An Implementation and Evaluation of the Ad-Hoc On-Demand Distance Vector Routing Protocol for Windows CE

David West

A dissertation submitted to the University of Dublin, in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

September 15, 2003

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Date: September 15, 2003.

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Date: September 15, 2003.

Acknowledgements

I would especially like to thank my supervisor, Jim Dowling, for his enthusiasm and guidance during the course of this project.

I would like to thank Stefan Weber and Pete Barron for their help and keeping me supplied with the huge amount of additional hardware I requested!

I would also like to thank Daniel Prince and Joe Finney from Lancaster University for their help and insights during the project.

Finally, thanks to all my classmates and friends, and especially to Conor, for keeping me sane throughout.

Abstract

There are a number of implementations of the Ad-hoc On-demand Distance Vector (AODV) routing protocol available for the Linux platform, but not for any other platform. Development of ad-hoc routing protocols has been slow because current operating systems do not provide adequate direct system-services for their implementation. This dissertation presents a design, implementation, and evaluation of AODV for the Windows CE operating system. It discusses the features of the Windows protocol stack that can be used for implementing ad-hoc routing protocols generally, and shows that the AODV routing protocol can successfully be used in a heterogeneous ad-hoc network environment.

Table of Contents

Table of Figures

1 Introduction

This dissertation presents a design, implementation, and evaluation of the Ad-hoc On-Demand Distance Vector (AODV) **[8]** routing protocol for the Windows CE platform. The field of ad-hoc networks is an area of much active research at the moment. An ad-hoc network is one consisting of devices equipped with wireless interface cards, which come together to form multi-hop wireless networks dynamically and automatically, the network having a continuously changing topology due to node mobility. The AODV routing protocol is an on-demand, or reactive protocol that discovers and maintains routes to other nodes only as they are needed. It has been shown to have promising characteristics, including performance figures, in simulation studies **[16], [19], [35]** compared with other proposed ad-hoc routing protocols.

Field studies using the AODV routing protocol have thus far been limited to devices running the Linux operating system, as the current implementations of AODV have all been developed for that platform. Thus real-world testing of the protocol has been limited to homogenous network environments. The protocol has not been accessible to non-technical users of mobile devices, as the majority of such users are not familiar with the Linux operating system. Users will be able to install our version on their Windows CE mobile devices, giving them the ability to connect to any network running AODV. They may, for example, wish to communicate with other users during a meeting where no pre-existing infrastructure is in place, or they may want to access the services of a metropolitan ad-hoc network, such as the Wireless Ad-hoc Network for Dublin (WAND), which is soon to be deployed around the campus of Trinity College, and on the streets of Dublin.

Modern operating systems have been designed with static networks in mind, where the routing protocol is strictly separated from the packet forwarding function. In an on-demand ad-hoc network, these two processes are closely linked, as the routing

protocol must be able to handle situations where packets are to be forwarded to a previously unknown destination by initiating a route discovery cycle. The network protocol stacks of modern operating systems have not been designed to deal with this situation; they do not provide adequate system services for the implementation of ad-hoc routing protocols. This greatly complicates the implementation of on-demand ad-hoc routing protocols, and has slowed their development. The implementation strategy of existing ad-hoc protocols in Linux is examined in this dissertation. Most such protocols rely on the packet filtering and mangling architecture called Netfilter **[30]** to handle packets for an ad-hoc routing protocol.

Unfortunately the Windows protocol stack has no direct counterpart to the Netfilter framework. In this dissertation a survey of the packet-handling mechanisms in the Windows CE protocol stack is presented, and each mechanism is assessed for its suitability for implementing an on-demand ad-hoc routing protocol. It is hoped this work will be of great help to future Windows protocol implementers.

The implementation is evaluated for interoperability with other AODV implementations, specifically the existing Linux implementations. It is shown that devices running heterogeneous operating systems can successfully interoperate in an ad-hoc network: they can communicate directly with each other, or if certain nodes within the ad-hoc network provide gatewaying services, nodes in the ad-hoc network may also access the services of external networks, such as HTTP access to the Internet.

1.1 Dissertation Roadmap

In this section the layout of the remainder of this dissertation is outlined.

Chapter 2 presents an introduction to the field of ad-hoc networks in general. The IEEE 802.11 protocol is described, with particular emphasis on its characteristics relevant to ad-hoc networks. The salient characteristics of ad-hoc networks, and some potential applications of ad-hoc networks, are described.

Chapter 3 introduces ad-hoc network routing protocols. The differences between conventional routing protocols and ad-hoc routing protocols are described. A classification of ad-hoc routing protocols is presented. Finally, a number of both proactive and reactive protocols are described, including AODV.

Chapter 4 presents the specific required system services that operating systems should provide for ad-hoc routing protocols. It describes a number of possible approaches taken in the Linux operating system for meeting these requirements. Finally, a survey of the available implementations is presented, describing the design decisions taken, the advantages and disadvantages of each approach.

Chapter 5 introduces the Windows CE operating system, and the Windows CE networking protocol stack. The packet handling mechanisms of the operating system are described, and each evaluated in terms of the required OS support mechanisms an operating system should provide to an on-demand ad-hoc routing protocol.

Chapter 6 summarises the possible approaches for implementing an on-demand ad-hoc routing protocol in Windows CE, giving the advantages and disadvantages of each approach. The design approach and decisions for our implementation are described and justified. Finally, a module level design is presented.

Chapter 7 evaluates the implementation for interoperability, and evaluates the success of our chosen approach.

Chapter 8 concludes this dissertation and presents future work.

2 Properties and Applications of Ad-hoc Networks

2.1 Introduction

This chapter introduces the concept of a Mobile Ad-hoc Network (MANET) **[18]** as a collection of mobile computing devices equipped with wireless network interfaces which can connect together dynamically to create a multi-hop wireless network, without the requirement for any pre-existing infrastructure.

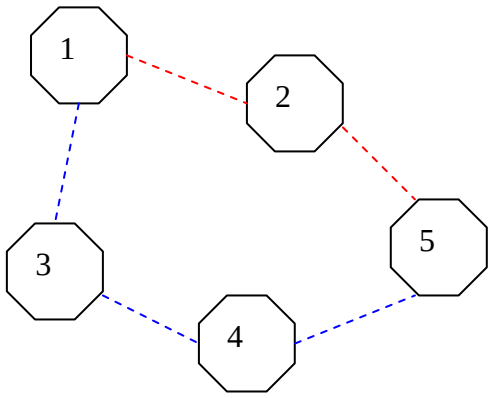

Figure 1: An example ad-hoc network. Two different routes exist between nodes 1 and 5. Nodes act as both a host and a router, offering their services to forward packets.

The usage of the term ‘ad-hoc’ in this manner specifically implies a multi-hop network in which wireless nodes in the network may not be in direct communication range with each other. They may communicate with each other by their network traffic being routed through intermediate members of the network. As such, members of an ad-hoc network must offer their services to other nodes for the purposes of forwarding packets. This is in contrast with the usage of the term ‘ad-hoc’ as used by the IEEE 802.11 standard **[40]**, in which an ad-hoc network simply implies the lack of any pre-existing network infrastructure, but does not imply that nodes offer forwarding services to their peers. In effect, as the term is used in

802.11, all nodes who wish to communicate with each other must be in direct range of each other.

The remainder of this chapter outlines the salient features of the IEEE 802.11 standard, the specific properties unique to wireless ad-hoc networks, as well as some potential uses of ad-hoc networks.

2.2 IEEE 802.11 Wireless Ethernet Standard

A number of wireless Ethernet standards have been developed, the two most common being IEEE 802.11, and the ETSI HiperLAN **[41]**, both of whom have standards describing physical layer operation at speeds of up to 11Mbps in the 2.4Ghz band, and 54Mbps in the 5Ghz band respectively. Both standards specify similar physical layers, but differ significantly in their Media Access Control (MAC) layers **[49]**. Since the technology used in the implementation and evaluation of this dissertation is the 802.11 standard, and since 802.11 has come into much more common general usage, the 802.11 standard is further described here.

There are three specifications for the 802.11 physical layer to operate in unlicensed radio bands. The original 802.11 standard operated at speeds of up to 2 Mbps. Subsequently, the 802.11b standard was developed allowing operation of speeds up to 11Mbps in the ISM 2.4Ghz frequency spectrum, and the 802.11a standard allowing operation of speeds up to 54Mbps in the 5Ghz band. Another standard, 802.11g **[50]**, specifies operation with speeds of up to 54Mbps in the 2.4Ghz band. Different modes of operation are allowed providing fallback operation at lower speeds when the channel is not clear.

There are two different modes of operation in the 802.11 MAC layer **[40]**. The Point Coordination Function (PCF) is used in infrastructure networks, where an Access Point (AP) is used to co-ordinate access to the radio spectrum. Of more interest to ad-hoc networks is the Distributed Coordination Function (DCF) which is used when

there is no AP available, and individual 802.11 nodes must contend with each other for access to the media in a distributed fashion. A Carrier Sense Multiple Access with Collision Avoidance (CSMA/CD) algorithm is used. Collision Detection cannot be used, as a transmitter cannot successfully sense the media for other transmissions while it is itself transmitting, so instead nodes use an exponential back-off scheme with positive acknowledgements to contend for the media.

A well known problematic side-effect of the 802.11 MAC scheme is the hidden terminal problem **[42]**. The hidden node is one that is close enough to the receiver of a transmission such that it can interfere with a transmission being received, but far enough from the sender of that transmission such that the sender does not know the channel is busy at the receiver's location. This causes a collision at the receiver of both transmissions, and a waste of network bandwidth.

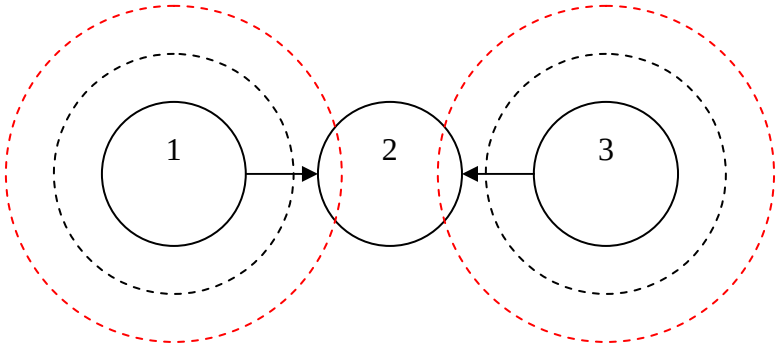

Figure 2: The hidden terminal problem. Nodes 1 and 3 cannot sense each other transmissions (they are out of range of each other), and so their transmissions collide at node 2.

A hand-shaking protocol is often used to deal with this problem **[43]**. A node wishing to make a transmission requests to do so using an *RTS* (Request-To-Send) message. The receiving node then sends out a *CTS* (Clear-To-Send) message if it detects the medium is idle. A *virtual carrier sense mechanism* is employed by nodes via their *Network Allocation Vector (NAV)*. Any node hearing either the RTS or the CTS message will update its NAV for a time period given in the RTS/CTS message,

and refrain from transmitting during this period. The channel is then effectively reserved for the sender.

Another problem associated with the 802.11 MAC is the *exposed node problem* **[44]**. Exposed nodes are those close enough to a transmitter to hear its transmission, and hence refrain from using the media, but far enough from the destination such that its own transmission would not interfere with the reception of the original message at that destination, due to the limited range of wireless transmissions. This leads to under utilisation of the medium.

Interference can cause packets to be incorrectly received at their destination. The 802.11 standard requires that nodes send explicit acknowledgements for unicast packets they receive **[40]**. If the sender does not receive the acknowledgement in a specified time frame, a number of automatic link-level retransmissions are performed for unicast packets. Broadcast packets, on the other hand, use neither positive acknowledgements nor virtual carrier sense mechanisms, and so loss rates of broadcast packets can be significantly higher than unicast packets.

2.3 Characteristics of Ad-hoc Networks

RFC 2501 **[18]** identifies some of the salient characteristics of mobile ad-hoc networks.

2.3.1 Dynamic Topologies

Wireless nodes in an ad-hoc network are free to move about at will. As such, the topology of the network, which is typically multi-hop, is highly dynamic, changing randomly at unpredictable intervals in unpredictable ways. Because of wireless radio propagation effects, such as interference, links may be either bidirectional or unidirectional.

2.3.2 Bandwidth-constrained, variable capacity links

The bandwidth capacity of wireless networks will remain significantly below that of their wired counterparts. The realisable throughput of wireless links above the data-link layer, due to effects such as noise, fading, interference, and the inability to use collision detection for media access control, is often significantly less than the radio's maximum throughput at the physical layer. The effects of this, and given that users of ad-hoc networks will demand similar high-bandwidth services to those on wired networks, means that congestion on wireless networks will be much more common than in wired networks.

2.3.3 Energy-constrained operation

Wireless networks will typically operate on laptop computers, hand-held computers, and other battery-powered devices. As such, ad-hoc routing protocols must be designed with the conservation of the device's energy in mind. There is a conflict between the requirements that nodes in an ad-hoc network must be willing to offer their services for forwarding packets for other nodes, versus the desirability from an energy conservation perspective that nodes sleep when they are not actively being used.

2.3.4 Limited physical security

There is an increased possibility of eaves-dropping, spoofing, and denial-of-service attacks on wireless networks, due in part to their relative lack of physical security in relation to their wired counterparts. Security enhanced versions of ad-hoc routing protocols could be used to ensure the operation of the routing protocol remains unaffected by attempts to forge or alter routing protocol control messages.

Care must be taken when transferring sensitive data across an ad-hoc network. This could be achieved by conventional encryption. However, Public Key Infrastructure (PKI), or more basic key exchange techniques are difficult in an ad hoc network due to the lack of authorities of trust and appropriate network infrastructure **[18]**.

2.3.5 Zero Configuration

Another desirable property of ad-hoc networks is that they should require little or no administrative overhead for their operation. It is desirable that when a group of wireless nodes come together, they can negotiate all the relevant networking parameters automatically without manual intervention. In IP-enabled ad-hoc networks, the most important parameter is a node's Internet Protocol (IP) address. This issue of assigning unique IP addresses to nodes in an ad-hoc network is another area of substantial research. Traditional wired networks typically use a centralised solution to the problem in the form of the Dynamic Host Configuration Protocol (DHCP) **[45]**. Given the lack of a central administrative body in an ad-hoc network, a distributed approach is required. It is likely the solution will involve nodes selecting their IP address at random, and using some means, such as examining Address Resolution Protocol (ARP) **[46]** traffic from other nodes, to prevent or resolve issues where collisions have occurred.

2.4 Potential Applications of Ad-hoc Networks

The emerging field of mobile computing is growing rapidly, and requires highly-adaptive mobile networking technology to manage multi-hop, ad-hoc clusters which can operate either autonomously, or integrated with fixed networks **[18]**.

Some applications of ad-hoc networks may support the formation of autonomous, spontaneous networks. For example, participants at a meeting, or any group of people may want the ability to quickly and easily share documents, files, and otherwise communicate in an area where there is no pre-existing networking infrastructure in place. In the event of a disaster where the existing infrastructure has been damaged or destroyed, the emergency services may wish to establish an on-site communications network, where nodes move around (e.g. nodes could be placed on emergency service vehicles, or on individuals themselves), and therefore the network topology continually changes.

Other applications of ad-hoc networks include where the ad-hoc network is itself connected to the Internet or other fixed networks at one or several points. In this situation nodes in the ad-hoc network communicate with nodes on the fixed network through gatewaying nodes connected to both the ad-hoc and fixed networks. Currently, the MANET working group **[1]** views such networks to most likely be *stub* networks, connected at the fringe of the Internet, but not allowing external traffic to pass through them. This view may be updated as ad-hoc networking technology matures. Examples of such networks include the area of Mesh networking. Mesh networking is a term **[23]** for multi-hop wireless networks that solve the 'last-mile' problem, providing broadband connections to end users in a cost-effective manner. A Neighbourhood Access Point (NAP) is installed with a high-bandwidth connection to the Internet. Users use their own wireless cards to communicate with the NAP, which is used as their gateway to the Internet. By using ad-hoc networking technology, each user effectively extends the range of the network by offering packet forwarding services to their neighbours.

3 Ad-hoc Routing Protocols

3.1 Introduction

The properties of ad-hoc networks as discussed in the previous chapter present some unique challenges for routing protocols. Conventional routing protocols for traditional multi-hop wired networks were designed with specific assumptions in mind that differ from the properties of ad-hoc networks:

- The *topology* of the network is relatively *static*, only changing very slowly over time.

- Individual network links are relatively *reliable*, and *bi-directional*.

- Routes should be maintained to *all* reachable destinations.

In contrast to the properties of traditional wired networks, and the assumptions used in the designing of routing protocols, ad-hoc networks exhibit the following properties:

- The topology of the network is highly *dynamic*, with mobile nodes constantly moving in and out of range with their neighbours. As such links in the network are constantly changing, breaking and being remade.

- Individual network links suffer from radio transmission propagation effects, such as interference from other sources and multi-path fading. Such effects can be different for two communication nodes such that one can communicate with the other, but not vice-versa. As such, communications links may not be symmetric.

- Given the potentially large scale and dynamic nature of ad-hoc networks, it may not be feasible to maintain permanent routing information about every node in the network (particularly about those with which a node does not communicate), as the overhead involved in maintaining the routes will be too great.

The design of ad-hoc routing protocols should attempt to find solutions which overcome these problems. The rest of this chapter describes existing routing protocols for wired networks, it presents a classification for ad-hoc routing protocols, and finally presents a brief description of a few selected ad-hoc routing protocols.

3.2 Conventional Routing Protocols

Traditional routing protocols, designed with the characteristics of relatively static wired networks in mind, involve the periodic exchange of routing information between distributed routers, to allow each router to set up the next hop in a multi-hop network for any particular destination. Link-state routing (e.g. Open Shortest Path First **[24]**) and distance vector routing (e.g. Routing Information Protocol **[25]**) are widely used in conventional networks. A brief description of these algorithms is presented here.

3.2.1 Routing Information Protocol (RIP)

RIP was written by C. Hedrick from Rutgers University in June 1988. It is based on the Bellman-Ford distance vector algorithm. An update process on each router is responsible for informing other routers of the current router's view of the network. Each router periodically sends an update message describing its routing table to all other routers it is connected to. When a router X determines that a router Y has a shorter route to a router Z, then it updates its own routing table accordingly. Shorter paths are as such quickly propagated throughout the network. Updates can be as frequent as thirty seconds to allow routers to react quickly to topological changes.

These updates are not fast enough for the requirements of an ad-hoc network however.

3.2.2 Open Shortest Path First Algorithm

The OSPF algorithm was developed in the late 1980s by the IETF, because the existing routing protocols were increasingly incapable of serving large, heterogeneous networks. The algorithm is a specification of a hierarchical algorithm based on Dijkstra's Shortest Path First (SPF) algorithm. Unlike the distance vector based RIP algorithm, it is a link-state routing protocol that calls for the sending of link-state advertisements (LSAs) to all routers within the same hierarchical area. Information on attached interfaces, metrics and other variables is included in the LSAs. As routers accumulate link-state information, they use the SPF algorithm to calculate the shortest path to each node. As with RIP, routing updates are not frequent enough to allow an ad-hoc network to react quickly to topological changes.

Because both RIP and OSPF aim to maintain a global view of the network, as mobility within a network increases, so too does the amount of control traffic that is required to maintain a consistent view of the network. Node mobility will always reach a level for which the control traffic will overwhelm the capacity of the network in such protocols. Both these algorithms have difficulty converging at high mobility rates, and can suffer from routing loops **[2]**.

3.3 Classification of Ad-hoc Routing Protocols

Ad-hoc routing protocols can broadly be classified into *proactive*, *reactive* and *hybrid* protocols **[39]**. The approaches involve a trade-off between the amount of overhead required to maintain routes between node pairs (possibly pairs that will never communicate), and the latency involved in discovering new routes as needed.

3.3.1 Proactive Protocols

Proactive protocols, also known as table-driven protocols, involve attempting to maintain routes between nodes in the network at all times, including when the routes are not currently being used. Updates to the individual links within the networks are propagated to all nodes, or a relevant subset of nodes, in the network such that all nodes in the network eventually share a consistent view of the state of the network.

The advantage of this approach is that there is little or no latency involved when a node wishes to begin communicating with an arbitrary node that it has not yet been in communication with. The disadvantage is that the control message overhead of maintaining all routes within the network can rapidly overwhelm the capacity of the network in very large networks, or situations of high mobility.

Examples of pro-active protocols include the Destination Sequenced Distance Vector (DSDV) **[2]**, and Optimised Link State Routing (OLSR) **[6]**.

3.3.2 Reactive Protocols

Reactive protocols, also known as on-demand protocols, involve searching for routes to other nodes only as they are needed. A route discovery process is invoked when a node wishes to communicate with another node for which it has no route table entry. When a route is discovered, it is maintained only for as long as it is needed by a route maintenance process. Inactive routes are purged at regular intervals.

Reactive protocols have the advantage of being more scalable than table-driven protocols **[39]**. They require less control traffic to maintain routes that are not in use than in table-driven methods. The disadvantage of these methods is that an additional latency is incurred in order to discover a route to a node for which there is no entry in the route table.

Dynamic Source Routing (DSR) **[20], [21]**, and the Ad-hoc On-demand Distance Vector Routing (AODV) **[8]** protocol are examples of on-demand protocols.

3.3.3 Hybrid Protocols

There exists another class of ad-hoc routing protocols, such as the Zone Routing Protocol (ZRP) **[4]**, which employs a combination of proactive and reactive methods. The Zone Routing Protocols maintains groups of nodes in which routing between members within a zone is via proactive methods, and routing between different groups of nodes is via reactive methods.

Additionally, routing protocols can employ temporal information, for example location co-ordinates from the Global Positioning System (GPS), to aid in the rapid establishment of routes to a new destination.

3.4 Description of selected Ad-hoc Routing Protocols

This section contains a brief description of some selected proactive and reactive ad-hoc routing protocols. For more detailed descriptions the reader is referred to the appropriate specifications, Internet drafts and RFCs.

3.4.1 Destination Sequenced Distance Vector (DSDV) Routing

DSDV **[2]** is one of the earliest attempts to deal with the problems of traditional routing protocols used in wireless networks. The authors note that most protocols exhibit their worst performance within the context of a highly dynamic interconnection topology, placing too heavy a computational burden on each mobile computer, and having poor convergence characteristics.

DSDV is based on the classical distributed Bellman-Ford algorithm used in wired networks. It is a proactive ad-hoc routing protocol which uses destination assigned sequence numbers to avoid the traditional counting to infinity problem associated with distance vector algorithms. Each node maintains a full routing table for all

nodes in the network, containing the next hop address, remaining hop count to the destination, and the sequence number of the last route advertisement for that route. Routing table updates are periodically broadcast by nodes using one of two different types of update packet.

Full dump packets contain the full routing table of a node. If the routing table is large, this packet type may require several Network Protocol Data Units to transfer the full table. This transmission type occurs infrequently to conserve network resources if the node experiences limited topological changes in relation to its neighbours. Incremental packets contain only the information that has changed since the last full dump was sent out by the node. As such they consume a much smaller portion of network resources than full dump packets.

Routing information received from a neighbouring node will be merged with its own if the information contains a newer route (given by a higher destination assigned sequence number), or a route with a lower hop count (that is also as least as up-to-date as the current route). Newly discovered routes will immediately be advertised by a node, and updated routes will cause an advertisement to be scheduled for transmission within a certain settling time (the time between the first route with a new sequence number and the shortest route). DSDV uses bidirectional links only.

Performance evaluations **[16], [19], [35]** of DSDV indicate that it experiences low throughput and problems converging at higher node mobility rates. Subsequently developed ad-hoc routing protocols have been designed to improve on these characteristics of DSDV.

3.4.2 Dynamic Source Routing (DSR)

DSR **[20]**, **[21]** is a reactive ad-hoc routing protocol which exhibits good throughput and convergence statistics in performance studies **[16], [19], [35]**. It utilises source routing, in which a sender constructs a source route in the packets header containing the address of each hop through which the packet should be forwarded. Mobile hosts

participating in the network maintain a route cache in which they cache source routes that they have learned. Caching of negative information in the form of unreliable links is also supported via a temporary 'black-listing' mechanism. Entries in the cache have a certain expiration period, after which the entry is deleted.

The *route discovery* protocol is the process responsible for allowing any node in the ad-hoc network to dynamically discover a route to any other host in the network. The node initiating the discovery broadcasts a *route request* packet which is received by those nodes in range of it. Each route request contains the address of the source node, the address of the target node, and a *route record*, initially empty, in which is accumulated a record of the sequence of hops taken by the route request packet as it is forwarded throughout the ad-hoc network. A route request id is used to prevent the forwarding of duplicate route requests. If a node receiving a route request packet is the target of the route request, then the route record is copied into a *route reply* packet, and this packet is returned to the initiator. Otherwise, the node either discards the route request if it has seen it before or its address is already present in the route record, or it appends its own address to the route record and rebroadcasts the route request to its neighbours. In order for the target node to send the route reply back to the initiator, it must know a route back to the initiator. If bidirectional connectivity is assumed, then the route in the route request may be reversed and this used as the return path. If the network may contain unidirectional links, then the target node will piggyback the route reply upon a new route request, with a new destination being the node that initiated the first route request.

The *route maintenance* procedure is responsible for monitoring the operation of the route and informing the sender of any routing errors. The basis of route maintenance is the sending of *route error* messages. Most wireless transmission protocols, such as 802.11, use data link level acknowledgements and retries for the early detection of errors in the transmission of packets. As such, the data link layer can report errors to the routing protocol when a packet cannot be transmitted, for example, after a certain number of retries have failed. Such an error is reported using a route error message. The packet contains the addresses of both the node that detected the error and the

node to which it was attempting to transmit a packet. When a node receives a route error message, it searches its route cache for any routes containing this hop, and any routes so found must be truncated before this hop. Where data link level acknowledgements are not available, other methods of detecting errors may be used. Passive acknowledgements involve a node listening for the transmission by a neighbour, to which the node has just forwarded a packet, of that packet to the neighbour's next hop. As such the receipt of the packet by the neighbour is implicitly acknowledged. Another method of acknowledgement involves including a bit in the packet header to allow the transmitting node to request an explicit acknowledgement by the receiving node. In order to return the route error message to the sender, the node detecting the error may use a route to the sender it already knows about in its cache, it may reverse the route in the packet that could not be transmitted (assuming the links are bidirectional), or it may piggyback the route error message on a route request message for the original sender.

A *route reply storm* occurs when many neighbours of a node initiating a route request contain cached entries to the destination of the route request. Their simultaneous replies can cause heavy media contention. To alleviate this problem the DSR protocol requires nodes to pause before replying to a route request. The length of the pause depends on the hop length of the route being returned. Nodes promiscuously listen to the media while pausing, and cancel their pending reply if they hear another reply.

3.4.3 Ad-hoc On-demand Distance Vector (AODV) Routing

AODV is an ad-hoc routing protocol for discovering routes between hosts, potentially over multiple hops, as they are needed, and only for the duration that they are needed. It is designed to take into account the problems of limited transmission range and node mobility, and hence a continually changing network topology, found in mobile ad-hoc networks.

AODV enables nodes to communicate with other nodes they are not in range of by routing packets through neighbouring nodes. The AODV protocol discovers these routes that packets may take between a source and destination. The protocol does this while ensuring that routing loops do not occur, and it also attempts to find the shortest route possible. It can handle changes in routes and discovers new routes when an old route no longer works.

The AODV protocol consists of a number of messages which it uses for route discovery, route maintenance and repair, and neighbour detection.

Route Request (RREQ) Messages:
When a node needs to send a message to another node that is not its direct neighbour, it broadcasts a Route Request message to initiate the discovery of a route. The RREQ message contains several important bits of information: the source IP address, the destination IP address, the lifespan of the message and a sequence number which uniquely identifies messages from this source.

When the neighbours of the node who initiated the Route Request receive the message, they can do one of two things: if they know of a route to the destination or they themselves are the destination, they can unicast a Route Reply (RREP) message back to the source node; otherwise, they will rebroadcast the Route Request to their neighbours.

The lifespan of the Route Request is decremented by one at each hop, and the message is simply discarded when the lifespan reaches zero. In this manner, the protocol can implement an expanding ring search, in which the lifetime of an initial Route Request is set to a low number to limit the propagation of RREQ messages. If no reply is received within a specified amount of time, the source node issues a new Route Request with a new sequence number and higher lifetime. A number of different attempts can be made using successively larger lifetimes, or after a fixed number of retries, the lifetime is set to be greater than the network diameter, so the Route Request will be broadcast to all nodes connected in the network.

All the nodes keep a list of the Route Requests, including sequence numbers, for a particular source that they have rebroadcast in a fixed interval, to ensure they do not rebroadcast the Route Request more than once.

Route Reply (RREP) Messages:

When a node contains an up-to-date route to a destination that is the target of a Route Request it receives, or is the destination itself, it unicasts a Route Reply (RREP) message back to the node it received the Route Request from. Each node along the path that the Route Request was propagated updates its routing table to mark the node from which it received the Route Reply as the next hop for the new route. As such, the Route Reply is propagated along the reverse path all the way to the source of the original Route Request and the routing table of each node along the way is updated to reflect the next hop along the route.

In case the node replying to the Route Request was not the destination but instead knew a valid route to the destination, then this node also sends a *gratuitous* Route Reply to the destination along the path it knows to that destination, such that the destination knows how to reply to the source when it receives data from it, without having to explicitly send out another Route Request to search for the source.

Figure 3 illustrates the route discovery process, involving the propagation of Route Request and Route Reply control packets.

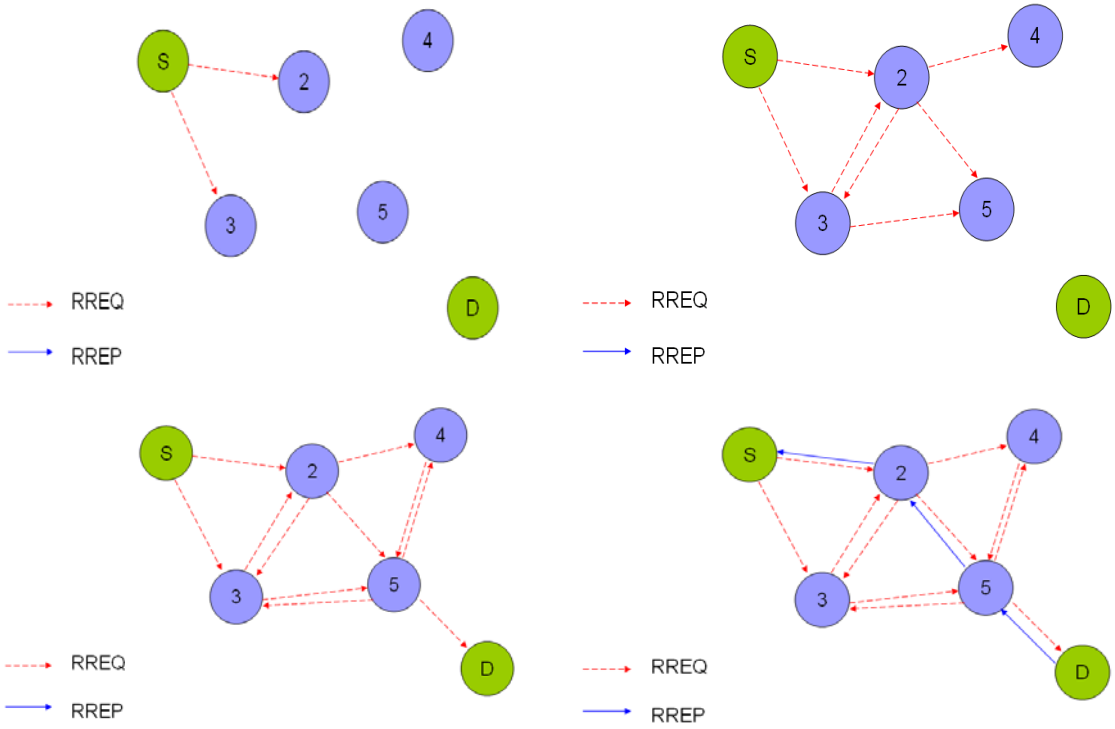

Figure 3: The route discovery process. Top right: The source node (s) broadcasts a Route Request message for the destination node (d). Top left: Nodes 2 and 3 rebroadcast the message to their neighbours. Bottom Left: Nodes 4 and 5 rebroadcast the Route Request message to their neighbours. Bottom Right: The destination receives the Route Request message, and broadcasts a unicast Route Reply message which is propagated along the same path that the Route Request message was first received.

Sequence Numbers:

The protocol uses sequence number to ensure loop freedom in routes and to act as a kind of timestamp such that nodes may detect when they receive more up to date routing information. Each node maintains its own sequence number, which it increases any time it sends out any kind of message. Each node maintains a record of the sequence number of all the nodes it has routing information for. A higher sequence number indicates a fresher route. Thus it is possible for other nodes to determine which Route Reply message has more up-to-date information. Nodes may for example update their routes to a destination if they observe a Route Reply that contains a higher sequence number for the destination than the one stored in their routing tables.

Route Maintenance Process:

In addition to the route discovery process just outlined, AODV is responsible for maintaining active routes in the network. Routes are only kept for as long as they are in use. After a timeout period, stale routes will be removed from a node's routing table. The route maintenance process is also concerned with detecting route breakages. Each node in the network monitors its connectivity to neighbours that are being used as next hops for active routes. It can use link layer notification methods to detect route breakages. For example, in the 802.11 standard, the absence of a link layer ACK or failure to get a CTS after sending an RTS, even after the maximum number of retransmission attempts, indicates loss of the link to this active next hop. In the absence of link layer information, a node uses passive acknowledgement to detect broken links. Receipt of packets from the next hop, including HELLO messages are usually used for this process, described next.

HELLO Messages:

In order for nodes to remain aware of who their neighbours are, they may periodically broadcast HELLO messages. HELLO messages are simply Route Reply messages sent with a hop count of zero, so it is not propagated. A node keeps track of its neighbours by listening for the periodic messages. After an allowable HELLO message loss, a node will detect a broken link by the absence of a HELLO message, indicating that the nodes can no longer directly communicate. If this link was in use by any active routes, this broken-link detection mechanism will result in the sending of a Route Error (RERR) message, as described next.

Route Error (RERR) Messages:

Route Error (RERR) messages allow AODV to adjust routes when nodes move around or otherwise lose the ability to transmit to one or more of their neighbours. When a node receives a Route Error message, it removes all the routes from its routing tables that contain the invalid next hop. There are three circumstances in which a node will broadcast a Route Error message.

If a node receives data from a neighbouring node for a destination to which it has no route, it will broadcast a Route Error message. In this case the neighbouring node had some stale or otherwise incorrect routing information. In the second scenario the node receives a Route Error message that causes at least one of its routes to become invalidated. If this happens, the node then sends out a Route Error message with all the new Nodes which are now unreachable. In the final scenario, if a node detects (through the absence of HELLO messages, or via other link level notification methods) that it can no longer communicate with one of its neighbours, it will check its routing table for all routes that use this neighbour as the next hop, and mark them as invalid. It then sends out a Route Error message for the neighbour and the invalid routes.

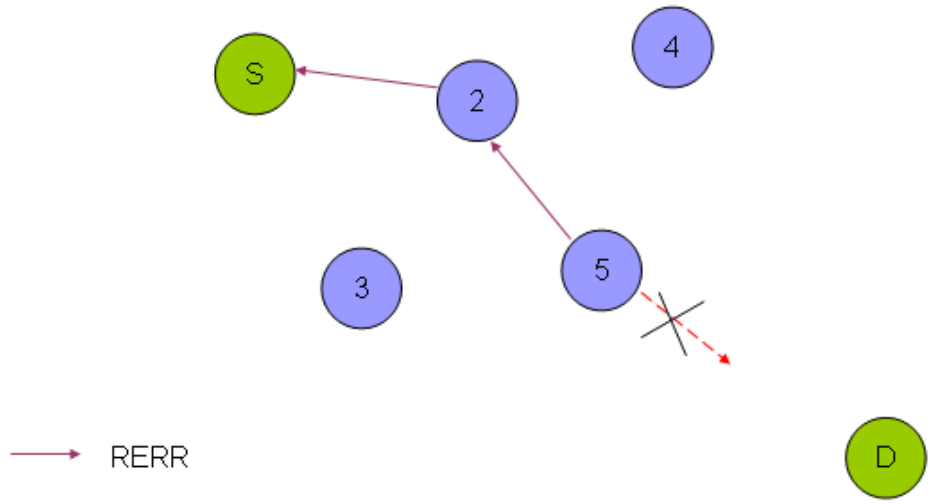

Figure 4: Route maintenance process – broken link. When node D moves out of range of its previous hop, node 5 detects the broken link either via link level mechanisms or the absence of HELLO messages. Node 5 sends a Route Error which is received by node 2. Node 2 in turn will invalidate the route and send a route error to the source node, S. S can then reinitiate the route discovery process if it wishes.

Local Repair Mechanism:

When a link break occurs in an active route, a node upstream of the break may choose to attempt to repair the route locally if it is within a certain number of hops away from the destination. In this case it issues a Route Request for the destination, and defers sending the Route Error message unless the local repair mechanism is

unsuccessful. Incoming packets at the node upstream of the break should be buffered by the node until the local repair is complete.

4 Survey of Existing AODV Implementations

4.1 Introduction

This chapter characterises the various publicly available, open source AODV implementations. There are a limited number of such implementations, mostly for the Linux operating system. At the time of writing, these include implementations by the National Institute of Science and Technology (NIST) **[9]**, the University of California, Santa Barbara (UCSB) **[10]**, and Uppsala University (UU) in Sweden **[11]**. The University of Illinois, Urbana-Champaign (UIUC) has an implementation which is based on their Ad-hoc Support Library (ASL) **[12]**, a user-space library which provides an API to facilitate implementation of routing protocols for wireless ad-hoc networks in Linux. The earliest implementation of AODV is the Mad-hoc implementation **[25]**. Finally, a number of extensions exist for the UU implementation, including support for the Multicast AODV (MAODV) protocol, and IPV6.

The remainder of this chapter is divided into three sections. The first describes the required system support for on-demand ad-hoc routing protocols. The next section describes the possible design approaches for writing ad-hoc routing protocols in the Linux platform. The remainder of the chapter describes the existing AODV implementations and the approaches they use to solve the problem. Chapter 5 describes the design choices available for the Windows CE platform.

4.2 Required OS Support for Ad-hoc Protocols

The protocol stacks of modern operating systems have not been designed with support for ad-hoc routing protocols. They have been designed for networks where routing links are configured and known in advance. Using the terminology of **[28]**,

the routing functionality in modern operating systems is typically divided in two parts: the *packet-forwarding* function, and the *packet-routing* function. In this terminology the packet-forwarding function consists of the routing function within the kernel, located within the IP layer of the TCP/IP stack, in which packets are directed to the appropriate outgoing network interfaces, or local applications, according to the entries in the kernel routing table. When the IP-layer receives a packet, either from a local application or on one of its network interfaces, the kernel routing table is consulted. The packet is either directed to a local application listening on the specified port number, dropped, or sent out to the corresponding next-hop neighbour on the specified network interface according to the destination IP address of the packet.

The packet-routing function typically consists of a user-level program responsible for populating the kernel routing table. Using this separation, packets can efficiently be processed solely in kernel space, minimising expensive context switches to and from user space, while allowing the flexibility to easily change the routing protocol.

Proactive ad-hoc routing protocols can operate within this architecture without difficulty. However, this architecture will not easily accommodate ad-hoc routing protocols. In a normal Sockets application, when the application attempts to open a Socket to a destination which is not contained within the kernel routing table, then the open Socket call immediately returns with an error code. However, in on-demand routing protocols not all routes are known in advance, they must be discovered as they are needed. In such cases a mechanism is required to notify the on-demand routing protocol that a route discovery cycle must take place for the destination, and any packets already being sent to the destination must be queued while the route discovery cycle completes.

Thus ad-hoc routing protocols require the protocol stack (such as TCP/IP) to have some additional capabilities for dealing with cases where a route to a node is not known in advance, such as the ability to buffer packets while a route discovery cycle takes place. In addition, ad-hoc routing protocols require the protocol stack to

provide notifications of particular network events, such as the need to initiate a route discovery cycle. The particular required extended protocol stack capabilities, including notification mechanisms, for the AODV routing protocol are **[28]**:

1. To determine when a route request is needed: Route Requests are needed when the IP layer receives a packet to be transmitted to an unknown destination, i.e., a destination with no matching entry in the route table.

2. The capability to buffer packets waiting for a route discovery cycle (or for some other reason) to complete: When an application attempts to send a packet to a destination for which the routing table has not a valid route, the IP layer should buffer the packet for a period of time while a route discovery cycle takes place. When the next-hop entry for the destination is successfully entered in the kernel routing table, the buffered packets for that destination should be released into the IP layer.

3. To determine when to update the lifetime of a route: On-demand routing protocols typically cache a route that has been discovered for a period of time before deleting it if it is inactive. The IP layer therefore must have the capability to notify the routing protocol when an on-demand route has been used, so that the routing protocol can update its timers for the route.

4. To determine when to send a route error message if a route does not exist for the next-hop IP address of a received packet: Normal operation of the IP layer on receiving a packet destined for a node for which it has no valid routing entries is to send a destination host unreachable ICMP message to the source of the transmission, and silently drop the packet. Instead, the IP layer must give notification to the AODV routing protocol such that it knows it should send a route error message to the original source or the packet.

5. To determine when to send a Route Error message if the node receives any packets during the DELETE_PERIOD: When a node reboots, the AODV

specification requires that it sends Route Error messages to any nodes attempting to communicate with it up until the end of DELETE_PERIOD seconds. This is required in order to avoid routing loops forming shortly after reboot.

These notifications and capabilities are not explicitly present in the protocol stacks of modern operating systems. The existing implementations have taken a number of different approaches to solving this problem. The next section describes a number of possible approaches that implementers have taken in Linux.

4.3 Design Strategies for Linux

The following design strategies for AODV implementations have been adopted:

- Snooping of ARP and data packets.
- Using the Netfilter packet-filter and packet-mangling architecture.
- Modifying the kernel to produce a new API for ad-hoc routing implementations.

4.3.1 Snooping

By snooping the Address Resolution Protocol (ARP) packets and data packets, AODV can be implemented without any kernel modifications. As such, the routing protocol can be implemented easily in either kernel space or user space. The routing protocol can determine when a route discovery cycle is needed by snooping ARP request packets, as an ARP request is sent to resolve the hardware address for an unknown IP address (if there is an appropriate subnet route entry set up for the correct interface). This is requirement 1 of section 4.2.

The routing protocol can observe incoming and outgoing data packets, and as such can determine when a route is being used (see requirement 3 of section 4.2), or when a packet is received for which we have no routing information (see requirements 4 and 5).

The main drawback of this approach is that there is no way of properly meeting requirement 2, that is, packets cannot be properly buffered while route discovery takes place, and will instead be dropped immediately by the IP layer. Most ARP implementations only buffer one packet at a time while an ARP resolution takes place, and any subsequent packets for the same destination will overwrite this. In addition, this packet is only buffered for the duration of the ARP timeout, which is often smaller than the time taken for a route request. While IP is a 'best-effort' protocol, it is still a good idea to avoid systematic problems that lead to definite packet losses **[27]**.

Another drawback of this approach is that the kernel generates an ARP request only if the destination belongs to the subnet of one of the network interfaces, or a host specific entry in the kernel routing table. Otherwise the packet will simply be discarded in the IP layer, without the AODV routing protocol ever knowing it existed. As such this violates the principle that AODV can operate with networks of nodes of unrelated IP addresses **[28]**.

Also, an ARP cache has a time-out value associated with each automatic entry; hence ARP requests will also be generated periodically for routes for which the next-hop IP address is already known. Spurious route requests will result. Similarly if the ARP cache contains an entry for a destination, but the route table does not (e.g. a manually configured ARP entry), then no ARP request will ever be generated for this destination, and route discovery will fail.

4.3.2 Netfilter

Netfilter **[30]** is a packet-mangling architecture, not included in the Berkeley socket interface, for the Linux 2.4 kernel. It consists of a number of hooks in the IP layer that are well-defined points in a packet's traversal of the protocol stack. The IP V4 stack defines the following five hooks: NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN, NF_IP_FORWARD, NF_IP_POST_ROUTING, and NF_IP_LOCAL_OUT. The routing hooks are called in the following fashion:

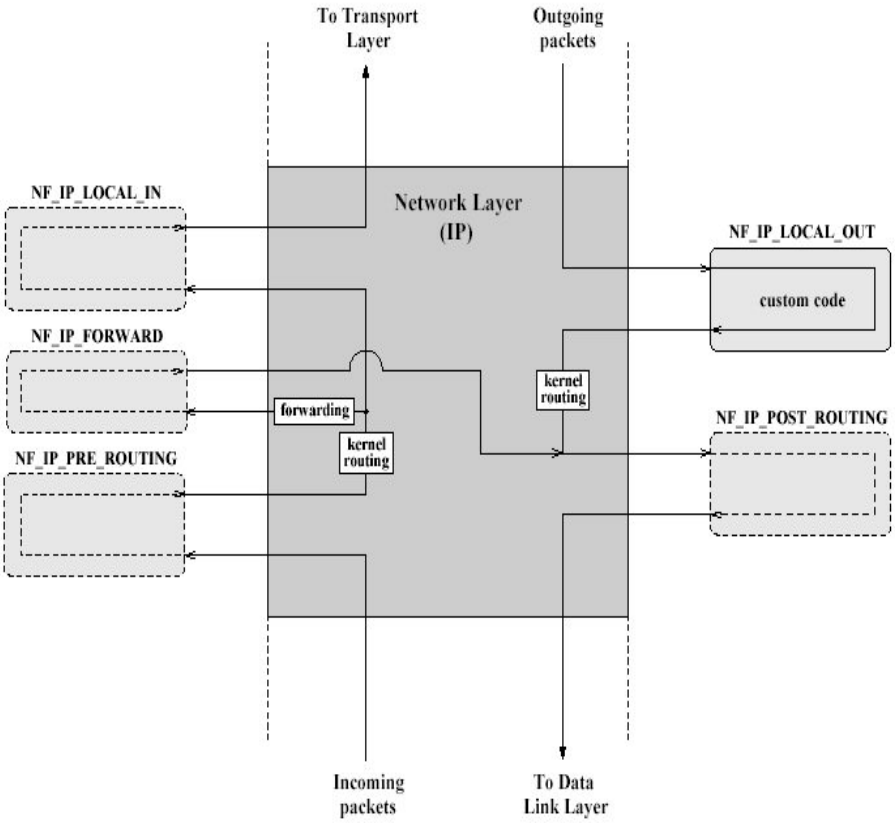

Figure 5: The Linux Netfilter Packet Mangling Architecture [47].

The NF_IP_LOCAL_IN and NF_IP_LOCAL_OUT hooks are for packets incoming to and outgoing from local processes on the current host. Either before a packet traverses NF_IP_LOCAL_IN, or after it traverses NF_IP_LOCAL_OUT, it is subjected to kernel routing. Here a routing decision on what to do with the packet is made. If it is an incoming packet, it may be sent to the NF_IP_FORWARD hook before forwarding, sent up to the NF_IP_LOCAL_IN hook for delivery to a local process, or dropped. If it is an outgoing packet, it is dropped or sent on the

NF_IP_POST_ROUTING hook before being released to the appropriate network interface driver for transmission across the network. Incoming packets also traverse the NF_IP_PRE_ROUTING hook as they enter the IP layer, before being subjected to kernel routing.

Kernel modules can register functions with these hooks. When a packet enters this part of the protocol stack, Netfilter checks to see if anyone has registered with that hook; if so, each function registered is called in turn with the packet as a parameter. These functions can view, or alter packets as they traverse the hooks. They then have the choice to either discard the packet (by returning NF_DROP), allow the packet to pass to the next registered hook (by returning NF_ACCEPT), grab the packet for their own exclusive use (by returning NF_STOLEN), or request that these packets be queued for later reinjection into the IP layer (by returning NF_QUEUE).

Finally packets that are queued (by returning NF_QUEUE) are buffered by the ip_queue driver, typically (though not necessarily) for user space. These packets are handled asynchronously and thus they can be returned to the IP layer at any later time, or discarded.

The Netfilter architecture can be used for firewall filtering (the Linux iptables tool uses Netfilter in version 1.4 of the kernel), all kinds of Network Address Translation (NAT) services, or for other advanced packet processing requirements.

Netfilter provides a very convenient and flexible mechanism for the construction of ad-hoc routing protocols. Outgoing packets can be examined by an AODV routing protocol on the NF_IP_LOCAL_OUT before routing decisions are made in the IP layer. As such the routing protocol can observe packets destined for unknown destinations and initiate a Route Request (requirement 1 of section 4.2). It can return a verdict of NF_QUEUE for these packets. This instructs the ip_queue driver to buffer these packets for later reinjection to the IP layer (requirement 2 of section 4.2). By examining all outgoing packets on this same hook, the routing protocol can determine when a particular route is being used and can update its timer for the route

accordingly (requirement 3 of section 4.2). By registering a function with the NF_IP_PRE_ROUTING, the routing protocol can determine when it receives packets to forward for which no next-hop route exists on this node. In this case, the routing protocol will send out a Route Error message to the source of the packet (requirement 4 of section 4.2). Similarly if the AODV routing protocol receives any packets for forwarding during the DELETE_TIME period at boot-up, it will detect them on the NF_IP_PRE_ROUTING hook (requirement 5 of section 4.2).

In order to use Netfilter, an AODV routing protocol needs to register a kernel module to register call-back functions with the required Netfilter hooks.

The Netfilter method is portable across Linux implementations, it is easy to install, and all the required capabilities and notifications for ad-hoc routing protocols can be easily determined. A weakness of this approach is that one cannot implement a user space only solution if desired.

4.3.3 Producing a system 'on-demand ad-hoc routing protocol API'

Perhaps the most interesting long term solution to allowing implementers to easily produce and test new on-demand ad-hoc protocols would be to provide built in support directly in the next generation of operating systems for ad-hoc protocols. Such an API would require mechanisms as outlined in section 4.2. An API would require protocols to register interest with the kernel in the relevant routing events (such as the requirement for a new Route Request). The kernel would then inform the ad-hoc routing protocol when a route Request is required; it would provide a mechanism to buffer packets for which a route request is being performed and to later reinject them; it would maintain timers associated with the route, etc. **[28]**, **[29]**.

A drawback of this approach is that it will require major changes to the operating system kernels, and will not be very portable for existing operating system kernels without requiring users to install a new kernel.

Kawadia, et al **[28]**, have performed interesting work towards such an approach using Linux kernel modules and user space libraries, with their Ad-hoc Support Library (ASL). This is further described in the following section.

4.4 AODV Implementations

This section describes the available implementations of AODV for the Linux platform. It describes their available functionality and draft compliance. It describes their design philosophy and any advantages/shortcomings of their specific design choices.

4.4.1 Mad-Hoc Implementation

Mad-Hoc was the first available implementation of AODV. It uses the method of snooping ARP and data packets as described in section 4.3.1 above, using the libpcap Linux packet capturing facility. It is a user space only solution. It does not comply with an up-to-date version of the AODV specification, and is no longer supported. As such, it does not interoperate properly with the later implementations, and is not recommended for use.

Some of the later implementations of AODV, such as the NIST implementation described below, were based on the Mad-Hoc code as a starting point.

The approach it takes of relying on ARP means that it suffers from many disadvantages, including the loss of packets as route discovery takes place.

4.4.2 NIST Implementation

The NIST Implementation of AODV is currently at version 2.1 at time of writing. It complies with version 11 of the AODV draft. The latest version has support for multicast AODV, as well as multi-hop Internet gatewaying. It has support for x86,

ARM, and MIPS architectures. It has support for monitoring the wireless signal strength between neighbours. The implementation has been successfully tested for interoperability with other publicly available implementations, including the UCSB and UU implementations **[6]**.

The protocol is implemented completely as a Linux kernel module. Being implemented as a kernel module means that it has more efficient access to system resources than a version of the protocol implemented as a user space daemon. If uses Netfilter from the 2.4 kernel to capture packets going in and out of the node instead of using the libcap library. It uses a Proc file to update the user about current routes and statistics for that node.

The implementation is written in C. It is multi-threaded and uses queues to limit the amount of time spent directly handling an interrupt from the Netfilter library to handle incoming and outgoing packets. It is implemented entirely as a Linux loadable kernel module. No direct kernel modifications are required.

Being a kernel only solution, packets that are queued (as a result of returning NF_QUEUE from the function registered with the IP_LOCAL_OUT hook), do not need to cross the barrier into user space, which requires an expensive context switch. As such, it should be able to perform routing faster than user space solutions. A drawback is that any bugs in the implementation may cause the entire system to become unstable or crash, as the code runs with all the privileges of any kernel level code.

4.4.3 Uppsala University Implementation

The UU implementation of AODV is at release 0.7.2 at time of writing. It complies with version 13 of the AODV draft. Multicast support is available via a patch implemented by a group of researchers from the University of Maryland. Multi-hop Internet gatewaying support is included, but is at an early stage of development in the current release. The implementation has support for x86 and ARM micro-

processors. The implementation has been successfully interoperability tested with the NIST and UCSB versions.

While the authors maintain that draft compatibility is a goal of their implementation, they also state that the main purpose of the implementation is as a research test-bed, and as such it contains some functionality not seen in the current AODV draft, such as optional unidirectional link detection and avoidance **[7]**. Their motivations for these improvements are concerned with properties of broadcast messages in the 802.11 standard which cause difficulties in using HELLO message for link detection. Because 802.11 broadcast messages are transmitted at lower rates than unicast messages, they also have a longer range. Because of this, nodes may believe a link is still active (as HELLO messages are still received), even though unicast messages are failing. This can have a serious detrimental effect on packet error rates. The authors propose some remedies for this problem.

The protocol is implemented as a user space daemon, and two loadable Linux kernel modules (kaodv and ip_queue_aodv). It uses the Netfilter library to intercept incoming and outgoing packets, but this is performed in user space. The kaodv module uses Netfilter to buffer *all* packets for user space by returning NF_QUEUE, and ip_queue_aodv queues them for userspace. The UU implementation then matches the destination address of all packets against the user space route cache. It buffers in user space those packets that require a route request for their successful routing, and it immediately returns those packets for which it already has a route. Copying all packets from kernel space to user space, including context switches, and back again is a wasteful and expensive operation. The NIST implementation described in section 4.4.2 does not suffer this drawback as it is entirely a kernel space solution. The UIUC implementation described in section 4.4.5 only requires that packets for which a Route Request is needed are copied to user space. However, the authors of the UCSB implementation state that this greatly simplifies coding, and they assert that they prioritised stability over performance for their implementation.

The implementation is written in C. It is written as a user space daemon which simplifies some of the code operation.

A further extension patch is available from the Simon Fraser University to modify the implementation for operation using IPV6.

4.4.4 University California, Santa Barbara Implementation

Version 0.1b of the implementation is described here. This implementation does not seem quite as well developed as the previous two. It complies with version 10 of the AODV draft. Multicast support is not available. Gatewaying support is not available, and the route configuration requirements are more complex than for the previous two implementations. The implementation has been successfully tested for interoperability with the NIST and UU implementations.

Similar to the UU implementation, the UCSB version is implemented as a user space daemon. It similarly uses the Netfilter library for intercepting incoming and outgoing packets from the chosen interface. In fact, the implementation uses directly the UU packet_input user space packet queuing module and the kaodv/packet_queue_aodv kernel modules. As such, it suffers from exactly the same problems as the UU implementation in that *all* packets must pass the boundary between kernel and user space twice.

It has been tested with Linux kernel 2.4.12, and should work with any kernel with Netfilter installed. The code is written in C.

4.4.5 University of Illinois, Urbana-Champaign Implementation

The UIUC implementation **[12]**, **[28]**, is based on their ad-hoc support library (ASL), which is a Linux specific library designed to provide all the services required by ad-hoc routing protocols. As such, the UIUC AODV implementation is a user space

daemon compiled against the ASL library. The implementation has not been interoperability tested against the other protocol implementations.

The code is written in C++. Much of the complexity of the user space ad-hoc routing module has been removed to the ASL library, a very desirable feature, as this should allow other, different, ad-hoc routing protocols to be developed using the same library.

The authors propose adding a flag to each kernel route entry to denote an on-demand entry. A deferred entry is then one that is awaiting completion of a route discovery cycle. They add an on-demand routing module (ODRM) which implements the on-demand routing functionality. It notifies the user space ad-hoc routing daemon through their API of a route request. It buffers packets and waits for the ad-hoc routing daemon to return with the route discovery status. Once the deferred routing entry is updated to reflect the new next-hop, the packet is reinjected into the IP layer for transmission, or dropped if the route discovery cycle was unsuccessful.

They add timestamp fields to each route entry in the kernel route table to indicate the last time that this entry was used to transmit a packet on this route. This field is used to expire stale routes.

They then provide an API in the form of a static library against which the ad-hoc protocol daemon links itself. They specify the following API:

```
int route_add(addr_t dest, addr_t next_hop, char* dev)
int route_del(addr_t dest)

int open_route_request();
int read_route_request(int fd, struct route_info *r_info);

struct route_info {
        addr_t dest,
        addr_t src,
        u_int8_t protocol
        }

int route_discovery_done(addr_t dest, int result);
```

int query_route_idle_time(addr_t dest)

The first two functions are used to update the kernel routing table with new routes.

The next two functions are used to discover when a Route Request is necessary. Route Requests are represented as structs and read from the file descriptor returned from open_route_request. route_discovery_done is used to inform the kernel when the Route Request is complete and the route table has been successfully updated; buffered packets may now be reinjected. query_route_idle_time is used to query the timer associated with routes so the ad-hoc routing daemon knows when to expire a route.

The researchers managed to implement their support library without making any direct kernel modifications, only using loadable kernel modules, and user level programs. Given that there is still interaction between kernel and user level programs, this implementation still suffers performance penalties in comparison to kernel space only solutions such as the NIST implementation.

They have also ported the UCSB implementation to use their ASL. In solving the problem of having to send every packet between kernel and user space, they showed their modified version of the UCSB implementation was faster and more efficient.

5 Ad-hoc Protocols in the Windows CE Operating System

5.1 Introduction

This chapter introduces some of the possible design choices for the implementation of on-demand ad-hoc routing protocols such as AODV for the Microsoft Windows CE platform. As the discussion of Linux implementations of AODV in chapter 4 illustrated, all of the modern implementations utilise the Netfilter architecture for examining and filtering packets asynchronously during their traversal of the IP layer in the protocol stack. Unfortunately there is no direct counterpart to Netfilter available for the Windows platforms.

Section 5.2 of this chapter describes the salient characteristics of the Windows CE platform for those unfamiliar with the operating system. Section 5.3 presents an overview of the Windows CE communications subsystem architecture. Following this, the various features of the Windows protocol stack related to packet manipulation are described, along with their suitability for the purposes of meeting the system requirements for ad-hoc routing protocols described in section 4.2.

5.2 Windows CE Overview

According to Microsoft, “Windows CE is an open, scalable Windows platform for a broad range of communications, entertainment, and mobile-computing devices” **[31]**. It has been designed for non-PC devices, such as PDAs, mobile phones, set-top boxes, embedded devices in cars, industrial automation devices, and the like. Windows CE .NET, the latest version of the Windows CE operating system, is a hard real-time operating system, with a pre-emptive multitasking kernel, designed to have a deterministic response to interrupts.

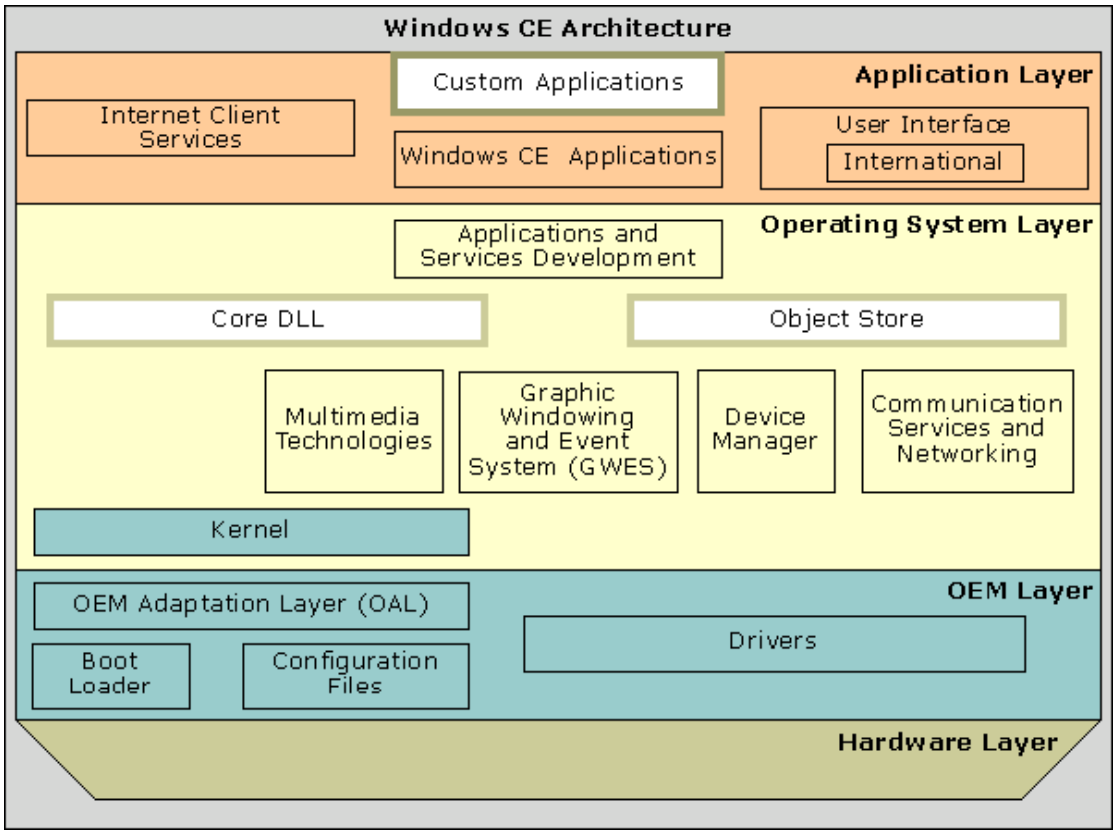

Figure 6: The Windows CE .NET architecture [31]

The operating system has been designed using a component based structure such that Original Equipment Manufacturers (OEMs) can choose only the operating system features that they require for their specific hardware platform. Windows CE has been designed for devices with small amounts of memory, storage and CPU processing power; as such it has a much smaller memory foot print than the Windows XP operating system. In only choosing the components they are interested in, OEMs can keep the footprint of their devices small. For example, the Pocket PC operating system is based on the building blocks from the previous version of Windows CE, version 3. It is not however the same as the Windows CE based operating system on all other Windows CE 3 based devices.

The Windows CE API is a subset of the Win32 API. Due to the componentised nature of the OS, different devices can contain different variants of this API, depending on the relevant included components in the OS image deployed on the

device. The Operating System has been designed to support a range of different CPU types, spanning the ARM, MIPS, SuperH (SH), and x86 architectures.

Much of the source code of the operating system is release under a Shared Source license. However parts of the OS source code are not readily available except under a premium source code license through a non-disclosure agreement from Microsoft, including the TCP/IP networking stack. The author of this dissertation has access to such a license, and the remaining operating system source code.

5.3 Windows CE Networking Protocol Stack Architecture

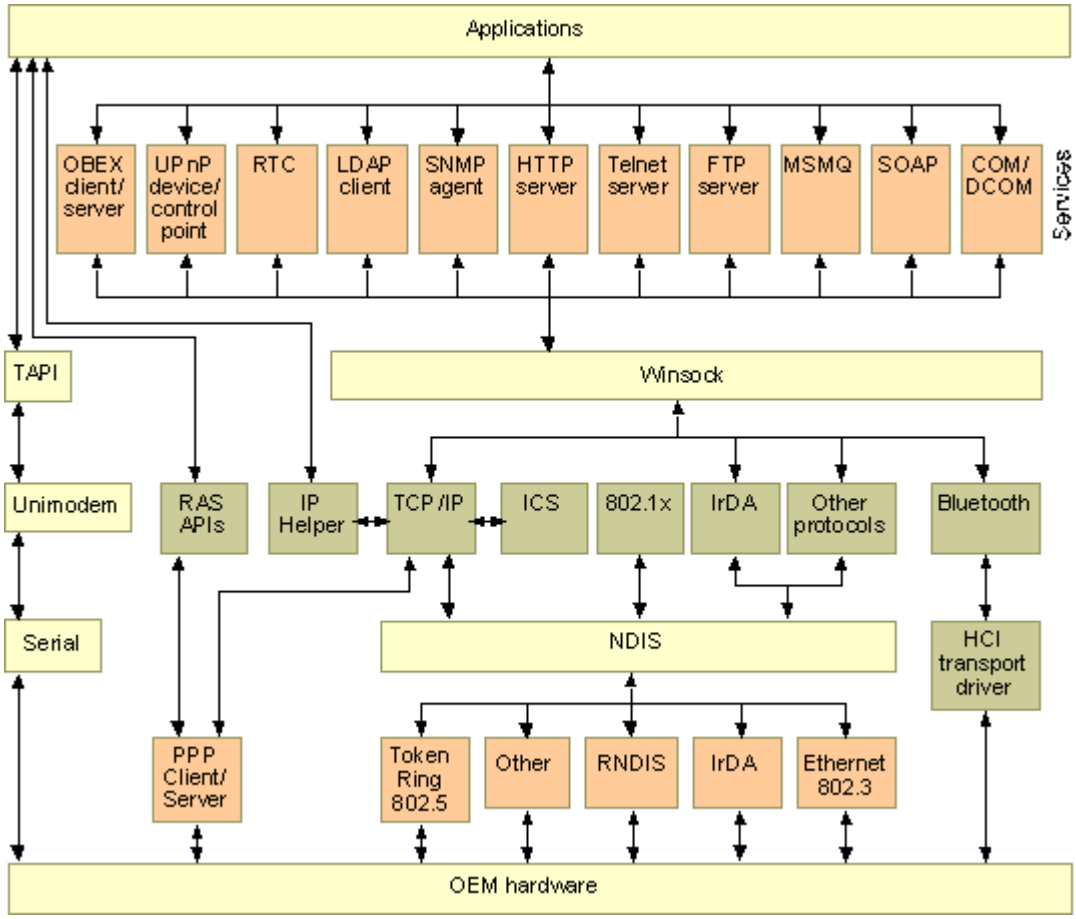

Figure 7: Windows CE Communication Architecture [31]

The Windows CE TCP/IP suite is very similar in architecture to the protocol stack on Windows XP. It consists of *core protocol elements*, *services*, and the *interfaces* between them. The Network Device Interface Specification (NDIS) is a public interface, documented on the Microsoft Developer Network (MSDN), which governs the communication between interface device drivers controlling hardware adapters, and the upper-level protocols, the most common being TCP/IP. The Transport Driver Interface (TDI) is present as the upper edge interface to Windows Protocol implementations like TCP/IP. TDI is a public interface. It is documented on MSDN for Windows XP, but not for Windows CE.

The Winsock DLLs communicate with the TCP/IP stack through the TDI interface. Winsock is the Microsoft Windows implementation of the Berkeley Sockets interface, with some Windows specific extensions. The Winsock interface is part of the win32 API, and is most commonly used by applications to send TCP/IP traffic to other hosts. In Windows XP, kernel mode device drivers cannot access the user mode Winsock DLLs. If a kernel mode driver in Windows XP wishes to send or receive TCP/IP traffic, it must access the TCP/IP stack directly through the TDI interface.

Windows CE removes the barrier between kernel space and user space for device drivers. As such all the networking device drivers in the Windows CE architecture effectively run in protected user mode. Hence, such drivers can link with the Winsock DLLs, and do not need to use the TDI interface directly as in Windows XP. This is relevant to any implementation of AODV written as a device driver, as the AODV routing protocol needs to send UDP control packets to other hosts.

5.4 Packet Filtering Options of the Windows CE Protocol Stack

As mentioned at the beginning of this chapter, the TCP/IP protocol stack implementation in the Windows operating systems does not contain a packet filter/mangling architecture directly similar to Netfilter on the Linux architecture.

However a number of other options for intercepting data and packets through the CE protocol stack are possible. These mechanisms are described here, with a view to assessing their suitability for meeting the requirements of an ad-hoc routing protocol implementation as outlined in section 4.2.

Figure 8 illustrates the protocol stack containing a Winsock application at the top, through the Winsock API and SPI DLLs, through the TDI interface into the TCP/IP stack itself, and on down through the NDIS interface into the Network Interface Card (NIC) drivers. Please note the architecture shown, reproduced from **[33]**, is that of Windows XP. The architecture of Windows CE is the same as that of XP, but the kernel mode drivers (with a .sys extension) are user mode dynamic link libraries (with a .dll extension) in CE.

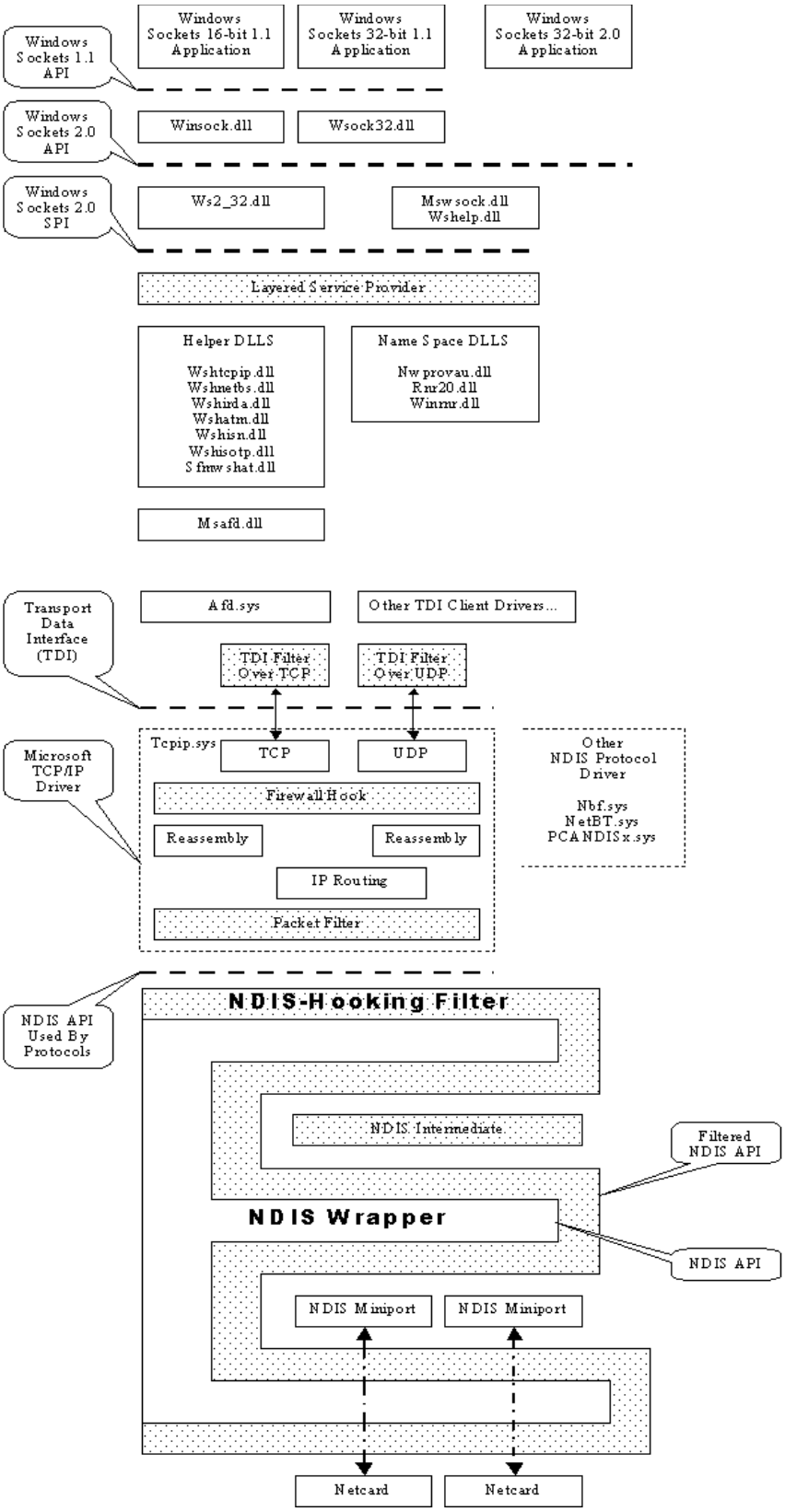

Figure 8: Network architecture diagram of Windows CE, with points where data can be filtered highlighted [33]

5.4.1 Winsock 2 Layered Service Provider

Microsoft defined a new interface with their latest version of Winsock known as the Winsock Service Provider Interface (SPI). The SPI is a standard interface between the Winsock Application Programming Interface (API), which is called by applications requiring Socket functionality, and the protocol stacks. As such, multiple protocol stacks are now supported by Winsock, not just TCP/IP. A Layered Service Provider (LSP) is a driver that implements the Winsock SPI at both its upper and lower edges. It relies on the existing underlying transport driver for its transmission functionality. A typical example of a Layered Service Provider might be to receive data passed into the Winsock API by an application, encrypt it, and send it on down the protocol stack. LSPs can be layered one on top of the other, as long as all LSPs in the chain support the SPI at both their upper and lower edges.

Using a Winsock Layered Service Provider, an ad-hoc routing protocol could intercept Socket open or send requests to an unknown destination. It could buffer the data while a route request takes place, and then release the data down to the protocol stack. It could determine when to update a cached route by examining the destination address of data passed through it. Thus it could meet requirements 1, 2 and 3 in section 4.2.

In the situation where a packet is received on one of the host's interfaces containing a next hop address that is unknown to the kernel routing table, the IP layer will discard the packet. A routing protocol implemented as an LSP alone will not have any means of being notified of this event, and so conditions 4, and possibly 5, cannot be met.

Another shortcoming of LSPs is that drivers, and possibly some applications in Windows CE, can bypass Winsock altogether and instead use the Transport Driver Interface to send data packets directly into the TCP/IP protocol driver. This is particularly true in Windows XP, where drivers cannot link with Winsock and as

such must use TDI (unless they use a companion user-level service). As such, on-demand routing could not be performed for such packets, and they most likely would fail to be delivered. In conclusion an LSP alone would not be suitable for implementing ad-hoc routing protocols.

5.4.2 TDI Filter Driver

The upper edge interface of a Windows protocol driver, such as TCP/IP, is the Transport Driver Interface. In Windows XP the TDI driver is a classical NT-style "legacy" driver that uses an I/O Request Packet (IRP) based API. Such an API can be filtered in two ways. The first uses a family of functions, the IoAttachDeviceXYZ API, to layer a filter above TDI. The second method involves filtering the IRP dispatch table for the TDI driver.

The TDI driver on Windows CE is not fully documented.

The type of filtering operations that can be performed with this driver is very similar to the operations that can be performed using a Winsock Layered Service Provider. Indeed Microsoft recommend using such an LSP over a TDI filter driver for Windows CE, as there is no kernel mode/user mode distinction between these two drivers in CE, and an LSP is much easier to program. In Windows XP, the Winsock LSP is a user mode driver, whereas the TDI filter driver is a kernel mode driver. Also, unlike an LSP, *all* IP traffic must pass through the TDI interface.

A TDI driver alone would not be suitable as an on-demand ad-hoc protocol implementation, for the same reasons as a Winsock LSP.

5.4.3 Filter Hook Driver

Filter hook drivers in the Windows networking architecture are somewhat similar to Netfilter in Linux, albeit less powerful and flexible. It is a driver that registers a call-back function with the system supplied IP filter driver. The call-back function then returns a decision on whether to continue processing each packet that passes through

the IP layer (PF_ACCEPT), or to drop the packet (PF_DROP). The position in the networking stack where the filter hook call-back function is called is somewhat similar to the location of the NF_IP_LOCAL_OUT and NF_IP_PRE_ROUTING hooks in Netfilter, i.e. those required for an implementation of AODV. As such, an implementation of an on-demand routing protocol as a filter hook driver could meet requirements 1, 3, 4 and 5 of section 4.2.

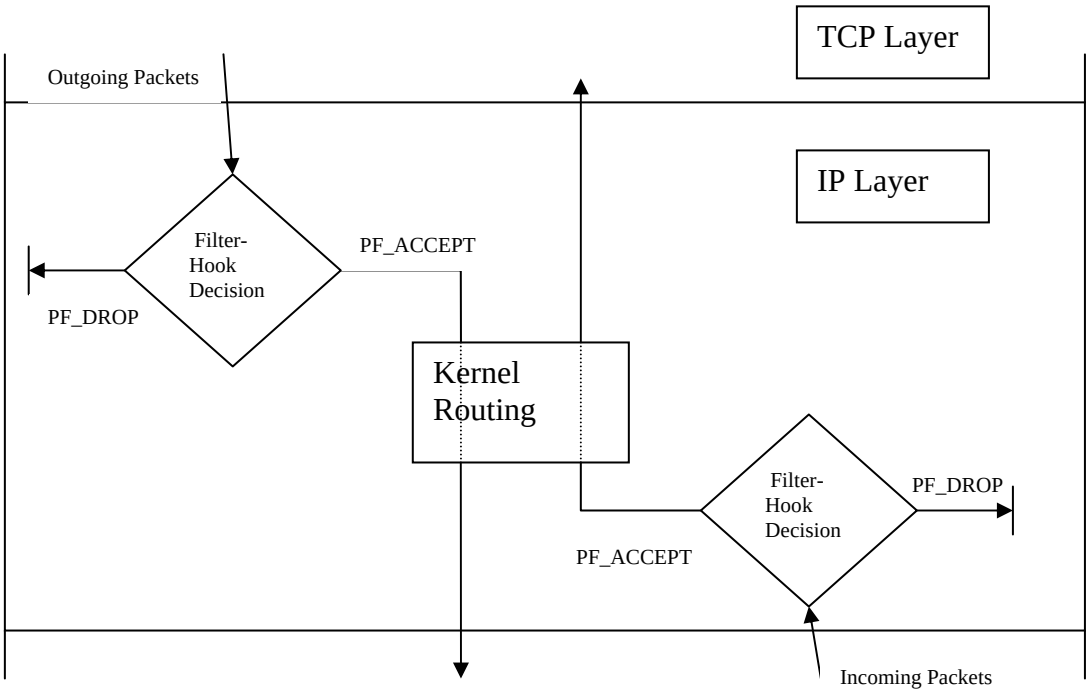

Figure 9: Packet Filter Hook Architecture. Packets arriving from TCP and the underlying network interface are subjected to the filter-hook decision to either pass or drop the packet before kernel routing takes place.

There are two major shortcomings of a filter hook driver. The most significant limitation that restricts its usefulness for on-demand routing protocols is that it cannot deal with packets asynchronously. That is, it cannot remove them from their traversal of the IP layer while it awaits the result of a route discovery cycle. Packets cannot be buffered by a filter hook driver, and so it cannot meet requirement 2 without significantly modifying the structure of the filter hook driver mechanism.

The second shortcoming of the filter hook driver is that only one call-back function can be registered at a time. As such if another application is using the driver, it will

not be available for use by the ad-hoc routing protocol, and vice versa. Netfilter in Linux allows multiple call-back functions to be registered per hook, and they are each called in turn.

5.4.4 Firewall Hook Driver

The firewall hook driver was introduced in beta versions of Windows 2000. It was designed with the intent of providing hooks for firewall implementations to filter packets. The mechanism is no longer supported by Microsoft, and could be removed from future Windows versions. It is not documented on MSDN.

Microsoft does not recommend the use of the firewall hook driver, as it 'ran too high in the network stack' **[31]**. They recommend the use of an NDIS intermediate driver instead. For the purposes of an ad-hoc routing protocol, the firewall hook, like the filter-hook, does not support filtering of packets asynchronously, so it cannot meet requirement 2 of section 4.2.

5.4.5 NDIS Intermediate Driver

In the Windows networking architecture, the Network Driver Interface Specification (NDIS) facilitates communication between the operating system, upper level protocol drivers (such as TCP/IP), and network drivers (that control the hardware network interface cards). The NDIS interface is located between an upper-level protocol driver on the top of the communications architecture, the intermediate and miniport drivers in the middle of the communications architecture, and the hardware network adaptors at the bottom. Thus an NDIS protocol driver like TCP/IP calls functions in the intermediate or miniport drivers, fully abstracted through NDIS, and vice versa.

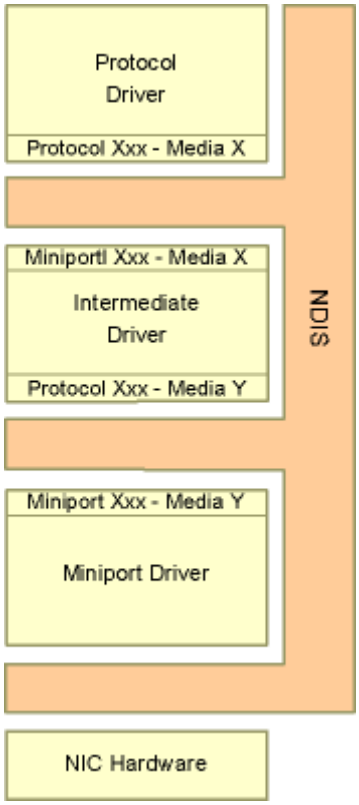

Figure 10: Layered NDIS driver architecture [31]

Network drivers are divided into a class driver and a miniport driver. The class driver is implemented by Microsoft and contains the common functionality of a class of device, e.g. PCI Ethernet cards. The miniport driver is written by the hardware manufacturer and contains the remaining functionality specific to the particular device.

Of particular interest for ad-hoc protocol implementations is the intermediate driver. The intermediate driver does not use NDIS functions to control adapter hardware; instead it is layered on top of another miniport (or intermediate) driver, or legacy device which does not conform to the NDIS specification. The latter type of intermediate driver is responsible for making a lower level legacy interface card appear like an NDIS miniport driver. Of particular interest to ad-hoc routing protocols is an intermediate driver which is layered on top of another miniport (or intermediate) driver. Such a driver, also known as a layered miniport driver, presents a miniport interface to overlying protocol drivers, and a protocol driver to underlying

miniport drivers, as illustrated in Figure 10. As such, to the underlying miniport driver, the intermediate driver appears to be a protocol driver, and to the overlying protocol driver, the intermediate driver appears to be a miniport driver. Such layered miniport drivers can be linked in a chain.

NDIS intermediate drivers can be used to filter packets and perform data mangling operations on them. For example, it can be used to encrypt or decrypt packets.

For the purposes of writing an ad-hoc routing protocol, packets can be buffered by intermediate drivers while awaiting the results of a route discovery cycle. Requirement 2 of section 4.2 can be met. The last time of use of an active route can be determined by inspecting packets from that route. Requirement 3 is met. Packets arriving with a next hop destination for which this node does not have an entry in its routing table can be observed by the intermediate driver for the purposes of sending route error messages. This meets requirements 4 and 5.

The only difficulty is with requirement 1, the ability to determine when a Route Request is needed, but this can be overcome. Normal operation for the IP layer is to discard packets for which it does not know the next-hop IP address, as is the case for a packet to a destination for which a route has not yet been discovered. This is a problem, as an intermediate driver implementation will not know to initiate a route discovery, and the first packets will have been discarded already in any case. To overcome this, we need to temporarily convince the IP layer that there is a valid next-hop for this destination. A default root to a fake IP address can be set up for such unknown destinations, and an ARP cache entry can be manually entered for this address to prevent a failure during ARP lookup.

Now packets will be sent down from the TCP/IP protocol driver to the intermediate driver with incorrect routing information. These packets can be queued in the intermediate driver while the route discovery cycle completes. Once the next hop IP address is known, the corresponding MAC address must be filled in in the Ethernet frame, and the packet transmitted. The MAC address of the next-hop can be

obtained directly from the incoming Route Reply packet, as this will have originated from the relevant next-hop. The kernel route table can be updated such that subsequent packets for this destination will be routed correctly in the IP layer, using the IP Helper API in Windows CE.

This approach has some disadvantages. First of all the routing for packets of as-yet undiscovered routes is replicated below the IP layer. This is wasteful at best. Second, such an approach will not be independent of the hardware type being used. By using layer 3 (IP) addresses, and relying on the conventional mechanism (ARP for Ethernet) for translating layer 3 addresses to layer 2 addresses (MAC addresses), an ad-hoc routing protocol can operate without knowledge of the data-link layer over which it is operating. An ad-hoc routing protocol implemented as an intermediate driver will need explicit knowledge of the data-link layer over which it is operating (e.g. 802.3, 802.11), in order to correctly modify the layer 2 address in the layer 2 frame.

5.4.6 NDIS Hooking Filter

Using an NDIS hooking filter, drivers intercept or ‘hook’ selected functions exported by the NDIS wrapper. Thus an NDIS hooking filter can be used to perform functionality similar to what is possible using an NDIS intermediate driver.

For the purposes of writing an on-demand ad-hoc routing protocol implementation in Windows CE, a NDIS hooking filter does not seem to offer any significant advantages over a simpler NDIS intermediate driver.

6 Windows CE AODV Implementation Design

6.1 Introduction

This chapter presents the design for the Windows CE AODV implementation. Section 6.2 presents some design approaches, discussing the advantages and disadvantages of each. Section 6.3 describes the adopted design, on a module by module basis.

6.2 Design Approaches

Further to the discussion of the Windows packet filtering mechanisms in Chapter 5, this section discusses some specific possible methods of implementing AODV, and outlines the reasons for the chosen approach.

6.2.1 Embedding AODV within the TCP/IP driver

With access to the source code for the Windows TCP/IP protocol driver, it is possible to directly modify the routing code with AODV functionality. An advantage of this approach is that it would be efficient.

However such an approach is unattractive for a number of reasons. The source code for the TCP/IP driver in Windows is proprietary to Microsoft, and cannot be viewed, modified or redistributed in binary form without special license. Currently on-demand ad-hoc routing protocols are in a relatively early stage of research. It is likely significant changes to such protocols, as well as new protocols, will appear in time. As such extensibility of the mechanism used to implement such protocols is an important requirement. By tightly coupling the TCP/IP driver with the ad-hoc routing protocol, it becomes extremely difficult to modify and update the protocol.

6.2.2 Implementing AODV as an Intermediate Driver

Section 5.4.5 introduced the NDIS intermediate driver, and described how the AODV routing protocol could be implemented using this mechanism.

The advantages of such an approach is that it is easy to install, it would be easy to port to other Windows versions which use NDIS, including Windows XP, and it would be possible to get such a driver signed by Microsoft under the Windows Hardware Quality Labs (WHQL) scheme.

The disadvantages include that such an implementation could be seen as being too low in the networking stack: as a filter mechanism between the networking layer and data link layer of the OSI model. Packets for an unknown route will have to be rerouted by the AODV intermediate driver after they are 'coaxed' out of the IP layer. Also, because the routing protocol is tightly coupled with the data link layer in this implementation, it is not independent of the specific transmission mechanism being used (e.g. 802.3, 802.11, HiperLAN, etc).

6.2.3 Modifying the Filter Hook Mechanism

As described in section 5.4.3, the filter-hook driver mechanism comes close to meeting the requirements for an on-demand ad-hoc routing protocol. The main shortcoming of the mechanism is that it cannot be used to deal with packets asynchronously: it must either accept the packet for transmission immediately, or discard the packet. Thus packets cannot be buffered while a route discovery cycle takes place.

It is possible with significant effort to modify the filter-hook mechanism, or more likely to introduce a new similar mechanism, such that packets can be removed and later re-injected into the IP layer, to provide functionality similar to that of Netfilter in the Linux operating system.

By this mechanism an implementation of AODV would consist of a separate driver that communicates with the TCP/IP driver using I/O Control Codes (IOCTLs). The IP layer would export IOCTLs for registering a call-back function to be called with a packet as a parameter as the packet traverses the relevant hooks. The AODV driver on initialisation would use the exported IOCTL to register its call-back function. The function would return a value indicating the packet should immediately continue its traversal of the network stack, should be immediately discarded, or should be removed from its traversal to be reinjected at a later stage. The IP layer will also export a mechanism for the attached filter driver to reinject packets processed asynchronously.

The advantages of such an approach are that porting effort for future ad-hoc protocols between Linux and Windows would be greatly reduced, and the new hooking mechanism would be very suitable for meeting the requirements of ad-hoc routing protocols. Such a mechanism would also be useful to many other applications that require asynchronous packet filtering and mangling facilities, similar to those which use Linux Netfilter.

Difficulties with this approach include that the code for the TCP/IP driver in Windows is proprietary Microsoft code, and cannot be viewed or modified without special license. Distributing such a mechanism would not be possible unless the mechanism is adopted by Microsoft for future versions of their operating systems. Installing such a mechanism on existing operating systems would not be straight forward.

6.2.4 Providing System Services Directly for Ad-Hoc Routing Protocols

An interesting approach would be to modify the TCP/IP driver to export IOCTLs to an external driver for the purposes of providing system services directly relevant to ad-hoc routing protocols. These would perhaps take the form of providing mechanisms for meeting the requirements outlined in section 4.2.

Kawadia, et al. **[28]** have made some progress towards an on-demand routing protocols API for the Linux platform, in the form of their Ad-hoc Support Library (ASL).

This is an interesting area that warrants further research.

6.3 Design Description

Two approaches were considered. The main approach used in the ad-hoc protocol for this Masters dissertation has been the intermediate driver approach described in sections 5.4.5 and 6.2.2. This approach was chosen as it is the only approach that does not require large changed to the TCP/IP protocol driver, and as such it is the only form of such a driver that would be easy to install and distribute.

In addition, extensive kernel modifications were performed to alter the filter-hook mechanism to provide asynchronous packet filtering facilities directly in the IP layer, as described in section 6.2.3.

A working implementation was achieved in both cases.

Some of the code in both cases was based on the code of the NIST Kernel AODV implementation for Linux by Luke Klein-Berndt **[9]**. The modules for our Windows version are described in overview next.

6.3.1 Module Description

The AODV code is written in C. As such it consists of a number of modules, whose functionality is described here.

aodv.h

This is an include file containing some important macros and type definitions. A number of important structs are defined in here, including the AODV control message types (RERR, RREP, RREQ), and various linked list structures for the AODV route table, precursor entries, the timer queue, event queue, etc.

aodv_driver{.c, .h}

This driver has the dual purpose of initialising the NDIS intermediate driver (or filter-hook driver), and the AODV structures. It contains the DriverEntry function which is the first entry point called in an intermediate driver. Its purpose is to register the intermediate driver with NDIS. It also initialises the AODV structures, and starts the event_queue thread. This module also contains the clean-up function which is called when the driver is unloaded by NDIS.

aodv_thread{.c, .h}

As control packets are received in the intermediate driver (or filter driver) on an interrupt, they are placed as an entry in the event_queue structure. To prevent doing a lot of processing on interrupts, the packets are processed by a separate thread which is created and managed in this module. The thread sleeps until a new control packet arrives. The control packet is placed in the event_queue list, and the aodv thread is woken. The types of events to be processed are:

EVENT_RREQ: occurs when a Route Request message is received on one of the node's interfaces.

EVENT_RREP: occurs when a Route Reply message is received on one of the node's interfaces. Since HELLO messages are Route Reply messages with a hop count of zero, HELLO messages are also processed with this event.

EVENT_RREP_ACK: a node can request by setting a flag in its Route Reply message that it should receive an explicit acknowledgement in the form of a Route

Reply Acknowledgement message. The acknowledgement is handled with this event.

EVENT_RERR: occurs when a Route Error message is received on one of the node's interfaces.

EVENT_CLEANUP: occurs periodically, and is used to clean up inactive routes in the route table, and the flood_id_queue. This event is generated internally, and not in response to an external control message as for the other events.

event_queue{.h, .c}

The event_queue module maintains a linked list of event_queue_entry structs. The event queue is a First-In First-Out (FIFO) structure. The module contains functions to initialise the queue, insert entries, remove the next entry, and cleanup the queue. Event queue entries consist of AODV control packets and cleanup events, and are used such that the bulk of the AODV routing protocol processing occurs in the AODV thread, and not in an interrupt thread.

flood_id_queue{.h, .c}

This module maintains a linked list of flood_id_queue_entry structs. When Route Requests are flooded through the network, a node must maintain a memory of the last number of Route Requests from a source that it has rebroadcast. It must refrain from rebroadcasting the same Route Request (received from different sources) more than once. Before processing a Route Request, a node will check the flood_id_queue and not reprocess the Route Request if it has seen it before.

The module provides functionality for initialising and cleaning up the queue, for searching for entries, for inserting new entries, and for deleting stale entries.

interface_list{.h, .c}

This module maintains a linked list of interface_list_entry structs. During initialisation of the AODV driver, a function is called in this module to initialise the

interface list entries. These consist of information about the interfaces such as IP address, hardware address and interface index. The struct also contains a Socket descriptor which is used for broadcasting UDP control messages from the particular interface. The module contains functions for managing the list and for finding particular interfaces by IP address or interface index. It also contains a function, start_HELLO, which starts sending HELLO messages out on a given interface.

miniport{.c, .h}

This module is relevant to the NDIS intermediate driver implementation, and not the modified filter-hook driver. It contains the NDIS miniport interface that is exported at the upper edge of the NDIS intermediate driver. As such, packets being sent from the TCP/IP protocol driver arrive in this module. From here they are passed to the packet_out module for AODV processing, before being passed down to the underlying miniport (or another intermediate) driver.

neighbour_list{.c, .h}

This module maintains a linked list of nodes directly accessible over the wireless interface from this one (i.e. within one hop). Each neighbour_list struct entry contains the neighbour's IP address, hardware address, the interface through which it can be contacted, and the route table entry for this neighbour. The module contains functions for managing this list. When entries in this list are timed out, this may initiate the sending of a Route Error message.

packet_in{.c, .h}

When a packet is received, either through the intermediate driver's lower-edge protocol interface, or the modified filter-hook driver's incoming hook, it is sent to a function in this module for processing. Only AODV packets (those UDP packets destined for the AODV port) are examined. Firstly the format of the packet is checked to see that it is a properly formatted AODV packet. Next, if the packet is a unicast packet (such as a Route Reply message) destined for another node to which we no longer have a route table entry, a Route Error message is sent. Next, the

lifetime of the route from the source is updated, and the packet placed in the event queue for processing.

packet_out{.c, .h}

Packets received through the intermediate driver's upper edge miniport interface, or the modified filter-hook driver's outgoing packet hook, are filtered in a function in this module. Unicast packets for which we have a route, or broadcast packets, are passed through without modification. Unicast packets for which we have no route, and hence for which a route discovery cycle is required, are passed to the packet_queue module for buffering, and a Route Request is initiated. The packets will later be reinjected (or dropped) when the route discovery cycle succeeds (or fails). The lifetime for valid routes is updated when unicast packets are sent on this route.

packet_queue{.c, .h}

This module maintains the queue of IP packets to be buffered while a route discovery cycle is being performed. It provides the ability to queue packets, and then either reinject them to the protocol stack at a later time, or drop them, depending on whether the route discovery cycle was successful. The queue is currently maintained as a linked list, FIFO structure. We intend to improve this by implementing it is a hash-table of doubly linked-lists, with the hash-table keyed by destination IP address of the route.

protocol{.c, .h}

Similar to the miniport module, this is specific to the NDIS intermediate driver implementation. Packets arriving at the lower edge of the intermediate driver (from the miniport driver, or a lower layered intermediate driver) from other hosts are filtered through this module. They are first passed to the packet_in module for filtering, and then released up to the overlying protocol driver for processing.

rerr{.c, .h}

This module maintains all the functionality required for dealing with Route Error messages, including their construction, processing link-breaks to send out Route Errors, handling route expirations, unreachable hosts, and receipt of Route Error messages from other nodes.

route_table{.c, .h}

This module maintains all the needed routing information for contacting other nodes. It provides functions for managing the route_table_entry structures, including the creating and deleting of entries, and deleting of invalid entries. It provides functionality for adding and deleting precursor entries to and from a route table entry. Finally, it uses the Windows IP Helper API to manage the kernel routing table, to add and delete appropriate entries.

rrep{.c, .h}

This module provides the functions necessary for correct handling of route reply messages, including receiving HELLO messages. It is passed packets from the AODV thread, and appropriate action is taken.

rrep_ack(.c, .h}

This module provides two simple functions for sending and receiving Route Reply Acknowledgements. Received acknowledgements are ignored.

rreq{.c, .h}

This module provides the functionality for handling Route Requests. Received Route Request packets are passed into it from the AODV thread, and they are processed as required here. This module also exposes the function required for generating and sending a Route Request for a particular destination.

timer_queue{.c, .h}

There are a number of operations within an AODV implementation that require specific timing. For example, HELLO messages are sent at a periodic interval,

Route Requests are rebroadcasted after a certain interval, etc. This module maintains a queue of timed events, sorted in increasing time of occurrence, such that the timed item to occur soonest is always at the front of the list. A separate thread runs to perform the timed operations. It sleeps until the time that the next timer_queue_entry is due (maintained as an absolute time in milliseconds according to the system clock). It then wakes up, performs any due timer items, and sleeps until the time the next new item is due. The following timed items are possible:

EVENT_RREQ: occurs after a Route Request has been sent, but no Route Reply has been received in a certain time. The Route Request will either be resent (according to the number of times it is configured to be resent), or cancelled and any queued packets from this route will be dropped.

EVENT_HELLO: HELLO messages are sent at a certain interval for each interface. When a HELLO message is sent, another EVENT_HELLO message event is placed in the timer queue, and set to occur in HELLO_INTERVAL seconds.

EVENT_CLEANUP: places an EVENT_CLEANUP event in the event_queue, and another EVENT_CLEANUP event in the timer_queue, such that cleanups of the routing table and flood_id_queue occur at periodic intervals.

EVENT_NEIGHBOUR: when, as a result of the receipt of a HELLO message, a new neighbour is added to the neighbour list queue, or the lifetime of an existing one is updated, then this entry must be set to expire after a certain timeout. This timer item is used for this purpose. It is not executed as long as the neighbour continues sending HELLO messages that are received at this node.

utils{.c, .h}

This module provides a number of utility functions necessary for the AODV routing protocol implementation, including the handling of Sockets (opening, closing, etc.) and the sending of messages (either broadcast or unicast) over these Sockets. It includes functions for getting the current system time in milliseconds since 1601, for

converting IP addresses between binary and decimal string notation form, and various other IP address manipulation and comparison functions.

7 AODV Implementation Evaluation

7.1 Introduction

This chapter presents the results of the evaluation of our Windows CE implementation of the AODV routing protocol. Section 7.2 describes the results of some interoperability tests with a number of other AODV implementations for Linux. Section 7.3 presents a qualitative analysis of the design approaches taken.

7.2 Interoperability

An AODV interoperability event **[13]** was held in the University of California, Santa Barbara in March 2002. The aim of the event was to test the interoperability of the different implementations available at the time. Most of the implementations described in section 4.4 were tested. To test the interoperability of our Windows CE implementation with Linux implementations, the same tests were carried out against the NIST Kernel AODV implementation, and the Uppsala University implementation. The tests are described in the following sections.

Four computers were used for the tests. Windows CE, with our NDIS intermediate driver implementation of AODV, was installed on two Dell Optiplex desktop machines, with Pentium III processors and 256MB of RAM. The Linux operating system, running the NIST and Uppsala implementations, was run on two Fujitsu Siemens B series Lifebook laptops, with Pentium Celeron processors and 256MB or RAM.

The test setup involved using wired Ethernet links (802.3), where all nodes were connected through a hub. To simulate point-to-point links, such that some nodes could only communicate with each other via multi-hop routes through other nodes, link-layer packet filtering was used. A small program called ‘Mackill’, developed at

Uppsala University **[11]**, was used to drop all packets originating from specified MAC addresses on the Linux boxes. To filter packets on Windows CE, we wrote an NDIS intermediate driver with an IOCTL interface defined for accepting MAC addresses for filtering. A companion user-mode application was developed to accept user input of the MAC addresses, and to pass these addresses to the filter intermediate driver using the defined IOCTL interface.

7.2.1 Hello and Ping

This tests two directly connected nodes of different implementations. Each node periodically broadcasts HELLO messages. The connected node receives the HELLO messages, and installs a route to the other node. Each node should then ping the other node and ensure the reply is received. The following actions are verified:

- Correct reception of neighbouring HELLO messages.
- Correct installation of route to neighbouring node.
- Deletion of route when nodes are disconnected.

7.2.2 2-hop RREQ/RREP

Nodes are configured in the topology 1-2-3, where node 1 is the source, and node 3 the destination, using two different implementations, X and Y. The configurations X-X-Y, and X-Y-Y, are tested. The first node pings the last node; a Route Request must be issued through the middle node. The following actions are verified:

- Node 1 issues a RREQ for node 3.
- Node 2 receives the RREQ, and replies with a RREP.
- Node 1 receives the RREP, install the route, and pings are correctly received.

7.2.3 RERR

Nodes are configured in the same topology as the previous test. After creating a route between node 1 and node 3 as in the previous test, the 2-3 link should be disconnected. The correct receipt of a Route Error at node 1 demonstrates the correct operation of precursor nodes. The following is verified:

- Node 2 issues a RERR for node 3 and removes its route to node 3.
- Node 1 receives this RERR, and removes its route to node 2 also.

7.2.4 Re-route

This test tests the ability of the protocol to find a different route to a node once the previous route breaks. Four nodes are configured in the topology 1-2-4, and 1-3-4. As such, node 1 can communicate with node 4 through either node 2 or node 3. Nodes 1 and 2 comprise implementation X, and nodes 3 and 4 comprise implementation Y. The test is repeated with implementations X and Y reversed. A route 1-2-4 is established. The link 2-4 is broken, and a new route 1-2-3 should be discovered. The following actions are verified:

- After the link 2-4 breaks, node 2 sends a Route Error to node 1.
- Node 1 deletes the route to 4, and issues a new Route Request.
- Node 3 replies to the Route Request with a Route Reply.
- Node 1 receives the Route Reply, and installs a route through node 3 to node 4.

7.2.5 3 hops RERR

This is the same as test 3, but with an additional node such that the Route Error traverses an additional hop. Four nodes are configured as 1-2-3-4. Nodes 1 and 2 are configured as implementation X, nodes 3 and 4 as implementation Y, and then the tests are repeated with the reverse configuration. After a route between nodes 1 and 4 is established, the link 3-4 is broken. The following actions must be observed:

- Node 3 deletes the route to node 4, and sends a Route Error to node 3.
- Node 2 receives the Route Error, deletes the route, and sends the Route Error to node 1.
- Node 1 receives the Route Error and deletes the route to node 4.

7.2.6 Results

The Windows CE implementation was found to interoperate correctly with the Linux implementations under all the provided scenarios. In addition, the scenario outlined in section 7.2.5 was modified such that node 4 provided gatewaying services to the fixed Internet (i.e. it had one network interface connected to the ad-hoc network, and another to the fixed internet. The Linux IP tables tools was used to set up this node to provide Network Address Translation **[48]**). This scenario demonstrates accessing a network service such as HTTP on a Windows CE device using a heterogeneous ad-hoc network with both Windows CE and Linux devices.

7.3 Design Analysis

Two different implementation strategies were employed for this dissertation. The first involved modifying the packet filter-hook mechanism of the Windows TCP/IP protocol stack. The second involved implementing the protocol as an NDIS intermediate driver.

The filter-hook mechanism is well situated in the protocol stack for processing packets before kernel routing takes place. Its main drawback is that it cannot buffer packets while a route discovery cycle takes place. For this implementation we had access to the TCP/IP kernel sources from Microsoft, and we modified the mechanism to provide for packet buffering. This was quite a successful approach, and makes for porting Linux Netfilter-based ad-hoc routing protocol implementations quite easy. The main drawback to this approach is that the implementation cannot be easily distributed: it would require permission from Microsoft as the TCP/IP stack is proprietary code, and it would require replacing the TCP/IP stack. This is difficult in the latest Windows operating systems, as they employ system file protection mechanisms.

On the other hand, the NDIS intermediate driver approach is easy to install and distribute. It does not require any changes to proprietary code. However there are other drawbacks to this approach. One of the main drawbacks is that independence of the underlying data link layer is lost. When a packet for which a route discovery cycle takes place reaches the intermediate driver, it has already gone through kernel routing, and ARP. As such, when the route discovery cycle completes it is necessary to insert the hardware destination address on the Ethernet frame. This effectively limits this implementation to interoperating with Ethernet (802.3) and wireless Ethernet (802.11), and other data-link layers directly supported. If the routing protocol was to be used on any other data-link layer, then support would have to be explicitly added for this. This is not the case for an implementation within the IP layer, such as the packet filter-hook mechanism. In addition, there is a minor overhead associated with rerouting packets which are buffered during a route discovery cycle in the intermediate driver (involving correcting the hardware destination address).

8 Conclusion

The contribution of this work has been to produce a real-world implementation of AODV for Windows CE, suitable for running on mobile and embedded devices, such as palm-tops and PDAs. This dissertation showed that our implementation can successfully interoperate with other Linux based implementations of AODV, such that users in an ad-hoc network may co-operate and share networking services with a wide variety of users running heterogeneous operating systems.

A study of the possible approaches to implementing the AODV routing protocol, given the poor support in current operating systems for ad-hoc routing protocols, has been presented. This work should prove of great use to future on-demand ad-hoc routing protocol implementers for Windows. In addition the system services that an operating system should provide to an ad-hoc routing protocol have been described. The next generation of operating systems should take these into account when reviewing their networking protocol stacks.

We have also provided a platform on which future performance studies of AODV will be performed. We intend to test our implementation for use on the soon-to-be deployed Wireless Ad-hoc Network for Dublin (WAND), an ad-hoc network test-bed that is to be deployed around the campus of Trinity College Dublin, and parts of Dublin's inner city. The network will consist of a number of base stations equipped with 802.11 wireless network cards, and running the AODV routing protocol. Users of the network will be able to use Windows CE based devices with our AODV implementation installed to access the services of the network, and to effectively extend the range of the network by offering packet-forwarding services to other users of WAND.

8.1 Future Work

With the recent deployment of the WAND infrastructure, we intend in the near future to perform a performance comparison between our implementation and other AODV implementations. This will also provide some very interesting insights into the operation of a real-world metropolitan area ad-hoc network, such as WAND. Relevant performance characteristics will include route discovery time, and packet loss rate due to route breakages and consequent rerouting during transmission.

Another area that warrants further study is the problem associated with packet loss due to Lundgren, et. al.'s 'gray zone' problem **[7]**: that broadcast HELLO messages are more likely to be received at the limits of a node's transmission range than unicast data packets. This is because HELLO messages are typically smaller than data packets, and so less likely to suffer from bit errors. Also, broadcast messages are sent at the lowest rate in 802.11, 2Mbps, and so are more likely to be successfully received. As such, the route maintenance process maintains routes without sending Route Error messages for longer than it should, and chooses new routes before nodes are sufficiently close to each other to communicate effectively. Using link-level feedback from the 802.11 Media Access Control (MAC) layer would be an important mechanism to be exploited to help solve this problem. Current operating systems and network interface card firmware do not adequately provide such information **[29]**.

As described in section 6.2.4, perhaps the most interesting long term solution for providing simple mechanisms for implementing new ad-hoc routing protocols would be to modify the kernel of operating systems to export an API suitable for such protocols. Such an approach would require significant modification to protocol stacks, including the ability to provide routing protocols with certain notifications, such as when a route discovery is required, and to introduce certain capabilities, such as the buffering of packets while a route discovery cycle takes place.

Bibliography

[1] The Official IETF working group Manet webpage, http://www.ietf.org/html.charters/manet-charter.html

[2] C. E. PERKINS AND P. BHAGWAT. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. Proceedings of the SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications, August 1994.

[3] D. B. JOHNSON, D. A. MALTZ, Y-C HU AND J. G. JETCHEVA. *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)*, Internet Draft, draft-ietf-manet-dsr-07.txt, *work in progesss*, February 2002.

[4] Z. J. HAAS, M. R. PEARLMAN AND P. SAMAR. The Zone Routing Protocol (ZRP) for Ad Hoc Networks, Internet Draft, draft-ietf-manet-zone-zrp-02.txt, work in progesss, July 2002.

[5] V. PARK AND S. CORSON. Temporally-Ordered Routing Algorithm (TORA) Version 1 Functional Specification, Internet Draft, draft-ietf-manet-tora-spec-04.txt, work in progress, July 2001.

[6] Thomas Clausen and Phillipe Jacquet. Optimized Link State Routing Protocol, 2003. IETF Internet Draft.

[6] L. M. FEENEY. A Taxonomy for Routing Protocols in Mobile Ad Hoc Networks, SICS Technical Report T99/07, October 1999.

[7] H. LUNDGREN, E. NORDSTRÖM AND C. TSCHUDIN. Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks. The Fifth

International Workshop on Wireless Mobile Multimedia (WoWMoM'02), September 2002.

[8] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. RFC 3561 – Ad hoc On-Demand Distance Vector (AODV) Routing. July 2003.

[9] NIST Kernel AODV homepage. Luke Klein-Berndt. http://w3.antd.nist.gov/wctg/aodv_kernel/. September 2003.

[10] UCSB AODV homepage. Ian Chakeres. http://moment.cs.ucsb.edu/AODV/aodv.html. September 2003.

[11] UU AODV homepage. Erik Nordström. http://user.it.uu.se/~henrikl/aodv/. September 2003.

[12] UIUC AODV homepage. Including ASL library. Binita Gupta. http://sourceforge.net/projects/aslib/. September 2003.

[13] Report on the AODV Interop. Elizabeth M. Belding-Royer. UCSB Tech Report 2002-18, June 2002

[14] Experiment and Evaluation of a Mobile Ad Hoc Network with AODV Routing Protocol. Kalyan Kalepu, Shiv Mehra, Chansu Yu. Cleveland State University Tech Report.

[15] Netpipe software. http://www.scl.ameslab.gov/Projects/Netpipe/

[16] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In Mobile Computing and Networking, pages 85-97, 1998.

[17] T. Camp, J. Boleng, and V. Davies. A Survey of Mobility Models for Ad Hoc Network Research. Wireless Communications & Mobility Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications, 2(5):483-502, 2002.

[18] S. Corson and J. Macker, RFC 2501: Mobile Ad Hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations, January 1999. Status: INFORMATIONAL

[19] Per Johansson, Tony Larsson, Nicklas Hedman, Bartosz Mielczarek, and Mikael Degermark. Scenario-based Performance Analysis of Routing Protocols for Mobile Ad Hoc Networks. In MobiCom '99. Seattle WA, August 1999.

[20] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, Mobile Computing, volume 353. Kluwer Academic Publishers, 1996.

[21] D. Johnson, D. Maltz and J. Broch. DSR: The dynamic source routing protocol for multihop wireless ad hoc networks, 2001.

[22] Jinyang Li, Charles Blake, Douglas S. J. De Couto, Hu Imm Lee, and Robert Morris. Capacity of Ad Hoc Wireless Networks. In Proceeding of the 7th ACM International Conference of Mobile Computing and Networking, pages 61-69, Rome, Italy, July 2001.

[23] The Economist. Watch this Airspace, June 2002.

[24] J. Moy. OSPF Version 2, Request For Comments (RFC) 1058, June 1988.

[25] G. Malkin. RIP Version 2, Request For Comments (RFC) 2453, November 1998.

[26] Mad-hoc AODV homepage. http://mad-ho.flyinglinux.net/. September 2003.

[27] E.M. Royer & C.E Perkins. An Implementation Study of the AODV Routing Protocol. Proceedings of the IEEE Wireless Communications and Networking Conference, Chicago, IL, September 2000

[28] Vikas Kawadia, Yongguang Zhang & Binita Gupta. System Services for Implementing Ad-hoc Routing Protocols. Proceedings of International Conference on Parallel Processing Workshops, 2002.

[29] Ian D. Chakeres & Elizabeth M. Belding-Royer. AODV Routing Protocol Implementation, Experiences and Observations.

[30] Netfilter homepage. http://www.netfilter.org/. September 2003.

[31] MSDN homepage. http://msdn.microsoft.com. September 2003.

[32] Dave MacDonald & Warren Barkley. Microsoft Windows 2000 TCP/IP Implementation details. White Paper. 2000

[33] Windows Network Data and Packet Filtering. http://www.ndis.com/papers/winpktfilter.htm. September 2003.

[34] Hui Lei and Charles Perkins. Ad Hoc Networking with Mobile IP. In Proceeding of the 2nd European Personal Mobile Communication Conference, 1997.

[35] Frank McSherry, Gerome Miklau, Don Patterson, and Steve Swanson. The Performance of Ad Hoc Networking Protocols in Highly Mobile Environments, 2000.

[36] C. Perkins and P. Bhagwat. Routing over Multi-Hop Wireless Networks of Mobile Computers, 1994.

[37] C.E. Perkins, E.M. Belding-Royer, and Y. Sun. Internet connectivity for ad hoc mobile networks, 2002.

[38] C. Perkins. Ad Hoc On Demand Distance Vector (AODV) Routing, 1997.

[39] E.M. Royer and Chai-Keong Toh. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks, 1999.

[40] B. Crow, I. Widjaja, J.G. Kim, and P.T. Sakai. IEEE: 802.11: Wireless Local Area Networks, September 1997.

[41] G.A. Halls. HIPERLAN: the high performance radio local area network standard. Electronics and Communication Engineering journal, December 1994.

[42] L. Kleinrock and F. Tobagi. Packet Switching in Radio Channel, Part II – The Hidden Terminal Problem in Carrier Sense Multiple Access and the Busy Tone Solution. IEEE Transactions on Communications, vol.23, pp. 1417-1433, 1975.

[43] Phil Karn, MACA – A New Channel Access Method for Packet Radio, ARRL/CRRL Amateur Radio 9th Computer Networking Conference 1990.

[44] Shugong Xu and Tarek Saadawi, Does the IEEE 802.11 MAC Protocol Work Well in Multi-hop Wireless Ad Hoc Networks? IEEE Communications Magazine, pages 130-137, June 2001.

[45] RFC 2131 - The Dynamic Host Configuration Protocol. March 1997.

[46] RFC 826 – Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. November 1982.

[47] Bjorn Wiberg. Porting AODV-UU Implementation to ns-2 and enabling Trace-based Simulation. Uppsala University, Masters Thesis in Computer Science. December 2002.

[48] RFC 1631 – The IP Network Address Translator (NAT). May 1994.

[49] Angela Doufexi, Simon Armour, Peter Karlsson, Andrew Nix, David Bull. A comparison of HIPERLAN/2 and IEEE 802.11a. Centre for Communications Research, University of Bristol, UK technical report.

[50] The 802.11g standard – IEEE. IBM developer works website. www-106.ibm.com/developerwords/wireless/library/wi-ieee.html. September 2003.