MANET's on Smartphones

Cyril O'Floinn
B.A.(Mod.) Computer Science
Final Year Project  April 2012
Supervisor: Dr Stefan Weber

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_______________________________________ April 30, 2012

Cyril O'Floinn
Abstract

Mobile phones these days have moved well beyond simple devices that just make voice calls with many models now having the computing power and wireless connectivity to rival netbook computers earning them the title of smart-phones.

And yet for all their wireless connectivity they still depend for the most part on a fixed or centralized network infrastructure for voice and data communications. To make a call or send data over the internet a user will invariably have to use some form of fixed or centralized infrastructure in the form of a wifi access point or a cell tower belonging to a mobile phone operator.

For reasons including economic underdevelopment, geographic remoteness, or areas affected by a recent disaster the fixed networking infrastructure may be severely limited, too costly to build or use or simply be non-extant. In such situations having immediate access to an alternative infrastructure that can be quickly assembled is self maintaining and free to use may be vital for communication. So why not use the phones themselves?

This project will look at how the wireless capabilities of devices such as smart-phones can be used to create ad-hoc or decentralized networks where the devices themselves form the physical network infrastructure. Such networks are known as mobile ad hoc networks (MANET's) and this project will examine the issues behind MANET's, the technologies and algorithms that can used as well as the work involved in providing a solution by actually building a MANET out of existing smartphones.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>2</td>
</tr>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 1</td>
<td>5</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Objectives</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Project Outline</td>
<td>6</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>7</td>
</tr>
<tr>
<td>Background and Related work</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Wireless Access</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Addressing</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Routing</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Service Discovery</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Voice over IP</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Related Work</td>
<td>12</td>
</tr>
<tr>
<td>2.6.1 Wireless Mesh Network</td>
<td>12</td>
</tr>
<tr>
<td>2.6.2 The Serval Project</td>
<td>12</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>14</td>
</tr>
<tr>
<td>Design Overview</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Platform</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Wireless Access</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Routing</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1 Netfilter - Linux network stack monitor</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 DYMO - A Manet IP routing protocol</td>
<td>15</td>
</tr>
<tr>
<td>3.3.3 Yamir – router</td>
<td>16</td>
</tr>
<tr>
<td>3.3.4 kyamir - kernel module</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Administration</td>
<td>18</td>
</tr>
<tr>
<td>3.5 SIP</td>
<td>18</td>
</tr>
<tr>
<td>3.6 Architecture Diagram</td>
<td>19</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>20</td>
</tr>
<tr>
<td>Implementation</td>
<td>20</td>
</tr>
<tr>
<td>4.1 Prerequisites</td>
<td>20</td>
</tr>
<tr>
<td>4.2 kyamir.ko - kernel module</td>
<td>20</td>
</tr>
<tr>
<td>Linux/Android issues</td>
<td>22</td>
</tr>
<tr>
<td>4.3 yamird - userpace routing daemon</td>
<td>22</td>
</tr>
<tr>
<td>4.3.1 dymo protocol</td>
<td>23</td>
</tr>
<tr>
<td>4.3.2 netlink interface</td>
<td>26</td>
</tr>
<tr>
<td>4.3.3 rtnetlink - kernel routing interface</td>
<td>27</td>
</tr>
<tr>
<td>4.3.3 Timers</td>
<td>28</td>
</tr>
<tr>
<td>4.3.4 Issues</td>
<td>29</td>
</tr>
<tr>
<td>4.3 Administration</td>
<td>30</td>
</tr>
<tr>
<td>4.3.1 Backend - shell scripts and binaries</td>
<td>30</td>
</tr>
<tr>
<td>4.3.2 Frontend - GUI</td>
<td>31</td>
</tr>
<tr>
<td>4.4 SIP Application</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>33</td>
</tr>
<tr>
<td>Conclusions and future Work</td>
<td>33</td>
</tr>
<tr>
<td>Bibliography</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This project presents an alternative to using a fixed infrastructure for communication using a Mobile Ad Hoc Network (MANET). A MANET is a self-organizing network composed of nodes that are both free to move around as well as join or leave the network at will. This project aims to demonstrate that by using a MANET architecture a free and independent telecommunications network can be built out of consumer devices such as smart-phones and laptops. This chapter will examine the motivation behind the project, its objectives and provide an overview of the rest of the paper.

1.1 Motivation

The most visible form of modern day communications these days is the mobile phone which is increasingly becoming more of a mobile data link to the internet rather than something just to make voice calls with. And yet even though the mobile phone is a wireless device it still exhibits one characteristic in common with the land-line based telephone service (POTS) it is intended to replace namely that it requires access to a large centralized infrastructure that is neither cheap to build or free to use.

In urban areas where there are sufficient numbers of people who willing to pay for a mobile phone service the cost of building and maintaining such an infrastructure can be justified but for developing society or remote or sparsely populated regions the technical difficulties or the economic return may be to poor or resulting in a limited or simply non existent network infrastructure. Even in urban areas where fixed network coverage is normally adequate it can suffer a degraded or complete loss of service due to recent disaster or even by a large concentrations of people attending social gathering such as a sporting or music event.

A MANET allows a data network to be quickly setup in areas where a fixed infrastructure is unavailable or the cost is prohibitive. As they are self organizing they do not require the same level of maintenance required of fixed infrastructures. They can be easily extended and the failure of one or more nodes does not necessarily mean the failure of the entire network.

Emergency services can make use of MANET powered walk-talkie [1] where existing infrastructure has been destroyed or is unavailable due to power loss. MANETS also have military applications where the rapid deployment of a resilient and trusted mobile communications network is vital for any operation. Such organization can afford to have specialist equipment built to order that can run MANETS but the widespread availability of consumer devices with wireless connectivity now make it possible for others to build MANETS as well. For example a local community in developing society could setup their own local phone network and share data using older laptops and phones using a MANET[2].

5
1.2 Objectives

This project hopes to achieve the following:

- Explain the technologies behind MANET
- Discuss the problem experiences by MANETS and various solutions
- Examine related work
- Design and implement a functioning MANET
- Conclusions and further work

1.3 Project Outline

The remained of the report is as follows. Chapter 2 explores the problems involved in building and running a MANET, possible solutions and other related work. Chapter 3 proposes a design for building a MANET outlining the required components, algorithms and architecture. Chapter 4 describes in depth how the design was implemented as well as any issues encountered. Chapter 5 will examine the results of the implementation, make some concluding remarks and discuss any future work.
Chapter 2

Background and Related work

MANETS as described earlier can be formally described as a communication network using wireless links where the nodes or devices can be mobile and not fixed to any type of infrastructure. Node are not just the source or destination of data traffic but can also act as intermediaries relaying traffic on behalf of nodes that are not within wireless range of each other. This chapter will look first at 4 problem areas that effect MANET's namely wireless access, routing, address configuration and service discovery. It will examine briefly how voice calls can be made over a MANET and the look at other related ad-hoc technologies and implementations.

2.1 Wireless Access

For node to communicate with one another they need a wireless protocol. The 2 most common found on both laptops and smartphones are Bluetooth and WiFi.

Bluetooth was developed as a cable replacement technology for connecting electronic devices such as headsets and keyboards over short range low power radio interface. The basic unit of a Bluetooth setup is a piconet which is made up a master node and up to 7 slave nodes with a theoretical maximum range of up to 100m depending on the class of master. More than one piconet can exist in the same area and they can be connected together via a bridge node forming an interconnected piconet known as a scatternet. Piconets are a form of ad-hoc networks as they can operate without any fixed infrastructure so long as one device is willing to be Master. Although the master can communicate directly with any slave, slaves can only communicate with the master and not directly with each other even if they are within wireless range. Although class 1 Bluetooth allows a theoretical wireless range of 100m most consumer devices such as mobile phones only support class 2 which limits their maximum range to 10m. The maximum data throughput of Bluetooth radio itself is around 2.1 M/s although the high-speed update introduced in version 3 of the protocol can boost this up to 24 Mbps by using 802.11 wlan radio to actually carry the data.

IEEE 802.11 or WiFi was originally developed to allows computers such as laptops to connect to existing LANs without the need for network cables. In 802.11 the basic unit is called a BSS or Basic service set which represents a collection of network elements known as stations (STA) and access points (AP). 2 types of networks are supported namely infrastrafe mode and independent mode also known as IBSS. With infrastructure mode the stations within a BSS all associate themselves to an access point (AP) which they use to communicate with each other or with another network. In independent or ad-hoc mode stations do not use an access point and can communicate directly with each other so long as they are within wireless range. Theoretical wireless range for stock antennas varies between 70 to 250m depending on transmission method (a/b/g/n). Although 802.11n standard supports transmission rates up to 300Mbps the IEEE standard specifies that ad-hoc mode only needs to support 11Mbps.
Wi-Fi Direct [3] is recently introduced extension by the Wi-Fi alliance to allow peer-to-peer connections between consumer devices. Wi-Fi direct introduces a group owner and client devices to a 802.11 network. A group owner allows 1 to many links to it in manner similar to a Bluetooth master enabling it to setup an ad-hoc network with slave devices. In addition group owners can talk to each other directly forming a true peer to peer network. For older legacy devices the group owner can act as a soft access point allowing them to associate with and form an infrastructure BSS.

### 2.2 Addressing

Nodes need to be assigned an address when they join a network. In IP based routing these address need to be unique for data packets to reach their intended destinations. When the number of nodes is small a fixed or static ip address assignment can be used in private or public networks[4]. Fixed address can greatly simply address configuration and service location as nodes do not depend on a discovery mechanism to begin communication. For some services fixed address may be necessary such as in the case of the Domain Name System (DNS) where nodes need to know the ip address of an internet name server to do address lookups.

The more usual case is for dynamic or auto address configuration. For a wired or infrastructure based wireless networks a fixed server or is assigned the the task of handing out a unique address from a pool of available addresses. In the case of MANET where nodes can rapidly and randomly join and leave the network a dynamic address configuration protocol must be in place to quickly ensure that addresses are unique and valid for the length of a nodes presence in the network and to cater for any network partitioning and merging.

Dynamic address configuration protocols can be divided up into 2 categories: Stateful and Stateless configuration [5].

Stateful address configuration is where nodes keep track of the state of the network by maintaining a table of available addresses. Because they keep track of the network address state they are considered conflict-free.

Stateful protocols involve clients nodes broadcasting address requests which are then responded to by other server nodes. In fixed networks the Dynamic Host Configuration Protocol (DHCP) is the most common example. However using one central server for address assignment is not appropriate for a MANET as if the node becomes unreachable the entire network can fail.

For MANETs one often quoted examples is MANETConf which uses distributed allocation scheme [6]. In ManetConf each network nodes keeps a list of addresses in use. When a new client node requests an address via a broadcast it will choose the first server that responds and then request from it a unique address. Before the server responds it will send a network broadcast to check if the chosen address is available for use and then wait for acknowledgment from all known nodes. Network partitions are detected if no acks come from addresses marked as in use. If the server receives a positive ack from all other nodes it will assign the address to the client node and then broadcast this fact throughout the network so that the rest of the node keep their tables updated.Stateful schemes such as
MANETConf have the disadvantage where the flooding of control traffic (address config) to network can reduce the available bandwidth for actual data traffic.

Stateless address configuration node manage their own ip address instead of maintaining an address allocation table. They initially select their own address either randomly from a known range or use an algorithm involving some unique attribute such as their MAC address. They then use a duplicate address detection (DAD) process to check if the chosen address is actually unique.

An example of a stateless address protocol in fixed infrastructure is the Stateless Address Auto-Configuration (SLAAC) used in IPV6. In SLAAC a client uses its MAC address to create an ipv6 address which it then broadcast over the network interface. This requires that all nodes are within a single hop and the generated address is unique. If a duplicate address is detected then address configuration fails and it has to be manually configured. This is unsuitable for MANETS as which are multi-hop based and address conflict is a distinct possibility.

Instead nodes can employ 1 of 3 schemes to ensure conflict free address allocation.

In strong or query based DAD a node will check if a selected address is use by using a broadcast or sending an ICMP message to the selected address. If no response is received the address is considered free to use.
In weak DAD a nodes can tolerate duplicate addresses for a time by generating an additional key to use in conjunction with the chosen ip address. If the routing protocol detects that 2 different keys in routing message have the same ip address then it can prevent packets being sent to the duplicate address and optionally inform them of the conflict.
In Passive DAD nodes simply monitor passing routing requests for hints about what address are in use.

2.3 Routing

In an ad-hoc network where all the nodes are within wireless range of each other they can communicate directly by just transmitting over the wireless medium n effect using it to do single hop routing. When nodes are not within range of each other they must use their neighbors (if they are willing) to forward the packets on their behalf towards their intended destination. In such a multi-hop environments a routing protocol is required to establish how nodes forward traffic from source to destination. The distance vector and link state are examples of routing protocols used in wired networks which designed for use in static environments where resources such as bandwidth, memory and processing power are relatively cheap compared to nodes in a wireless ad-hoc environment.

In manets the highly dynamic nature of the network topology where node can appear and disappear randomly combined with fluctuating quality of the wireless links between make the links state and distance vector wired routing protocols unsuitable for use in MANETS[7].

The IETF manet working group has divided routing protocols into 2 main categories: proactive and reactive routing protocols [8].
Proactive or table driven protocols are where nodes try to maintain routing information for the entire network topology by periodically exchanging routing information usually by flooding the network with update requests.

They are essentially derived forms of the distance-vector or link state algorithms used in wired networks but optimized for MANET environment. Their main advantage is that there is no network delay for any network traffic for routes that have been established. Their main disadvantage is that in a highly unstable network they may risk saturating the network with control or routing information at the expense of actual data traffic. One well known example is Optimized Link State Routing (OLSR) [9]. It is a modified version of OSPF (Open Shortest Path First) where instead of flooding the network with routing updates a node selects a subset of its one-hop neighbors known as multi-point relays (MPR's) where link quality is judged best. It is these MPR's that perform routing and routing updates on behalf of the entire network.

Reactive or on-demand routing protocols do not maintain the network topology. Instead they only discover and maintain routes between nodes that are actually needed. Their main advantage is that they require much less resources than proactive protocols (memory, CPU) and perform better at route maintenance when the network is unstable. Their main disadvantage is that applications may experience long delays while routing discovery is taking place.

3 well known examples are Dynamic Source Routing (DSR), AdHoc On-Demand Distance Vector (AODV) and Dynamic MANET On-demand (DMYO) [10]. Although there are differences all 3 use the same concepts of route discovery and route maintenance. In route discovery mode an initiator will broadcast a routing request message to its neighbors who in turn will broadcast it to theirs until it reaches the target node which will then send a route reply message back to the initiator usually by the path the request traveled. In route maintenance if a node receives a data packet for an unknown address or a route that is marked as broken it will send out a route error message for that address. All nodes that receive the request error and consider it valid will mark any routes that use that address as unusable.

2.4 Service Discovery

For MANET's using dynamic or auto-address configuration nodes need to be able to establish the address of other to make contact or make use of a particular service [11].

For example when making a SIP or voice over IP call (VOIP) where nodes have dynamically assigned address the called party or target must register its address with a server that is accessible to the caller so that it knows where to send the VOIP messages to. To locate the server a node needs to make use of a service discovery protocol (SDP).

Examples in the wired world include Jini and UDDI and UPnP for wireless devices. All such protocols involve periodic flooding of the network with advertisements by service providers nodes, discovery requests attempts by clients nodes or both. Nodes may also choose to listen to SDP for relevant data.

SDP's can be divided into Directory based or Directory less architectures.
A directory based architecture involves nodes containing a list of all known services available in the network. This list may be centralized to one node or distributed across several. Directory-less involves all nodes broadcasting service discovery request to locate a resource.

To reduce the number of broadcasts some directory-based SDP's designed for MANETS attempt to piggyback the service information onto the routing protocol messages or extend the routing protocol itself. In the SipHoc protocol [12] all SIP address registration by client applications are accepted by a proxy running on the local device. Then as routing messages are being sent out it piggybacks some extra service information which inform other SipHoc nodes of the address.

An example of a directory less SDP for emergency response operations is [13] which uses location awareness along with client broadcasts to locate a needed service and to reserve it for use. Each part of the MANET is divided into geographical zones and resources such as emergency personal and equipment each carrying or connected to a PDA. Resource are assigned service categories such as surgeon,nurse,driver,supplies etc. Service requests are the classified according to type and need such as severe ,medium, and low. When a client issues a service request its is simply broadcast through the network and matching provides send back responses detailing location an availability. If a service cannot be find in the current zone of the requester then nearby zone are tried. The protocol can then issue a reservation request for the closest match and service personnel is informed or the equipment is reserved for use.

2.5 Voice over IP

To make a call over a MANET a voice over ip (VOIP) service needs to be run on both the sender and recipient nodes. One open protocol that can be used is the Session Initiation Protocol (SIP) [14]. SIP was developed by the IETF as an a simpler alternative to ITU's H323 protocol. It is modeled after HTTP and uses the same text based MIME encoding used in email messages. The protocol can either use UDP datagrams or TCP for its network transport layer.

In SIP callers and callles are known as user agents (UA). In the simple case where the calling party knows the IP address of the called party a UA can start a a voice call by sending an invite message directly to the calle UA. The called party will then respond with a ringing message and then an okay message (when the user picks up) which the caller then acknowledges. A media session is then setup between both parties allowing them to exchange voice traffic Either party can hang up with a BYE message which the other acknowledges.

In the case where the UA does not know the address of the called party it must contact a SIP proxy or redirect server to perform the call setup or return the called party's ip address. In either case the called party must have registered its own ip address with another proxy that can be contacted by the callers server.
2.6 Related Work

We introduce here wireless mesh networks which are closely related to MANETS and the Serval project which provides mesh based phone networks.

2.6.1 Wireless Mesh Network

Wireless Mesh Networks (WMN) are another type of wireless ad hoc network similar to MANETS but without the problems of constantly moving nodes. The IEEE have introduced a draft standard 802.11s [15] that adds mesh support to existing 802.11 networks by adding multi-hop frame forwarding and path-selection capabilities at the data link or MAC layer. This solves many of the problems experience by existing WMN or ad-hoc implementations that do multi-hop packet forwarding and routing at network or IP layer. IP layer ad-hoc protocols have a indirect view of the radio environment based on IP packet monitoring whereas a MAC layer implementation can maintain very accurate link metrics as well as maintaining quality of service and congestion controls for the radio interface.

802.11s introduces the concept of a Mesh Basic Service Set (MBSS) which consists of 3 network elements a Mesh Point or Station (MP), a Mesh Access Point (MAP) and a Mesh Point Portal MPP. A Mesh Point implements the MAC layer multi-hop forwarding and path discovery. A Mesh Access Point is a normal 802.11 AP with Mesh capability and a MPP acts as a direct bridge to other non 802.11 networks. The path selection algorithm employed is called Hybrid Wireless Mesh Protocol and supports both a proactive tree based method as well as reactive (on-demand) method based on the MANET AODV protocol.

Although a draft protocol 802.11s has already been used by the OLPC foundation as the underlying technology for mesh networks formed by their XO laptops which are for use in developing countries.

2.6.2 The Serval Project

The Serval Project is an mesh based phone network developed by a team from the Australian Flinders University [16][17]. The focus of the project is provide a phone system that does not require any kind of infrastructure and is free to use allowing its use in remote or disaster areas and by people in developing countries who cannot afford access to a fixed infrastructure.

The project uses android based mobile phones running a sip client (sipdroid) and can make use two IP based proactive routing protocols either the manet OLSR or the Better Approach To Mobile Ad Hoc Routing (BATMAN). The BATMAN protocol was developed by the Freifunk community based in Germany in response to perceived shortcoming to OLSR protocol ranging from its complexity to its actual performance in real world scenarios outside of simulators [18][19]. The protocol can be described as based on the idea of stigmergy exhibited in ants and termites where traces left behind by actions of one agent change the behavior of others to the benefit of the whole group. The protocol involved nodes broadcasting hello or originator messages (OGM) to their neighbors. The OGM
consisting of an originator address, sequence number and node sender address which a neighbor will change to its own before rebroadcasting. Upon receipt of its own message (identified by originator address) a node can judge by its reception speed and frequency which neighboring links are the best hop for sending data. A more recent version of the BATMAN protocol called batman-adv (not used by Serval) was developed which is now located in the data link layer for the same quality of server reasons behind 802.11s [20].
Chapter 3

Design Overview

Building a MANET from scratch requires a large number of elements incorporating many features discussed in the previous chapter such as Wireless Access, Addressing, Routing and Service Discovery, including some administrative tools and applications that make use of it. As such it was decided that only a subset of these features could be realistically built in given the time available and so fixed addressing was used for the MANET removing the need for address auto-configuration and service discovery. This then leads to a system design involving Wireless Access, Routing, Administrative tools, a VOIP application as well as a choice of what platform to run it on.

3.1 Platform

We choose Linux as our target platform as some initial research suggested some fairly low-level network stack operations where going to be required requiring access to the kernel source. This meant that code could be fully tested on some laptops before integration onto some smartphones. Choosing Linux meant Android was the default smartphone choice as these are widespread ARM based platform running the Linux kernel.

3.2 Wireless Access

As outlined in chapter 2 when it comes to laptops or smartphones the choice of wireless technology comes down to Bluetooth or IEEE 802.11. Although Bluetooth is available on most mobile handsets is not a standard option on laptops. The WDM 802.11s was considered outside the project scope as the project is study of IEFT MANET which are layer 3 or IP based and WiFi Direct is only available on some recent high-end handsets leaving 802.11 IBSS or ad hoc mode as our wireless access choice.

3.3 Routing

Routing in MANET’s can be divided into 3 areas, packet forwarding, route discovery and route maintenance. The Linux kernel takes care of packet forwarding as it simply consults its routing tables to decide what network interface to send a packet out on. It is the setting up and maintenance of these kernel routing tables that is essentially the task of a MANET router which it does with the help of an IP based route discovery and maintenance protocol. For a MANET router to operate correctly though it must first be able to monitor IP activity in the Linux network stack. This gives rise to 2 necessary components for our routing solution: a network stack monitor and a routing protocol implementation.

3.3.1 Netfilter - Linux network stack monitor

The Linux provides an IP packet filtering framework called Netfilter for monitoring its network stack[21]. This framework provides hooks where user defined functions can be called as packets traverse the network stack. Netfilter defines 5 stages where a used defined function can be called.
1. **NF_INET_PRE_ROUTING** is where packets have just arrived into the stack before any routing decisions have been made.
2. **NF_INET_LOCAL_IN** called after routing has decided if a packet is destined for a local process.
3. **NF_INET_FORWARD** is called if a packet is to be sent out onto another network interface.
4. **NF_INET_LOCAL_OUT** is called when a local process is sending a packet before any routing decision is made.
5. **NF_INET_POST_ROUTING** is called after all routing decisions have been made and the packet is being out of the stack.

When a user defined function is called they can return one of the following return codes to the netfilter framework.

- **NF_ACCEPT** - to keep the packet and continue traversal.
- **NF_DROP** - discard the packet and stop traversal.
- **NF_STOLEN** - tells netfilter the function has taken control of the packet and stop traversal
- **NF_REPEAT** - tells netfilter to call this hook again.

### 3.3.2 DYMO - A Manet IP routing protocol

Because our router has to run on resource constrained nodes such as smartphones we opted for a reactive routing protocol. We chose to implement DYMO (Dynamic MANET On-Demand/draft 21) routing protocol because its the most recent reactive protocol produced by the IETF manet working group and is described as being simpler to implement than AODV[22]. As mentioned earlier DYMO can be divided into two protocol operations: route discovery and route maintenance.

Route Discovery mode is triggered when a data packet needs to sent to an IP destination address (daddr) for which there is no entry in the routing table.

A route request message (RREQ) is constructed by the router containing the target address set to the daddr and nodes own ip address called the origin. This packet also contains a hop limit and a sequence number which are used to detect duplicate or stale requests and to limit network flooding. This request is then encoded using the binary MANET encoding format also known as packetbb [23]. The encoded request is then sent out as a UDP packet with the source IP address set to the local node, the destination port set to 269 and destination address set to the manet link-local multicast address 224.0.0.109.

The router then sets a timer and waits for a reply. All routers that receive the request will after ensuring its not a duplicate, invalid or stale will update their routing tables to indicate that IP traffic can be routed back to the origin using the the ip source address of the udp packet that the request arrived in on. If the receiving node local adders does not match the target address it will relay the request using another link-local udp multicast after first checking the hop limit has not been exceeded (in which cast it drops the request). This is the method by which a data route is built for the original destination address (daddr) that triggered the route discovery.
If the receiving node local address match's the target address it will create a reply message (RREP) with the target address set to the request origin and the origin address to its own local address which should be the same the request target address. It will then send this back as a manet encoded udp packet with the destination address set to one of of its available routes, which is usually the ip address of the udp packet that the request arrived in on. This will then be unicast relayed back by intermediate nodes using an appropriate route (usually the one establish by RREQ) back to the origin node which upon reception will cancel its timer, add a new route to its table and allow the original data packet to be forwarded using the new route. In the event of no reply the timer will expire and the router will attempt a number of retries which if all unsuccessful will cause the original IP packet to be dropped and an ICMP destination unreachable error send to the application that tried to send the packet.

Route maintenance is triggered when routes expire due to non-use or if a router receiving a data packet for which it has no route. All route entries have an accompanying timer which gets reset every time they are used by a data packets. Upon actual expiry the entry will be discarded from the routing table. If a router received a data packet for which there is no route it will discard the packet and send a route error (RRER) message recording the unreachable address to the the manet multicast address. All routers that receive the multicast will remove any routes that contain the unreachable address as the next hop.

### 3.3.3 Yamir – router

YAMIR (Yet Another Manet IP-based Router) is the core of our project. The routing component is composed of 2 parts a kernel module called kyamir which makes use of the netfilter framework and and a userspace daemon called yamird which implements the routing protocol. We decided on the kernel module approach to network stack monitoring even though there is a userspace variant called libnetfilter_queue. As the userpace version would have required all IP network traffic to be sent up to userspace we felt this would have negatively impacted the performance of any real time traffic such as SIP. For communication between the kernel modules and userspace code yamir makes use of the Linux Netlink socket interface which allows full-duplex communication between them.

### 3.3.4 kyamir - kernel module

Kyamir is the our kernel module that plugs into the linux kernels netfilter framework. It can be divided into a netfilter and command section.

kyamir makes use of the 3 netfilter hooks NF_INET_PRE_ROUTING, NF_INET_LOCAL_OUT, NF_INET_POST_ROUTING.

In the NF_INET_PRE_ROUTING hook a message is sent to yamird indicating the source ip address (saddr) is in use. If a route entry for the source address kyamir allows the IP packet to process (NF_ACCEPT) else it sends a message to yamird indicating a routing error and tell netfilter to drop the packet with a NF_DROP response code.

The NF_INET_LOCAL_OUT hook is used to initiate route discovery for locally created
data packets. If a route exists for the packets destination address (daddr) it is allowed to proceed else we details of the packet are added to a queue and netfilter is told to stop processing it with a response code of NF_STOLEN. If the queue was previously empty for that address a message is sent to yamird that a route is needed.

The NF_INET_POST_ROUTING will send message to yamird that the destination address (daddr) is in use.

kyamir supports 3 commands which can be sent from yamird via netlink:

• **route-notfound** - sent when a route discovery fails it. kyamir will discard any queued packets for the given address and send an ICMP destination unreachable to the original destination.
• **route-add** - sent when a new route has been added for the given address. kyamir will inject any queued ip packets back into the network stack.
• **route-del** - this is sent by yamir when a route has been deleted. kyamir will discard any packets as per a route-not-found message.

3.3.5 yamird - userspace routing daemon

yamird is the userspace router that implements the DYMO protocol and maintains the kernel routing tables. It can be divided up into 3 areas: the command-interface, the dymo-protocol and route-table maintenance.

command-interface

yamir supports the following commands which kyamir can send via the netlink interface.

• **route-need** - yamir will initiate route discovery for the given address. If route discovery is successful it will update the route table and send a route-add message to kyamir else it will send a route-notfound message.
• **route-inuse** - yamir will restart any timer for the route associated with the given address.
• **route-err** - yamir will send a dymo route error message (RERR) to the network

dymo-protocol

This deals with the dymo protocol as described in the draft rfc and whose operation was described earlier. The following protocol messages are supported:

• **RREQ** - any valid routing info be used to create or update route table entries. RREQ will be relayed or a RREP created if the target
• **RREP** - route updates as per RREQ. And RREP will be relayed towards destination if not the target.
• **RERR** - all routes used by unreachable node will be discarded. The message will be relayed if hop count not exceeded.
route-table

The route table has an entry for all reachable destination address via this node and supports the following commands:

- **route-update** insert a new entry or simply update an existing one. A new route will also trigger a kernel route update.
- **route-delete** will discard the route and remove the associated kernel route.
- **route-find** use the longest matching prefix algorithm as described rfc1812 required by the dymo protocol

3.4 Administration

Ad-hoc network configuration on a linux laptop is straightforward as the preinstalled network tools such as command line version such as iwconfig and ifconfig (or the newer ip link) can used to switch a network interfaces into ad-hoc mode following which the kernel module can be loaded and the router started.

For Android smartphones it is not quite so straightforward. The Wifi Manager on android handsets filters out all IBSS or ad-hoc networks from the scan results reported by wpa_supplicant making it impossible to configure 801.11 ad-hoc mode using the standard android framework. Various solutions to the problem have been proposed ranging from modifying the wpa_supplicant tool to report IBSS networks as infrastructure access points to actually forcing wpa_supplicant to associate to an ad-hoc network and then try to prevent Wifi manager from changing it. As our manet requires a permanent reliable network connection on each node we opted for an alternative solution which effectively takes charge of the network interface while the manet is running preventing the Wifi manager and hence other android apps from interfering with it.

The administration tool is an android app made up of 2 parts a backend comprising a set of shell scripts and binaries needed for starting and stopping the manet and a gui frontend for both entering network settings and controlling the Manet.

3.5 SIP

3 of the most popular sip clients for android are SipDroid, cSipSimple, Linphone. We choose Linphone because they provide versions that work on both android mobiles as well as linux workstations and provide the source code for both allowing any changes required to make use of the MANET.
1. Sip Client - LinPhone.
2. Admin Gui - Android Application
3. Admin Backend – yamr.sh, yamird and kyamr.ko
Chapter 4

Implementation

This chapter covers the implementation of yamir on both laptops and android smartphone. We will first briefly cover prerequisites for our implementation and then examine each of our components (kyamir, yamir, config, sip) in turn noting where appropriate issues specific to a Linux workstation or android smartphone.

4.1 Prerequisites

Building the project requires a Linux workstation running a recent Linux kernel with both the gcc and kernel development system installed. On our build platform (Fedora 16 laptop running a 3.3 kernel) this means gcc and kernel-devel rpms. For the android app development the android sdk and if you're using eclipse for you're IDE the ADT plugin as well[24]. Compiling code natively for an android handset requires an ARM tool-chain which includes things such as a compiler, linker, header files and libc. Although you can roll you're own most now use either the Android NDK or CodeSourcery cross compilers which allow people to build ARM binaries on their own workstation rather than the on the handsets [24-25].

As yamir needs to make changes to the network interface it requires direct access to the linux kernel. To to this it require a handset that is rooted. Rooting involves changing the phones firmware from the default that was shipped to a version that allows application be run with Linux superuser access. The process is different for every handset but an internet search or look at http://androidforums.com quickly turns up the appropriate guide.

4.2 kyamir.ko - kernel module

As mentioned earlier kernel module is a netfilter plugin that uses the NF_INET_PRE_ROUTING, NF_INET_LOCAL_OUT, NF_INET_POST_ROUTING hooks for monitoring the network stack. The module takes one command line parameter ifnames which is a comma separating list of ipv4 network interfaces or device names that it should monitor for MANET data traffic.

e.g.

insmod kyamir.ko ifnames=eth0

On startup the module creates a queue for storing in ip packets state information waiting for route discovery. A queue entry is defined as follows:

```c
struct queue_packet {
    struct list_head list;
    struct sk_buff *skb;
    __be32 addr;
};
```
int (*okfn)(struct sk_buff *);
}

static DEFINE_RWLOCK(queue_lock);
static LIST_HEAD(queue_list);
static unsigned int queue_total;

The skb and okfn functions pointers are given to us by netfilter and are kernel buffer containing the raw ip packet and the callback function to reinject the packet back into the network stack. The addr is the destination address (daddr) taken from the ipv4 header. All read and write access to the queue is guarded by queue_lock which is a kernel spinlock.

The module also creates a route list define as follows:

struct route_entry {
    struct list_head list;
    __be32 addr;
};

static DEFINE_RWLOCK(route_lock);
static LIST_HEAD(route_list);
static unsigned int route_total;

We need maintain a separate routes list inside the module as our understanding is that you cannot access the kernel routing tables directly from another module without creating another netlink socket which would add unwanted latency to the ip packet filtering.

struct dymo_device {
    char ifname[IFNAMSIZ];
    int vaddr;
    int ifindex;
    __be32 address;
    __be32 mask;
    __be32 broadcast;
};

For each name given in the ifnames list arg it looks up the corresponding entry in the kernels device list and loads a table entry as show above for that the device. Note changing the network interface requires the module to be reloaded.

Finally the module setup its netlink and netfilter hooks via calls to netlink_kernel_create() and nf_register_hook().

The netfilter entry point for yamir_nf_hook() and consists of a switch statement implementing the logic described in chapter 3 for the 3 hooks NF_INET_PRE_ROUTING, NF_INET_LOCAL_OUT, NF_INET_POST_ROUTING with the addition of some code that always return NF_ACCEPT if:

- the IP header daddr is a global broadcast or multicast address
- the UDO source or destination port is DYMO 269
- the device is not in the list specified at start up
- the data packet address is the devices broadcast address
Linux/Android issues

Build kernel modules requires the kernel source headers to be installed and a Makefile entry that invokes the kernels kbuild system. For linux workstation the kernel headers are usually located in /lib/modules and the standard kbuild mechanism is invoked according to the kernel

For handsets the kernel source code need to be located that matches that version used by the handset. These can be downloaded from manufacturers websites but there are issues. To compile a handset kernel the compiler flags, kernel config and Makefile settings must be correct or kernel modules will not load.

The gingerbread kernel source provided by Samsung for the gt-i9100 model contains a default config file that had settings different to the one used for production handsets.

Although htc desires handsets have a readable /proc/config.gz the Froyo kernel source contains a Makefile with a EXTRAVERSION var that needs to be set to the same value as used on the phone. After fixing these problems the handset kernel then need to be cross compiled using an ARMtool-chain as mentioned in the prerequisites. Once the handset kernel headers have been generated kyamir needs to be cross compiled for each handset.

4.3 yamird - userpace routing daemon

As mentioned in chapter 3 yamird implements the route discovery and maintenance using the manet DYMO protocol. It is a single threaded process that uses select based i/o multiplexing to handle all dymo and netlink traffic. We choose such a design because its essentially i/o bound router waiting for dymo packets or netlink events from kyamir. Although the the newer epoll() method could have been used we felt this was unnecessary for a system with just 3 open file descriptors. As timers are also being used during the select() loop the implementation can be described as using an asynchronous design where all code is triggered via callback functions by the occurrence of i/o or timer events.

As yamird is quite a large program incorporating some very low level networking code we will only highlight the main logic and algorithm implementation referring the reader back to the chapter 3 for its design logic and the appendix for the full code.

yamird requires the name of the network interface that the manet is to be run on. Optionally it can be started in daemon mode and with logging (to stderr) turned on.

e.g.

```
./yamird -dl eth0
```

On startup the program will create 3 interfaces for the dymo protocol, the netlink to kyamir and rtnetlink to the kernel routing tables. In addition it also initializes a software timer facility required for the routing protocol.

The program will then enter a select driven i/o loop which will wait until either a i/o packet is detected or a timer expires as can be seen below.
max_fd = MAX(0, dymo_socket);
max_fd = MAX(max_fd, yamir_socket);
max_fd = MAX(max_fd, route_socket);

while (keep_running) {
    timer_process(&wait);
    FD_ZERO(&rset);
    FD_SET(dymo_socket, &rset);
    FD_SET(yamir_socket, &rset);
    FD_SET(route_socket, &rset);
    ns = select(max_fd+1, &rset, NULL, NULL, &wait);
    if (ns != -1) {
        if (FD_ISSET(dymo_socket, &rset)) {
            dymo_recv(dymo_socket);
        }
        else if (FD_ISSET(yamir_socket, &rset)) {
            yamir_recv(yamir_socket);
        }
        else if (FD_ISSET(route_socket, &rset)) {
            route_recv(route_socket);
        }
    }
}

4.3.1 dymo protocol

yamird uses a standard datagram socket for the dymo protocol.

    fd = socket(AF_INET, SOCK_DGRAM, 0);

On creating the socket the program uses a number ioctl() system calls to retrieve information about the named network interface such as the ip address its is bound to. This is how yamird router discovers its own local address.

ioctl() must be called with with a file descriptor which can be an existing socket and the address of an ifreq structure. The alternative is to use the call getifaddrs() and scan the return list of matching network interface name but getifaddrs() is not supported by the NDK cross compiler for android so the direct ioctl() system call was used instead.

e.g.

    ec = ioctl(fd, SIOCGIFADDR, &ifreq);

We then set the a number of options on the socket as follows:

- IP_PKTINFO - request extra udp header state information
- IP_TTL     - set the udp time to live to 255 as per the protocol
- SO_REUSEADDR - allows yamir to rebind immediately to the dymo port
- SO_BINDTODEVICE - forces all packets on the socket to only egress the bound interface
- IP_ADD_MEMBERSHIP receive dymo packets from the link-local multicast
address 224.0.0.109.,
- IP_MULTICAST_LOOP turn off multicast loopback

On detecting a readable data on the dymo socket the dymo_recv() callback invoked. We use a version of recvfrom based on [27] which gives us the ip destination address and interface. This is necessary as the normal recv() variants do not return this information and we need this to establish if the UDP packet received on the dymo socket was sent to a unicast or multicast destination address when relaying dymo messages. The code then decodes the initial manet packet header followed by any dymo messages as can seen below.

```c
static void dymo_recv(int fd) {
    static unsigned char rbuf[RBUF_SIZE];
    struct pkt_buf pbuf, mbuf;
    struct pkt_hdr phdr;
    struct pkt_msg msg;
    ...
    nr = recvfrom_wstate(fd, rbuf, sizeof(rbuf), 0, &state);
    ...
    mapbuf(&pbuf, rbuf, nr);
    ...
    // decode all (unless pkt or msg hdr error)
    err = decode_pkt_hdr(&phdr, &pbuf);
    while (!err && pbuf.rem > 0) {
        msg_init(&msg);
        err = decode_msg_start(&msg, &mbuf, &pbuf);
        if (!err && decode_msg_rest(&msg, &mbuf) == 0) {
            switch(msg.type) {
                case DYMO_RREQ:
                case DYMO_RREP:
                    handle_rn(&msg, &state);
                    break;
                case DYMO_RERR:
                    handle_rerr(&msg, &state);
                    break;
                default:
                    LOG("Unknown type %d\n", msg.type);
                    break;
            }
        }
        msg_deinit(&msg);
    }
}
```

Reading and writing dymo messages stored inside a raw UDP packet requires a manet message (rfc5444) encoder and decoder. Unlike many binary network protocols simply casting a c structure pointer to a raw io buffer will not work as the message fields have both dynamic positions and sizes. We use a pkt_buf structure along with some supporting functions to decode and encode protocol primitives such as integer and arrays as well as compound structures.

```c
struct pkt_buf {
    uint8_t *data;
    int len;
    uint8_t *pos;
    int rem;
    enum PKT_FIELD err;
```
When decoding a message we simply load the pkt_buf with the address of the network packet data and its length from which we can then pull or read the message fields. We use macro helper functions which return appropriate error messages if there is not enough data in the buffer to extract the expected field. This makes the decoding logic easy to follow without being obscured with by a large number of nested if statements as the following code illustrates:

```c
#define UNPACK_FIELD(dst, buf, field, need) 
  if ((buf)->rem < (need)) { 
    return dec_err((buf), (field), (need)); 
  } 
  (dst) = unpack_int((buf)->pos, (need)); 
  (buf)->pos += (need); 
  (buf)->rem -= (need)
#endif
```

```c
#define UNPACK_INT1(dst, buf, field) UNPACK_FIELD(dst, buf, field, 1)
#define UNPACK_INT2(dst, buf, field) UNPACK_FIELD(dst, buf, field, 2)
```

```c
static int decode_pkt_tlv(struct pkt_tlv *tlv, struct pkt_buf *src)
{
  ...
  UNPACK_INT1(tlv->type, src, PF_TLV_TYPE);
  UNPACK_INT1(tlv->flags, src, PF_TLV_FLAGS);
  ...
}
```

Encoding is easier as we know exactly what the number and size of the fields we are going to place into the network buffer. The only snag are the length field for headers or compound structures which must be encoded into the network buffer before the data fields themselves. To do this we first mark the current encoding position in the network buffer and then reserve space for the size of length field without actually encoding it. Next we encode the header or compound structure into the network buffer. We then subtract the new encoding positron from the one we marked which then gives the length of the previously encoded
Finally we encode this length into position we originally marked.

```c
static void encode_pkt_hdr(struct pkt_buf *dst, struct pkt_hdr *hdr) {
    int value;
    value = (hdr->version & 0x0F) << 4;
    value |= (hdr->flags & 0x0F);
    pack_field(dst, value, 1);

    if (BITSET(hdr->flags, PHAS_SEQNUM)) {
        pack_field(dst, hdr->seqnum, 2);
    }

    if (BITSET(hdr->flags, PHAS_TLV)) {
        // encode tlv-block
        struct pkt_tlv *tlv;
        uint8_t *pos = make_space(dst, 2);
        for (tlv=CDL_NEXT(&hdr->tlvs); tlv != &hdr->tlvs; tlv=CDL_NEXT(tlv)) {
            encode_pkt_tlv(dst, tlv);
        }
        // set tlvs-length
        pack_int(pos, dst->pos - pos, 2);
    }
}
```

4.3.2 netlink interface

We create the interface to our kernel module using the netlink socket family and raw socket type. The macro `NETLINK_YAMIR` is an alias for `NETLINK_USERSOCK` defined in the system header file `netlink.h` and reserved for netlink user mode socket protocols.

```c
fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_YAMIR);
```

We then simply bind the netlink socket to our address which is yamird process number. netlink broadcast messages will then be received by yamir everytime a route event occurs in kymair.

```c
static struct sockaddr_nl yamir_addr;
...
// bind to address
memset(&yamir_addr, 0, sizeof(yamir_addr));
yamir_addr.nl_family = AF_NETLINK;
yamir_addr.nl_pid = getpid();
yamir_addr.nl_groups = NETLINK_YAMIR_GROUP;
ec = bind(fd, (struct sockaddr *) &yamir_addr, sizeof(yamir_addr));
```

When a write event is detected by the `select()` the `yamir_recv()` callback will be invoked as can be seen below (error check code removed for clarity):

```c
static void yamir_recv(int fd)
{
    static uint8_t rbuf[NLMSG_SPACE(sizeof(struct yamir_msg))];
```
struct sockaddr_nl addr;
socklen_t addr_len = sizeof(struct sockaddr_nl);
ssize_t nr;
struct nlmsghdr *hdr;
struct yamir_msg *msg;
size_t msg_len;

nr = recvfrom(fd, rbuf, sizeof(rbuf), 0, (struct sockaddr *) &addr, &addr_len);
....
hdr = (struct nlmsghdr *) rbuf;
msg_len = NLMSG_PAYLOAD(hdr, 0);
msg = NLMSG_DATA(hdr);

switch(hdr->nlmsg_type) {
case YAMIR_ROUTE_NEED:
    route_discover(msg->addr, msg->ifindex);
    break;
case YAMIR_ROUTE_INUSE:
    route_inuse(msg->addr, msg->ifindex);
    break;
case YAMIR_ROUTE_ERR:
    route_err(msg->addr, msg->ifindex);
    break;
case YAMIR_ROUTE_HIGH:
    route_high(msg->addr, msg->ifindex);
    break;
case YAMIR_ROUTE_KINDNOTSUPP:
    LOG("Unsupported netlink msg type %d\n", hdr->nlmsg_type);
    break;
}
}

4.3.3 rnetlink - kernel routing interface

Access to the kernel routing tables is via the same netlink api calls that are used to communicate for kyamir except the socket protocol is NETLINK_ROUTE.

fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

Updating a kernel route via rnetlink is somewhat involved but it can be summarized as creating and sending a netlink message with its type set to one of the following values:

- RTM_NEWROUTE - create a kernel route
- RTM_DELROUTE - delete a kernel route

And setting some or all of the following field attributes:

- RTA_DST - dest address for which a route needs to be setup
- RTA_PRIORITY - the next hop count (or metric)
- RTA_OIF - next hop interface index
- RTA_GATEWAY - next hop ip address

This is wrapped by calls to the route_send() function:

static int route_send(int type,
    uint32_t dest_addr,
Struct dymo_route {
    struct dymo_route *prev, *next;
    uint32_t addr;
    uint8_t prefix;
    int seqnum;
    uint32_t nexthop_addr;
    uint32_t nexthop_ifr;
    unsigned int flags;
    uint32_t dist;
    struct timeval timestamp;
    struct timer *age_timer;
    struct timer *seqnum_timer;
    struct timer *used_timer;
    struct timer *delete_timer;
};

Our routing table entry are defined follows:

And is updated by a call to update_route() every time a valid dymo request or reply is received. The code will only update an existing route if it deems it to be superior for fresher which is done by checking the routing message sequence number and distance values.

4.3.3 Timers

The dymo protocol mandates use a number of timers for both route expiry and request message timeouts. As the number of route and inflight request messages is indeterminate a timer facility is needed that can support potentially thousands of events. A software timer facility was implemented which uses a minium binary heap or priority queue algorithm.

The idea here is that a given a relative time offset an absolute timestamp can be calculate by added the current time and this value used to place a timer entry into correct position int the priority queue where the oldest timestamps entries bubble up towards the top as they approach their expiry time.
struct timer *t;
struct timeval now;
t = malloc(sizeof(*t));
t->cb_fn = cb_fn;
t->cb_arg = cb_arg;
gettimeofday(&now, NULL);
timeradd(&now, timeout, &t->when);
timerheap_push(heap, t);
return t;
}

route->used_timer = timer_add_wsec(used_timeout_cb, route, ROUTE_USED_TIMEOUT);

Before calling select() we first process the timer queue to both look for entries that have expired and to calculate the wait time to give to select() as can be seen below.

void timer_process(struct timeval *wait)
{
    struct timeval now;
    struct timer *t;

    gettimeofday(&now, NULL);

    while (1) {
        t = timerheap_min(heap);
        if (!t) break;
        if (timercmp(&t->when, &now, >)) break;
        timerheap_pop(heap);
        t->cb_fn(t->cb_arg);
        free(t);
    }

    if (t) {
        timersub(&t->when, &now, wait);
    } else {
        wait->tv_sec = 5;
        wait->tv_usec = 0;
    }
}

4.3.4 Issues

If a firewall is running on the linux workstation such as iptables it may have to be configured to allow ip multicasts to pass through to. Initially tests between 2 fedora laptops failed due to multicast packets not being delivered to the yamird process even though the
wireshark network protocol analyzer clearly showed the udp packet arriving on the correct interface. Subsequent investigation showed the iptables was the culprit. A quick way to test if the firewall is the culprit is to do a iptables -F which flushes the current ruleset. On fedora 16 the following iptables command allows manet multicast packets through:

    iptables -I INPUT 1 -p udp --dst "224.0.0.109" -j ACCEPT

Testing on HTC Desire handset using the stock rom we found that the dymo packets where not sent out on the wireless interface. The problem appears to be related to the firmware being loaded onto the bcm4329 chipset which drops ad-hoc traffic. The solution according to both the androidforums and xda-developers is to load the fw_bcm4329_ap.bin from a stock HTC Evo 4G instead onto the chipset instead of fw_bcm4329.bin.

Running yamird in userspace requires the following permissions:

- **cap_net_bind_service** uses privileged port 269
- **cap_net_raw** - uses SO_BINDTODEVICE
- **cap_net_admin** - uses netlink multicast

If the linux system support setcap this can be done as follows (requires root):

    setcap cap_net_bind_service,cap_net_raw,cap_net_admin=+ep yamird

### 4.3 Administration

As mentioned in chapter 3 the administration tool consists of 2 parts, a back end set of shell scripts and binaries for running the manet and a front end gui to control it. These are packaged together into an android application yamir-android.apk.

#### 4.3.1 Backend - shell scripts and binaries

This consists of a script called yamir.sh, the yamird routing dameleon and kyamir kernel module compiled for the ARM architecture. We also need a kernel module and shell script specific to each android handset as both the kernel versions and the method to start and stop the network interface differ for every handset. In addition the linux iwconfig (from wireless-tools) and killall (from psmisc) binaries need to be shipped as well as these are not standard on android handsets.

The yamir.sh script can be called with either a start or stop command line option and uses a setting file created by the gui.

The start option invokes the appropriate device script which first turns off the wifi service and then assigns the MANET ip address and netmask. The kernel module kyamir.ko is loaded (via insmod) and finally the yamir process is started in daemon mode. The stop option shuts down the yamir process, unloads the kyamir kernel module and then switch off ad hoc mode on the wireless interface via the device script.

All of the binaries and shell scripts are located in the android apps raw resouce folder where they must be first unpacked before use. This is because both the android SDK apk build tool and the handset apk installer allow native libraries but blacklist any executables.
4.3.2 Frontend - GUI

The gui or android app yamir consists of 2 files YamirActivity and SettingsActivity.

YamirActivity contains most of the control code for starting and stopping the manet. On first being loaded by android it locates the apps installtion dir and the location of the su tool which is needed to launch the backend scripts with root access. It then unpack the backend binaries and shell scripts into the apps files work area with the correct permissions.

On pressing the start button the code checks if the backend has been installed correctly and then writes out the apps settings to a config text file that can be read by the yamir.sh script. It then execs via su the yamir.sh script with the start option and the pathname of the config file and displays the results to the user in the status log. Pressing stop likewise calls yamir.sh with the stop and the previous config file path.

SettingsActivity is subclass of the android PreferenceActivity. It allows the following number of settings to be modified and saved to the apps preference file. The settings are:

- **Device** - The device type. Used to select both the device script and kernel module to load.
- **Network Interface** - the name of the network interface to use (defaults to eth0)
- **Network Name** - then ad-hoc SSID network name to associate with (defaults to yamir)
- **Channel** - The wireless channel to use (defaults to 6)
4.4 SIP Application

Initial tests of the manet using the Linphone sip client running on a number of linux laptops was successful with routes being setup and voice calls being received correctly. However on using the Linphone app on a samsung handset we found it would only intermittently use the connection. We discovered that the app was dependent on the handsets connectivity manager to report at least one available network connection before it enabled sip calls. The network connection was found not to be manet wifi but the 3G mobile connection. On switching off mobile network access confirmed this. After modified the linphone app backend (eXutils.c,misc.c) and frontend to make use of the MANNET we where subsequently able to make calls between the samsung and our laptops.

Using the modified version of Linphone we where also able to make calls between a pair of htc desires and our laptops. However we initially unable to use the samsung with htc's. The initial sip call setup traffic was being received by the samung but the corresponding reply's being sent from the Samsung (confirmed by wireshark) where being ignored by the htc. On closer examination it was found that the samung wirless sip traffic contained 802.1Q or VLAN tags in the ethernet frames which the htc's did not understand. The problem was quickly located to linphones backend modules (eXtl_udp.c,rtpsession_inet.c) setting the IP_TOS option on the UDP sockets it was using for sip and rtp traffic. Disabling this removed the VLAN ethernet frame extensions enabling sip calls between the samsung and the htcs.
Chapter 5

Conclusions and future Work

The primary goal of the project was to build a MANET which could be run on some smartphones which it achieved. However, the amount of work required to configure some android handsets to use it was far greater than what was originally envisioned. Building a fully functional MANET router is a major task in itself but the added complexities of maintaining separate kernel modules for each handset combined with the problems encountered in switching on ad-hoc mode made the project goal almost unachievable.

The main problem is that Android handsets do not normally support 802.11 ad-hoc mode. The requirement for a rooted handset together with the device specific changes that need to be made to enable ad-hoc mode makes maintaining a MANET feasible only in the lab if the exact handset and firmware cannot be established in advance. Even with existing handsets the MANET system can be easily disabled if the existing kernel or firmware is changed without updating the kernel module.

802.11 ad-hoc may however be soon displaced thanks to the recent Android 4.0 release which now contains direct support for WiFi Direct although its a little unclear yet what form of ah-hoc networks are supported.

However if both the kernel and handset are known and updates can be controlled then running a MANET is feasible if the firmware configuration problems can be solved.
Bibliography


22. dymo - Dynamic MANET On-demand (DYMO) Routing, daft-ietf-manet-dymo-21


