] Patin, F., *An Introduction to Java fast true colour 2D animation and games graphics*, Frédéric Patin 2004, last viewed 04/05/05, http://www.yov408.com/javagraphics/javagraphics.html

[20] Samar, P., Pearlman, M.R., Zygmunt, J.H., 'Hybrid Routing: The Pursuit of an Adaptable and Scalable Routing Framework for Ad-Hoc Networks' in *The Handbook of Ad Hoc Wireless Networks*, Ch. 14 p. 15, ed. M. Ilyas, CRC Press, Boca Ranton

[21] Suh, Y-J., Kim, W-I., Kwon, D-H., 'GPS-Based Reliable Routing Algorithms for Ad Hoc Networks' in *The Handbook of Ad Hoc Wireless Networks*, Ch. 21 p. 9, ed. M. Ilyas, CRC Press, Boca Ranton

[22] *About the JFC and Swing*, Sun Microsystems, Inc. 1995-2005, last viewed 04/05/05, http://java.sun.com/docs/books/tutorial/uiswing/14start/about.html

[23] *Wireless Ad Hoc Networks: Mobile Ad Hoc Networks*, National Institute of Standards and Technology 2004, last viewed 04/05/05, http://www.antd.nist.gov/wahn_mahn.shtml

Appendix: Code Listings

Due to the large size of the source code for the program accompanying this report, it is supplied on the CD supplied with this report, found on the inside back cover. The code is preserved in the original .java files, found in `<cd drive>:\Appendix – Program Source Code`. Should this CD not be present, the code is available on request by emailing *Nicholas.Payne@ncl.ac.uk.*