Cluster Based Ad hoc Network Routing Protocol Implementation under Windows® CE.NET

A Project Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

Bachelor of Technology

by
Varun Varshney
Vivek Vishal Shrivastava

Under the guidance of
Prof. Gautam Barua

to the
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
India

April, 2004

Certificate

It is certified that the project entitled "Cluster Based Ad hoc Network Routing Protocol Implementation under Windows® CE.NET" has been carried out under my supervision by Varun Varshney and Vivek Vishal Shrivastava as their B. Tech. project and that this project has not been submitted anywhere else for a degree.

Prof. Gautam Barua
Department of Computer Science and Engineering,
Indian Institue of Technology Guwahati

April, 2004.

Acknowledgements

We are grateful to Prof. Gautam Barua without whose invaluable guidance we couldn't have progressed so far in the same time frame.

We also extend our thanks to Dr. Mike Hall and Mr. Stewart Tensley from Microsoft Research, U.S.A., for their help in solving many technical problems and discussing the technical aspects of the project.

Lastly, we would like to thank Mr. David West, who has provided us valuable insights into the implementation techniques of ad hoc routing protocols.

Abstract

There are a number of implementations of the Ad-hoc routing protocols 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 CRESQ 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 CRESQ routing protocol can successfully be used in an ad-hoc network environment.

Cluster based Routing for End-to-end Security and Qos satisfaction, CRESQ, is an innovative protocol for handling QoS satisfaction in multi hop mobile ad hoc networks. Windows CE .NET, the successor to Windows CE 3.0, combines an advanced real-time embedded operating system with the most powerful tools for rapidly creating the next generation of smart, connected, and small-footprint devices. Support for secure and scalable networking and enhanced real-time processing makes it one of the best operating systems for mobile devices networked in an ad hoc environment. We present here the implementation details of the CRESQ protocol on Windows CE .NET. This report focuses on the application level implementation of the algorithm and provides the framework for integrating the protocol in Windows CE .NET in form of an NDIS intermediate driver.

Contents

List of Figures

Chapter 1

Introduction

This dissertation presents the design, implementation, and evaluation of the CRESQ [4] 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 CRESQ 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 compared with other proposed ad-hoc routing protocols.

The real-world testing of the ad hoc routing protocol has been limited to the Linux Platform. 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 CRESQ. They may, for example, wish to communicate with other users during a meeting where no pre-existing infrastructure is in place.

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 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.

1.1 Dissertation Roadmap

In this section the layout of the remainder of this dissertation is outlined. Chapter 2 discusses ad-hoc networks and QoS in general as well as the QoS requirements in wireless and 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.

Chapter 4 describes the cluster based routing algorithm that we use for QoS satisfaction. We introduce the various parameters involved and the different phases in the algorithm.

Chapter 5 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 6 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 7 presents the user level implementation which was completed in Phase I of our implementation. We discuss the daa structures used and the various parameters that need to be fine tuned through simulations.

Chapter 8 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 9 evaluates the success of our chosen approach. We show how we have tested out application level implementation.

Chapter 10 concludes this dissertation and Chapter 11 presents the future work.

Chapter 2

Ad-hoc Networks

This chapter introduces the concept of a Mobile Ad-hoc Network (MANET) 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.

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 , 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 discusses the concept of quality of service, the specific properties unique to wireless ad-hoc networks, as well as some potential uses of ad-hoc networks.

2.1 Quality of Service

QoS (Quality of Service), refers to the capabilities that enable source applications and network infrastructure to request, setup and enforce deterministic delivery of connection traffic. This ability of the network to selectively provide different delay, bandwidth, packet loss ratios to demanding applications has assumed a lot of importance, owing to the mix of real-time, data and other traffic that networks of today encounter. Different QoS schemes and frameworks have been designed and implemented, so as to enable QoS in different underlying networks like wireless, wireline and on different underlying technologies like ATM, IP etc.

Conergence of different network services is what the present day technology is leading to. All services, including wireline and wireless communications, internet services, FTP, telnet, etc. are expected to converge on the single IP based network. However, these services differ in the very nature of their existence leading to the need for QoS. Applications like FTP, Web Browsing etc., have low priority, low bandwidth requirements and high tolerance to delay. For real time applications, convergence is not that easy as they have stringent delay requirements which can

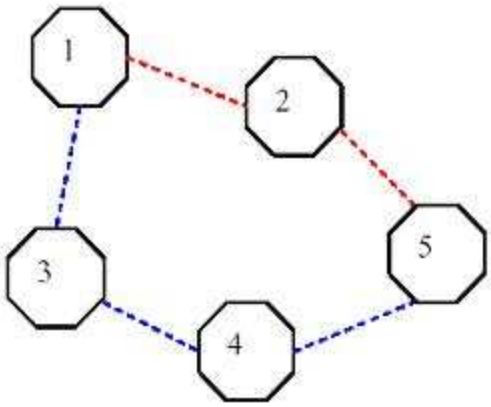

Figure 2.1: An example ad-hoc network. Two different routes exist between nodes 1 and 5. Nodes act as both a host and a router, offering their services to forward packets.

not be assured by the IP's *best effort service*. Hence the flow through a network needs to be divided into different services to provide QoS at different levels.

QoS schemes can be as granular as providing guarantees for each connection by resource reservation (IntServ) or can provide a particular behaviour for a particular class of traffic by prioritization with no guarantees for individual connections (DiffServ).

2.1.1 QoS in wireless networks

As mobile devices become more and more powerful, they will begin to demand the same real time services as available to devices on the wireline networks. But the mobility has a significant impact on the QoS guarantees. The main QoS parameters for real time services are packet delay, packet loss rate, delay jitter and throughput and they suffer from:

- Change in data flow path and packet delay due to mobility.
- Change in packet loss rate due to congestion in the new location of a mobile node.
- Lower throughput due to insufficient bandwidth at the new location.
- Temporary disruption of service during the formation of the new path.

A protocol was proposed for ensuring QoS in wireless networks and that is based on RSVP. It is the *Mobile* RSVP (MRSVP), in which the mobile nodes are expected to be aware of their future positions or positions might be determined by certain algorithms. This information is contained in Mobility Specification or MSPEC. The protocols introduces the concept of proxy agents which are identified by the nodes from the MSPEC and then used for exchanging information regarding the flow specification of the mobile node.

2.2 QoS in ad hoc networks

Many unique problems are faced while ensuring the quality of service in ad hoc networks. Additional challenges in adhoc networks are attributed to mobility of intermediate nodes, absence of routing infrastructure and low bandwidth & computational capacity of the nodes. When an intermediate node vanishes, a new route has to be established which may not provide the desired level of services. Another important issue is that the nodes participating in such networks usually have low computational and bandwidth resources, thus requiring a QoS scheme that is much distributed and that also has a low computational complexity. In fact, being a comparatively newer field, ad hoc networks suffer from a dearth of efficient routing algorithms that have been tested in various scenarios.

One of the major requirements of a QoS scheme for ad hoc networks, thus, is that the time required for hand-off (establishement of a new route due to the disruption of the older one) is minimum. Thus QoS set up during hand off should be done quickly using minimum number of message overhaeads. Only the affected segment of the end-to-end path must have a QoS re-establishment.

Hence, a protocol for mobile ad hoc networks should have a low routing overhead and should be computaionally less complex than any protocol over wireline networks. Also, it should provide a good security and QoS support. Existing protocols for ad hoc networks, like DSDV, AODV, DSR, TORA, etc. aim to achieve the first goal, and they succeed in doing it. Thus the challenge now is to achieve the QoS support. The CRESQ algorithm proposes a way in which this can be done. We now provide a brief overview of ad hoc routing protocols followed by an introduction to the CRESQ protocol before going to our implementation of the same.

Chapter 3

Ad-hoc Routing Protocols

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 neighbors. 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.

3.1 Classification of Ad hoc routing protocols

Ad-hoc routing protocols can broadly be classified into proactive, reactive and hybrid protocols . 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.

3.1.1 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) [1], and Optimized Link State Routing (OLSR).

3.1.2 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 . 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) , and the Ad-hoc On-demand Distance Vector Routing (AODV) and CRESQ protocol are examples of on-demand protocols.

3.1.3 Hybrid Protocols

There exists another class of ad-hoc routing protocols, such as the Zone Routing Protocol (ZRP), 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.

Chapter 4

CRESQ

CRESQ[1][4], [5], [2] is cluster based routing protocol, which effectively uses its clustering in route discovery process to minimize the routing overhead. The clustering is also used in order to provide QoS framework, as it helps in implementation of resource reservation. CRESQ is a kind of source routing, but also provides a way for speedy recovery from a route failure by local route correction. Local route correction is accomplished by replacing the next hop node whose link has failed, by some other node in its cluster.

Each node participating in the protocol is classified either as NONE, SLAVE, BRIDGE or MASTER. A master is the head of a cluster that has bi directional links with more nodes than anyone else in the cluster. A slave is a node that has only one master while bridges have more than one master and are used to communicate across the clusters. A node that has not yet been classified is called none.

4.1 CRESQ algorithm

The CRESQ algorithm works as follows:

1. Initially all the nodes are labelled as none. Then the whole network is clusterized using an initial clustering algorithm. Although, any algorithm can be used, but the proposal recommends the use of the algorithm explained in [3].

2. The cluster management algorithm (Section 4.2) then takes over. It uses HELLO packets for this purpose, as well as for other purposes like QoS management.

3. For establishment of a new connection between S and D, the CRESQ layer in S receives the QoS specification from the QoS layer.

4. CRESQ layer in S sends a ROUTE REQUEST packet to its master M, which broadcasts the packet to its bridges.

5. All the bridges of M further broadcast the packet and packet is received by all the masters of the bridges, which further broadcast to their bridges. This process continues till master

[1]This protocol is explained in detailed form in [2]. We only provide a brief overview here.

of D hears the request, which adds the destination IP address in the *Addr_ List* field of ROUTE REPLY packet and sends to the node from which it heard the request.

6. A master on receiving a ROUTE REPLY, appends the IP of a suitable slave, while a bridge appends its address to the *Addr_ List*. Both then send the reply back to the node from which they received the request.

7. When the source receives the reply, it uses the path specified in *Addr_ List* to route its data packets.

4.2 Cluster Management Algorithm

The cluster management algorithm [2] has two parts, one is executed on trigerring of *Hello_ Event* and the second is executed during the receipt of any packet.

On trigerring of *Hello_ Event*, if a slave hadn't heard from its master recently it marks itself as none, while a bridge and a master removes the masters and slaves, respectively, from which they haven't heard recently. Every node also transmits a GENERAL_HELLO packet during such an event. Note that the *Hello_ Event* is triggered after a fixed emphHello_Interval

On receiving a packet, nodes restamp the master/slave from which they receive the packet. If a slave/bridge receievs a HELLO packet of type BE_MY_SLAVE, then it adds the master in its *Master_ List*, or if the sender is a master then a CAN_I_BE_YOUR_SLAVE packet is send to it. A master adds a slave from which it receives a packet of type CAN_I_BE_YOUR_SLAVE. If it receives a packet from another master, then it becomes the slave of that master after another round of HELLO messages.

4.3 Route Discovery, Establishment and Maintenance

Route discovery is initiated by a node that wants to transmit data to another node and it does not have a cached route to the destination. The process is carried out through a series of ROUTE_REQUEST packets. The source sends a request packet to its master which broadcasts it to its bridges, which in turn broadcast it to their masters. A master in the cluster of ehich the destination lies stops this propagation by sending a ROUTE_REPLY packet to the previous hop of the request. The propagation of the request can be controlled by specifying a TIME TO LIVE.

Route establishment is carried out by the propagation of the reply packets. Bridges add their own IPs while master add the IP of the slave which they deem as fit enough to meet the QoS requirements of the request, to the *Addr_ List*. The source, on receiving the reply, starts sending data packets to the destination using the route specified by the*Addr_ List* of the reply packet.

Route failure is detected by any node on the route when a link is broken. However, the maintenance work is carried out by the master only. Slaves and bridges remove the masters that have failed. Otherwise, they send a ROUTE_FAILURE packet to their master. The master removes the node from its list that has failed and then tries to find a mapping that can substitue the failed node for the route. If such a mapping is successfully found, then the new node is directed all the data of the failed node, otherwise, a ROUTE ERROR packet is sent back to the source.

Chapter 5

Concepts of Existing Ad-Hoc routing protocol Implementations

This chapter characterizes the various publicly available, open source ad hoc routing protocol implementations. There are a limited number of such implementations, mostly for the Linux operating system.

The remainder of this chapter is divided into two 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. Chapter 6 describes the design choices available for the Windows CE platform.

5.1 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 [23], 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 neighbors 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, minimizing 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 CRESQ routing protocol are [23]:

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 routing protocol such that it knows it should send a route error message to the original source or the packet.

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.

5.2 Design Strategies for Linux

The following design strategies for ad hoc routing protocol 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.

5.2.1 Snooping

By snooping the Address Resolution Protocol (ARP) packets and data packets, an ad hoc routing protocol 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 5.1.

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 5.1), or when a packet is received for which we have no routing information (see requirement 4).

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 [22].

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 routing protocol ever knowing it existed. As such this violates the principle that the protocol can operate with networks of nodes of unrelated IP addresses [23].

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.

5.2.2 Netfilter

Netfilter 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 layers 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:

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

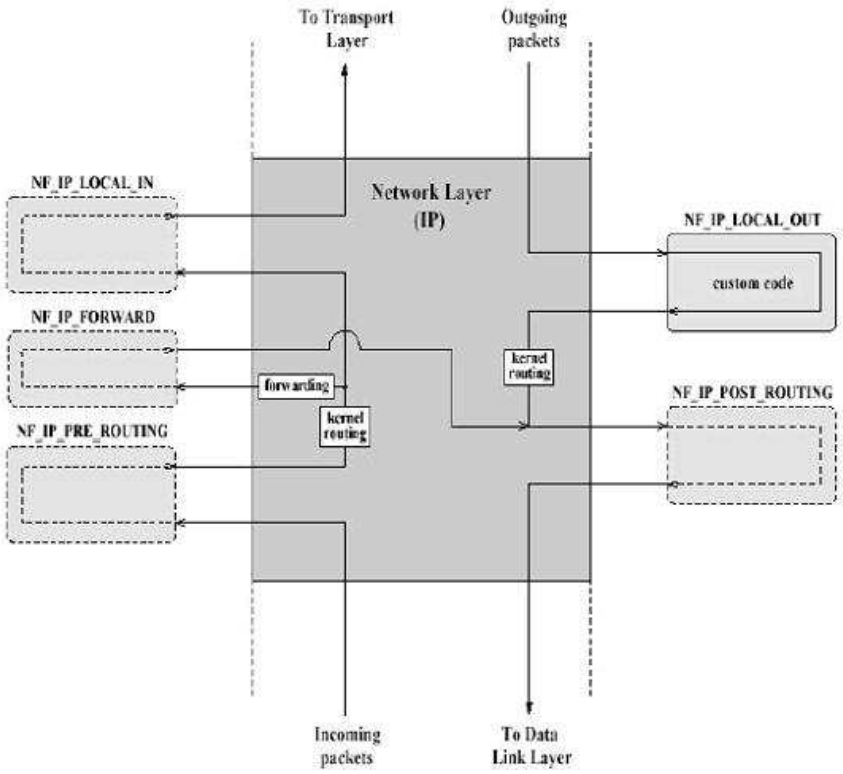

Figure 5.1: The Linux Netfilter Packet Mangling Architecture

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 reinsertion 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 the 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 5.1). It can return a verdict of NF_QUEUE for these packets. This instructs the ip_queue driver to buffer these packets for later reinsertion to the IP layer (requirement 2 of section 5.1). 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 5.1). 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 5.1). Similarly if the 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.

In order to use Netfilter, the 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.

5.2.3 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 5.1. 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 reinsert them; it would maintain timers associated with the route, etc. [23].

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 [23], have performed interesting work towards such an approach using Linux kernel modules and user space libraries, with their Ad-hoc Support Library (ASL).

Chapter 6

Microsoft® Windows® CE .NET

Microsoft Windows CE .NET is an open, scalable, 32-bit operating system (OS) that is designed to meet the needs of a broad range of intelligent hardware devices, from enterprise tools such as industrial controllers, communications hubs, and point-of-sale terminals to consumer products such as cameras, Internet appliances, and interactive televisions. A typical Windows CEbased embedded platform is targeted for a specific use and requires a small-sized OS that has a bundled, deterministic response to interrupts. Windows CE .NET offers the application developer the ease and versatility of scripting languages, along with the versatile environment of the Microsoft Win32 application programming interface (API). It also offers bundled support for multimedia, Internet, LAN, and mobile communications and security services. The windows CE .NET OS architecture is shown in Figure 6.1.

6.1 Modular Structure

We can build a Microsoft Windows CE .NETbased platform using a number of discrete modules. This minimizes the memory needed by the platform. By selecting only those modules that your platform requires, you can minimize the amount of memory that your device requires. A module contains a collection of related application programming interface (API) functions. Some modules are composed of components. Each component can contain a collection of API functions.

6.2 Catalog Features

The Microsoft Platform Builder version 4.2 Catalog consists of a list of board support packages (BSPs), drivers, configurations for core operating systems (OSs), and Platform Manager transports. The items in the Catalog represent the technologies you can select when designing your Microsoft Windows CE .NETbased platform. These technologies are organized and displayed asfeatures. Features are specific implementations of the technologies available in the Windows CE .NET 4.2 OS.

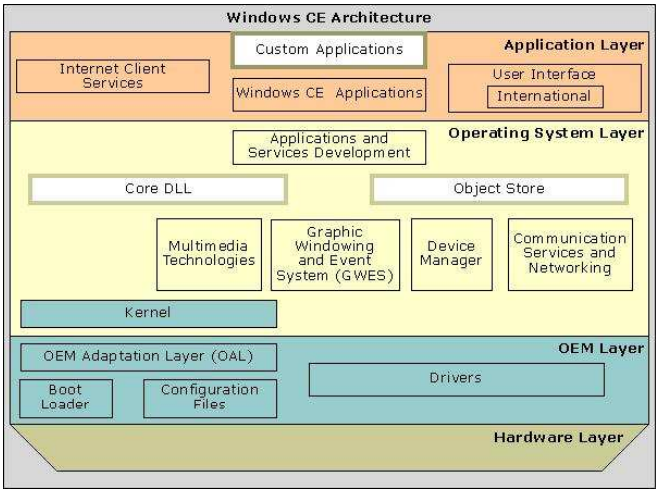

Figure 6.1: Windows CE .NET OS Architecture

6.3 Operating System Features

The core operating system (OS) services provide a common foundation for all Microsoft Windows CE .NET OSs. The services enable low-level tasks such as process, thread, and memory management, and provide some file system functionality.

The Windows CE OS offers a rich set of features that you can use to customize a platform. Component services, networking capabilities, multimedia support, and many other capabilities are contained within individual OS features.

6.4 The Kernel

The kernel, which is represented by the Nk.exe module, is the core of the Microsoft Windows CE operating system (OS). The kernel provides the base OS functionality for any Windows CEbased device. This functionality includes process, thread, and memory management. The kernel also provides some file management functionality. Figure 6.2 shows the general structure, emphasizing the kernel as the conduit for the rest of the core OS components.

We can use the kernel process and thread functions to create, terminate, and synchronize processes and threads and to schedule and suspend a thread. Processes, which represent single instances of running applications, enable users to work on more than one application at a time. Threads enable an application to perform more than one task at a time. Thread priority levels, priority inversion handling, interrupt support, and timing and scheduling are all included in the Windows CE kernel architecture. Together, they provide real-time application capability for time-critical systems.

The Windows CE kernel uses a paged virtual-memory system to manage and allocate program memory. The virtual-memory system provides contiguous blocks of memory, in 1,024-byte or 4,096-byte pages along 64-kilobyte (KB) regions, so that applications do not have to manage the actual memory allocation. For memory requirements of less than 64 KB, an application can use

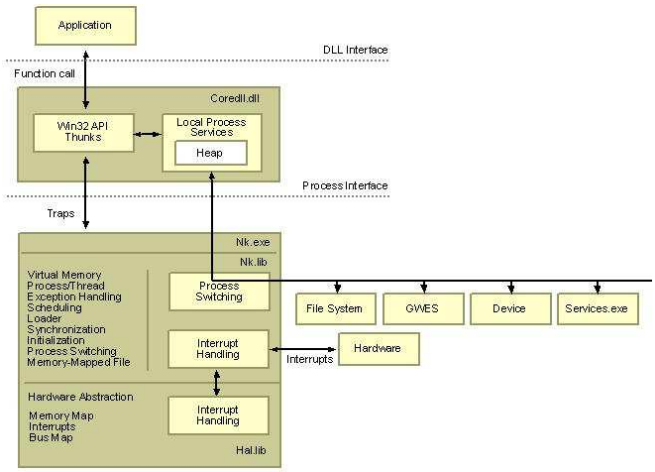

Figure 6.2: Windows CE .NET Kernel

the local heap provided for all Windows CE applications or create separate heaps.The kernel also allocates memory to the stack for each new process or thread.

One can use the kernel memory functions to allocate and de-allocate virtual memory, use memory on the local heap, create separate heaps, and allocate memory from the stack. Your code can use the unused memory from the static data block that is allocated to load the application. Processes also can use memory-mapped objects to share data.

6.5 Communication Services and Networking

Microsoft Windows CE provides networking and communications capabilities that enable devices to connect and communicate securely with other devices and people over both wireless and wired networks. The following list shows the tasks that Windows CEnetworking drivers, protocols, and application programming interfaces (APIs) enable:

- OEMs can create network-connected devices such as personal digital assistants (PDAs), smart phones, digital cameras, and gateways.
- Independent software vendors (ISVs), carriers, and enterprises can create rich network-aware applications and services usinAPIs and services such as Extensible Markup Language (XML), SOAP, Winsock, Message Queuing (MSMQ), and MediaSense.
- Independent hardware vendors (IHVs) can create networking drivers using Network Driver Interface Specification (NDIS) 5.1 and NDIS test tools for example 802.11, Bluetooth, GPRS, CDMA, and Ethernet.

Windows CE .NET also includes wireless features such as Bluetooth, 802.11 (802.1x, Extensible Authentication Protocol and 802.11 automatic configuration), and MediaSense; server features such as Remote Access Service (RAS)/ Point-to-Point Tunneling Protocol (PPTP) and File Transfer Protocol (FTP) Servers, and services and APIs such as Communications Service,

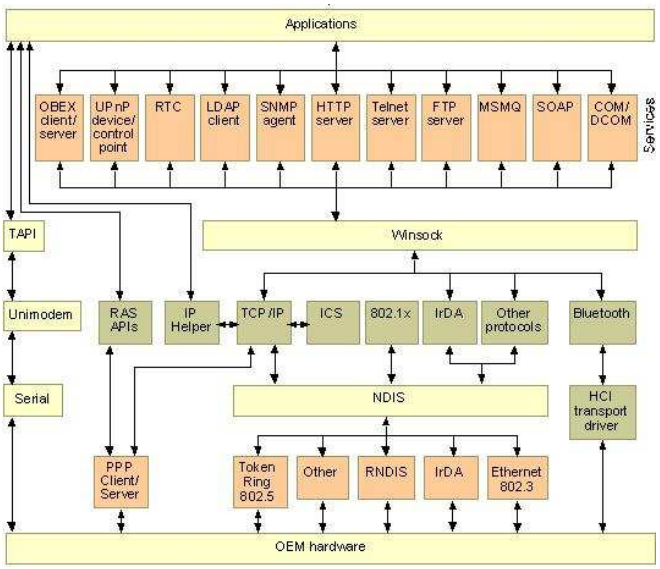

Figure 6.3: Windows CE .NET Network Architecture

Winsock 2.2, and Object Exchange Protocol (OBEX). In addition, Windows CE .NET includes an updated TCP/IP stack and Network Driver Interface Specification 5.1. The network user interface has also been updated to enable certificate logon, discovery of devices over Bluetooth, and setup of 802.11 networks.

6.6 TCP/IP

This topic describes how the Microsoft Windows CE .NET operating system (OS) implements Transmission Control Protocol/Internet Protocol (TCP/IP). Microsoft TCP/IP protocol suite is examined from the bottom up.

The Microsoft Windows CE OS includes a standards-based TCP/IP stack, allowing Windows CEbased devices to participate as peers and servers on local area networks (LANs) and remote networks.

The TCP/IP suite for Windows CE was designed to ease integration of Microsoft systems into the embedded market, telecommunications, the enterprise, the consumer market, government, and public networks. It also provides the ability to operate over those networks in a secure manner. Windows CE is an Internet-ready operating system.

Windows CE also has the following performance improvements:

- Protocol stack tuning, including increased default window sizes.
- TCP Scalable Window sizes (RFC 1323 support).
- Selective Acknowledgments (SACK).
- TCP Fast Retransmit

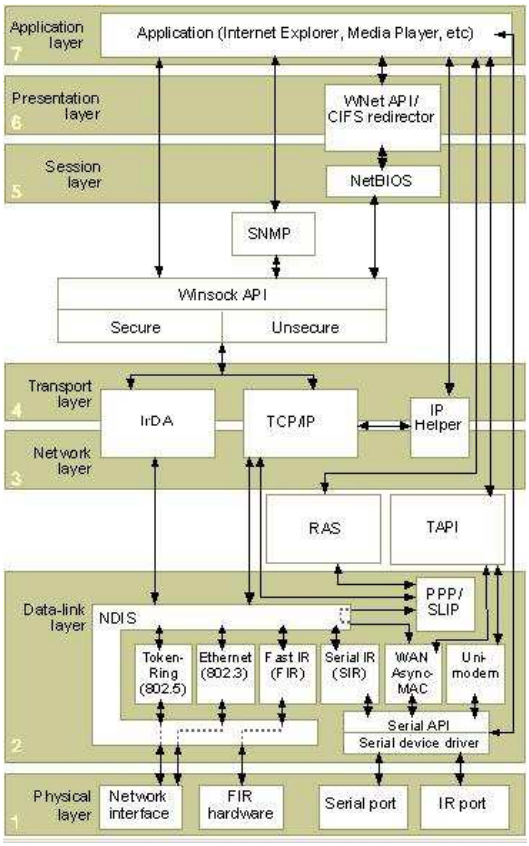

Figure 6.4: TCP/IP Architecture

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 CRESQ written

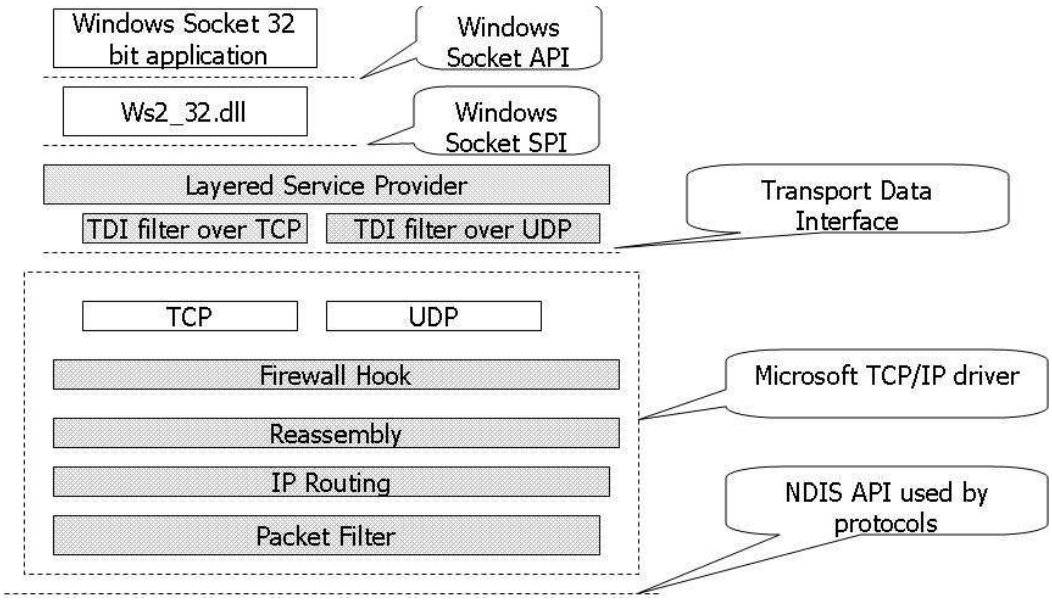

Figure 6.5: Packet Filtering Options of the WinCE Protocol Stack (upper half)

as a device driver, as the CRESQ routing protocol needs to send UDP control packets to other hosts.

6.7 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 5.1.

Figure 6.5 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 [24], 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.

6.7.1 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,

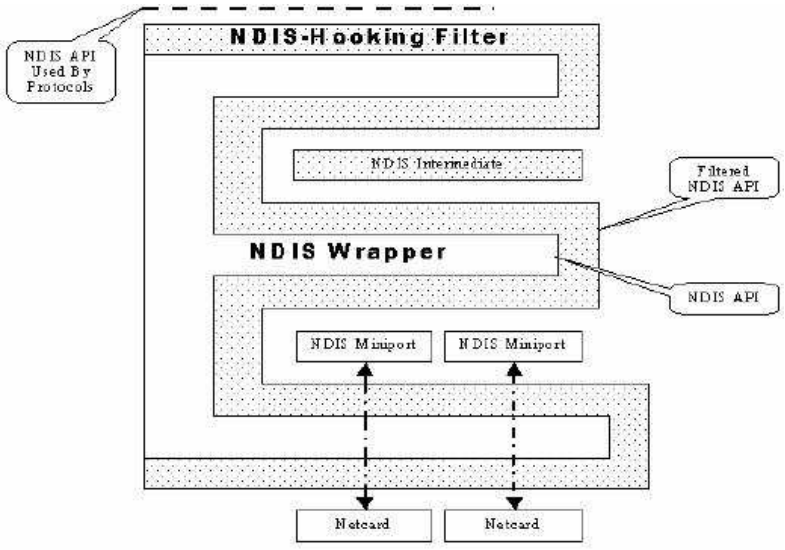

Figure 6.6: Packet Filtering Options of the WinCE Protocol Stack (lower half)

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 5.1.

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 condition 4 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.

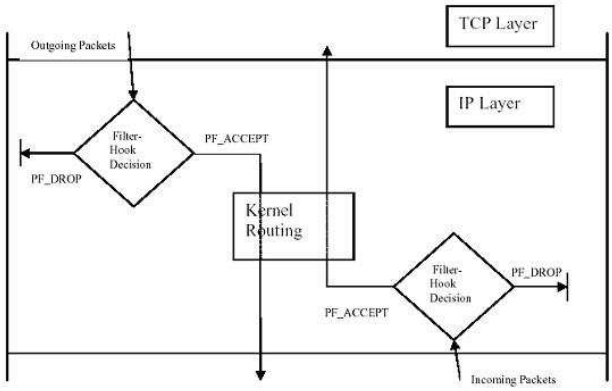

Figure 6.7: Packet Filter Hook Achitecture

6.7.2 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.

6.7.3 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 callback 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 CRESQ. As such, an implementation of an on-demand routing protocol as a filter hook driver could meet requirements 1, 3 and 4 of section 5.1.

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.

6.7.4 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' . 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 5.1.

6.7.5 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.

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 6.8. 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.

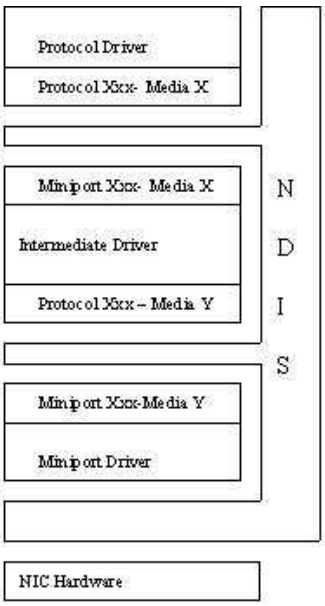

Figure 6.8: Layered NDIS driver architecture

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 5.1 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 requirement 4.

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.

6.7.6 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.

Chapter 7

User-level implementation of CRESQ

This section contains the core of the report giving details about the user level implementation of the CRESQ protocol as an independent application for the Windows CE emulator. This user level implementation is developed for testing the algorithm on actual mobile devices on actual operating system. This implementions gives us a fair idea of the use and performance of the algorithm when it will be implemented at the kernel level. This implementation provides us an opportunity where we can perform simulations of the algorithm on a test bed set up. Such simulations provide more realistic results than the ns simulations. This implementation would be followed by the more rigourous phase II implementation, in which we would integrate our protocol with Windows CE, so that our protocol can run directly above the data link layer and applications can be built overour protocol. The details of the phase II implementation are given in Chapter 8.

7.1 Data Structures

We now explain in detail the data structures made by us to implement the CRESQ algorithm. These data structures cover the entites that are exchanged during the protocol and that are used by the nodes for management purposes. The structures used by us can be basically classified into the following groups:

- **Lists of nodes.** Each node in the network has one or more lists of nodes with itself. Slaves and bridges have lists of their masters. Masters, on the other hand, have lists of their slaves. The *Addr_List* sent inside a ROUTE REPLY packet is also a list of nodes. Such an address list is also transmitted in packets required for route maintenance. Since our protocol needs to run as a system process, we have essentially made the implemenation in a manner such that all the memory needed is allocated statically. Thus the size of these lists are fixed using macros. Each item in such lists contains of an address field and a "valid" field. Since masters and slaves expire because of not being heard it is important to have the valid item to keep track of the items that have expired. No extra variables keep the size of the filled portion of the lists because after the algorithm has run for sometime, the entries in these lists will be somewhat scattered. Timers are also attached with every entity that make the entity to expire when it has not been heard for a certain amount of time. See Section 7.2 for more details on timers.

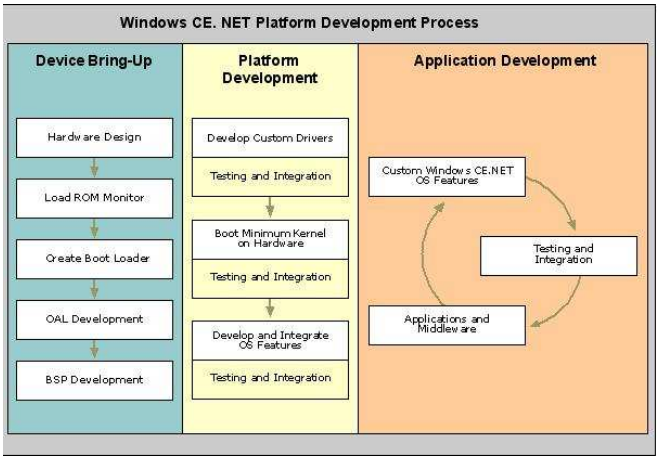

Figure 7.1: Application Development in Windows CE .NET

- **QoS and links.** Each node stores the QoS that it can provide for data transmission. Also QoS parameters need to be sent inside the ROUTE REQUEST packets. Nodes also need to store what links they currently have, i.e., who are their current neighbours. At persent, only bandwidth over several links is seen as a QoS parameter which will be changed after we identify more parameters from the results of our simulations. We just need to add more fields to the structures of QoS when we do so.

- **Packets.** There are different types of packets that can be transferred during a successful operation of the algorithm. The various packet types are HELLO, ROUTE REQUEST, ROUTE REPLY, ROUTE ERROR, FAIL, FAIL REPLY and DATA packets. All these packets have different properties and hence need to be represented uniquely. However, every packet is also a single transferrable entity and thus has to be encapsulated into a single structure for the transferring process. Hence finally a union of all these different types of packets is taken.

- **Caches.** The routes that are established by the protocol need to be cached so that a later reply for the same destination can be made easily. We need structures for caching such that these routes are easliy retrieved and that they also expire after a certain amount of time if no transmission has taken place on them. We also need to store the back pointers during a route discovery process. These back pointers store the details of the route requests that have been forwarded to other nodes, so that route replies can be sent to them.

7.2 Timers

Nearly every entity attached with a protocol for ad hoc networks has an expiration time. This is due to the mobility experienced by such networks. Nodes may move at random places at random intervals of time. This leads to breaking of existing clusters, formation of new ones, breakage of existing transmission routes, formation of new ones, new QoS specifications formed and others. Such changes are impossible to predict accurately because of the random nature

of ad hoc networks. Hence the expiration of entities are associated with the duration of time since that entity was last known to be present at the expected place. If this duration crosses a threshhold, then we can assume the entity to have expired.

Such an expiration process is essentially carried out by the use of timers. Timers associated with different entites are initialised every time the entity is heard. The timer expires if it is not re-intialised and the threshhold discussed above is crossed. This event is called the expiration of the timer and can be captured. Control can then be transferred to a method that produces the neccessary operations depending on which timer has expired and it was attached with which entity. The threshold values are heuristic only and their optimum values can be found out by performing extensive simulations of our implementation and noting the performance of the network each time.

The following are the timers used by us:

- **Hello Trigger Timer.** This timer is used to trigger hello events at the nodes. Its value serves as a basic unit for other timers, thus, the expiration values of the other timers are in a way multiples of the expiration value here.
- **Slave Timer.** This timer is used to remove a slave from the *Slave_List* of a master if the master has not heard from the slave recently.
- **Master Timer.** This timer is used to remove a master from the *Master_List* of a slave/bridge if the slave/bridge has not heard from the master recently. In case the timer expires on a slave node, the node becomes none.
- **Cached Route Timer.** Cached routes expire on the expiration of this timer whih happens when no data transfer has taken place on them recently.
- **Back Pointer Timer.** Back pointers used to store the properties of a route request to prevent multiple propagation of the same request and to send route replies to the nodes on the previous hop of the request, expire during the expiration of this timer. This may happen if a route for the destination of the request does not exist.

7.3 The protocol

The various algorithms that are part of the protocol are coded in the C programming language in the interface provided by the Windows CE 4.2 emulation kit. These algorithms are the Cluster Management algorithm, the route discovery and the route establishment algorithms. No initial clustering algorithm is used as our aim is to test the CRESQ algorithm which essentially covers the cluster management part. For the purpose of results, we can start recording only after some initial clustering is done by the cluster management algorithm itself.

The data transfer part is done by using the Winsock Application Provider Interface. This is only a Windows interface for the conventional socket programming. We use sockets on the UDP layer. Preference was given to the UDP layer over TCP as our protocol manages all the routing for which it only needs to send individual messages which are not related to each other. However, for the kernel level implementation we will assume a reliable data link layer below our protocol.

7.4 Simulations

We will test the user level application developed by us by using the facility of Emulator. The Emulator is a tool that mimics the behaviour of hardware that supports a Microsoft Windows CE based platform. With the Emulator, you can design and build aWindows CE based platform and test it using software that mimics hardware rather than testing the platform on hardware.The Emulator also allows you to provide anapplication developer with a virtual hardware platform that the developer can use to test applications for your platform.

In the absence of PDAs, we can run the application on normal PCs of our LAN by running the emulators on them.The emulator can run our application in the same manner a normal PC will do , only with slight degradation in performance. On an average an emulator gives 80% of performance of the normal PC on which it runs. So running the emulator on various PCs , we can easily create clusters of nodes (emulators running on PCs).

Now we will refer to the emulators running on various PCs as nodes.Snce currently we have implemented the protocol over TCP/IP suite, so we have toface some problems in cluster management.Since all the nodes are connected via LAN , so if we simply run the algorithm , all the nodes will form a single cluster because all of them see the other nodes as just one hop away. Thus we can simply hardcode the neighbours of every PC for testing purposes. This step will not hamper the protocol simulation in any way , as the CRESQ protocol is detached from the cluster management and can be run with any types of clusters provided they satisfy the cluster property. So after hardcoding the neighbours in the appliaction for every node, we can run the software and test the performance of the protocol.

There are many parameters which have to be determined on the basis of simulations. Currently we have assigned arbitrary values to many parameters like:

- MAX_MASTERDEG 10 : Refers to Max. no. of Slaves/Bridges of a Master
- MAX_MASTERS 10 : Max. no. of Masters of a Bridge/Slave
- MAX_ROUTE_SIZE 10 : Max. size of a route
- MAX_LINKS 20 : Maximum no of links of every node
- MAX_BACK_POINTERS 30 : Maximum no of back pointers a node can store at a time
- MAX_CACHED_ROUTES 30 : Maximum no of routes that can be cached by a node

Also we need to determine the length of various timers associated with the protocol.Determining the optimum values of theses timers is crucial to the performance of the protocol. If the timers are too short, then there is a danger of valid routes being invalidated and accordingly if the timers are too long , then it may allow stale routes. The various timers for which optimum values are to be found are given below. If optimum values of these timers are not found, then the performance of the protocol will decrease as lower timer values may result in expiration of slaves/masters when they are still connected, and also more frequent hello trigerring resulting in large overheads. Higher than optimum timer values will result in stale routes and links at the nodes.

- HELLO_TRIGGER_TIMER

- SLAVE_TIMER_LENGTH
- MASTER_TIMER_LENGTH
- CACHED_ROUTE_TIMER_LENGTH
- BACK_POINTER_TIMER_LENGTH

So by running the application by varying certain parameters , we can determine the optimum values of all the above parameters. Also it will help us to ascertain any dependencies of these parameters on each other.On the basis of simulation results , we can suggest certain more QoS parameters , which are neccessary for protocol to give the optimum performance.

7.4.1 Application Debugging

The Windows CE OS kernel provides debugging support for applications, including printing debugging messages and registering debug zones. Debug zones allow you to debug by enabling macros that control the output of debugging messages.

Chapter 8

Windows CE CRESQ Implementation Design

This chapter presents the design for the Windows CE CRESQ implementation. Section 8.1 presents some design approaches, discussing the advantages and disadvantages of each. Section 8.2 describes the adopted design, on a module-by-module basis.

8.1 Design Approaches

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

8.1.1 Embedding CRESQ 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 CRESQ 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.

8.1.2 Implementing CRESQ as an Intermediate Driver

Section 6.7.5 introduced the NDIS intermediate driver, and described how the CRESQ routing protocol could be implemented using this mechanism.

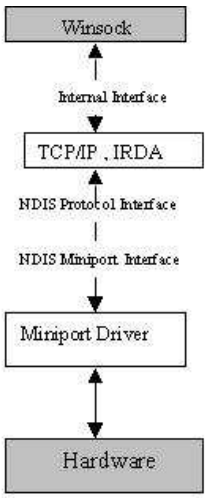

Figure 8.1: Modifying the Filter Hook Mechanism

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 CRESQ 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).

8.1.3 Modifying the Filter Hook Mechanism

As described in section 6.7.3, 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 CRESQ 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 CRESQ driver on initialization 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 reinsert 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.

8.1.4 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 5.1.

Kawadia, et al. [23] 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).

8.2 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 6.7.5 and 8.1.2. 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 8.1.3.

8.2.1 Module Description

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

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

cresq_driver{.c, .h}

This driver has the dual purpose of initializing the NDIS intermediate driver (or filter-hook driver), and the CRESQ 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 initializes the CRESQ 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.

cresq_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 CRESQ 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_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 initialize the queue, insert entries, remove the next entry, and cleanup the queue. Event queue entries consist of CRESQ control packets and cleanup events, and are used such that the bulk of the CRESQ routing protocol processing occurs in the CRESQ thread, and not in an interrupt thread.

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 CRESQ 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 CRESQ packets (those UDP packets destined for the CRESQ port) are examined. Firstly the format of the packet is checked to see that it is a properly formatted CRESQ 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 CRESQ 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 CRESQ 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 CRESQ 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_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 CRESQ 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.

8.2.2 Completed Work

We have started working on the kernel driver code. We have completed the coding of modules cresq.h, cersq_driver{.h, .c}, event_queue{.h, .c}, miniport{.h, .c}, packet_in{.h, .c}, packet_out{.h, .c}, protocol{.h, .c} and neighbour_list{.h, .c} and are currently working on modules cresq_thread{.h, .c} and timer_queue{.h, .c}.

Chapter 9

CRESQ Implementation Evaluation

Here we will discuss the testing of implementation of CRESQ at the user level.

We tested the algorithm on emulators provided by Platform Builder. There were three levels of testing. Firstly we tested algorithm on two emulators which exchange hello messages with each other and form a cluster, with one becoming the master and other the slave. This setup comprised of two computers running Platform Builder provided with Windows CE.NET 4.2. The Platform Builder is used to build our CRESQ application. It is then used to generate a plaform, which is a specific implementation of Windowcs CE.NET based on an "Enterprise Web Pad" and using "Emulator: X86 BSP (Broad Support Package)".

We downloaded the OS image into the emulator running in "Virtual Switch - Fixed IP" mode. It is imperative that the two PCs running the Emulator have installed Microsoft Loopback Adapter, without which the emulators fail to run in Vitual Switch Fixed IP mode. When the emulators came up, we set their IP addresses & the subnet masks corresponding to the LAN and ran the CRESQ application from the Platform Builder by selecting it from the drop down menu of the "Run Programs" tab in the Platform Builder. Our application shows messages in pop-up windows during the occurence of specific events. This helps us in debugging as well as it shows the progress of the protrocol. We easily ascertained that both the nodes were sending Hello Messages to each other and the node which first exhausted its NumHelloTries became a Master and sent a "Be My Slave" Hello packet to the other node from whom it had received a Hello Packet. On receiving the "Be My Slave" packet, the other node changed its state from None to Slave and made appropriate changes to its master list.

In another implementation testing, we tested our application on a CEPC running Windows CE.NET. We used three computers for this test (Pentium IV , 256 MB RAM). We ran the emulator on one of the PCs and the other two PCs were used for CEPC. We ran the Platform Builder on one of them and then booted the third PC using X86 boot disk floppy created using Boot disk utility. The IP address for the CE was specified in the boot disk. The OS image is transferred from the development workstation to the CEPC through ethernet. We now ran the application from the Platform Builder. Following the above procedure we were able to run our application between an emulator and a CEPC.

In the third type of implementation testing , we ran the application simultaneously on three emulators and let them form a cluster using Hello Messages. When the cluster was complete and each node had assumed a state of Slave/Master/Bridge, we started an application that would send a specific data from one slave to the other. We start the application on the three

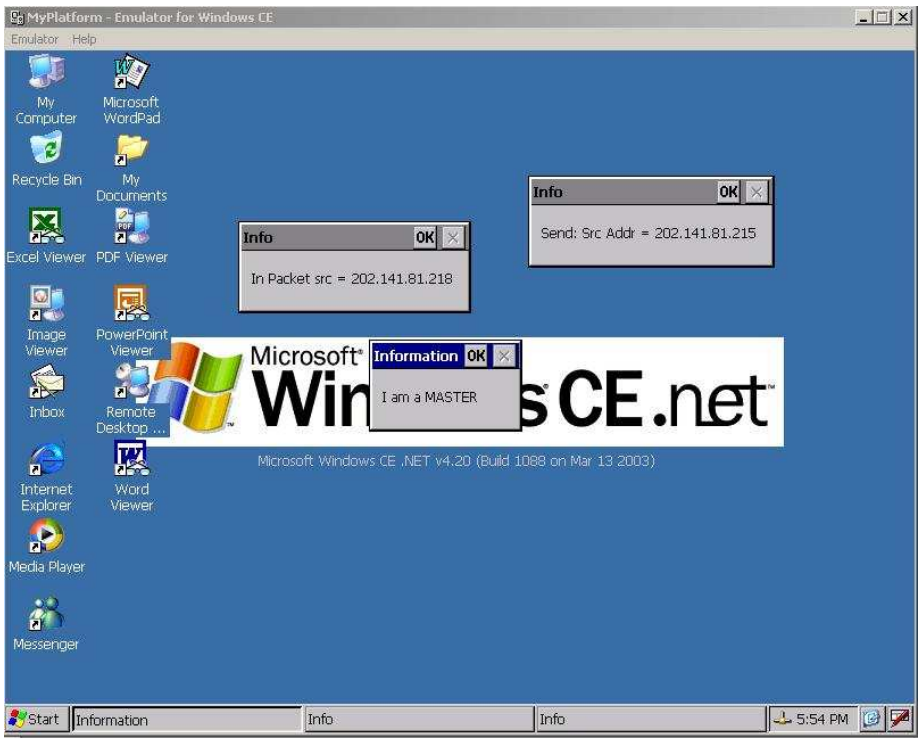

Figure 9.1: A node becomes a master

emulators at different times so that only the preferred node becomes the master and the other two become slaves. The would-be slaves were programmed such that they could not contact each other directly. Since a route to the other slave didn't exist, the former had to initiate a ROUTE REQUEST packet and send it to the master. The master gave the route to the other slave through itself and after receiving the route reply, the source slave was able to send the data packet. The other slave received it intact, via the master. Thus the routing and controlling aspects of the protocol were tested in a minimal way.

We also checked the working of our application in the scenario when one of the nodes accidentally goes off and then comes again. We reset the emulator at one of the slave terminals. When it came back again and we started our application on it, the node started in the NONE state and on communication of three hello messages it became a SLAVE again. The three hello messages were (i) A general hello sent by the master, (ii) "Can I be your Slave" hello from the node, and (iii) "Be my Slave" hello from the master. We also show one screen shots of the successful runs of our application here. More screenshots are provided in Appendix A.

Chapter 10

Conclusion

The contribution of this work has been to produce a real-world implementation of CRESQ 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 run at the application level in windows CE.NET.

A study of the possible approaches to implementing the CRESQ routing protocol, given the poor support in current operating systems for ad-hoc routing protocols, has been presented. 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 CRESQ will be performed.

The NDIS intermediate driver approach is easy to install and distribute. It does not require any changes to proprietary code. However there are 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).

Our work has also shown the sucessful implementation of our protocol on CEPC devices. This work should prove helpful for the WinCE developers who work in the area of network protocol deployment.

Chapter 11

Future Work

We have developed the NDIS driver for Windows CE.NET and we reuire to test the driver on laptops/PDAs with ad-hoc support. The driver needs to be integrated with the system and then used for data communication in an ad-hoc network. Thus, robust testing of the driver should be done based on test-bed based set ups.

After the robust testing process we should move to the optimisation of the various parameters that govern the execution of our protocol. These include the various timers and sizes of tables involved. The optimisation will depend on the performance charactestics of the protocol when run run in test-bed based set ups. The performance parameters may be the route discovery time, the average latency, and the percentage of data packets out of the total packets transferred.

Then the other objective of the whole process should be looked into, that is, the various QoS parameters that we can include and the logistics of handling them in a real-world network. We realise that this alone can be a cumbersome process given the number of QoS parameters that can be identified and the unique ways of handling each one of them.

As described in section 8.1.4, 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.

Appendix A

Screenshots

The following pages show some screenshots of the successfull run of the application level implementation of CRESQ. Gien below is a small reference list for the figures that follow.

- Figure A.1 shows a pop up window on the master node when it has received a hello message from a node whose state was NONE. Thus the master is trying to send a "BE MY SLAVE" hello message to that node. Here the IP address of that NONE node is 202.141.81.215.
- Figure A.2 shows the picture from our third kind of evaluation run. When the slave node recognises that it has to send a message to a node for which it has no route (here that node is 202.141.81.181), it initiates a route request process and thus is trying to send a RREQ packet to its master 202.141.81.218.
- Figure A.3 shows the master sending a route reply. The master, on receiving the route request finds out that 202.141.81.181 is its slave and there is no direct contact between the source and the destination. It now constructs a route reply for the source.
- Figure A.4 shows the slave receiving the route reply.
- Figure A.5 is the picture from the source slave side again. It shows that the reply has the address of the destination as its zeroth item. Note that in our implementation the route address list is in reverse direction, *i.e*, the next hop is the last valid item in the list, thus the destination address will always be the zeroth item.
- Figure A.6 shows the master sending a data (here 78) to the destination. It acts as an intermediate node and this is the instance when it has received the data packet from the source and it has now to be sent to the destination.
- Figure A.7 shows the destination receiving the data. It clearly says that the source of the data was 202.141.81.215 and the immediate previous hop (shown by "via") is the master (202.141.81.218).
- Figure A.8 shows the Platform Builder on the development workstation when an OS image is being transferred to a CEPC.

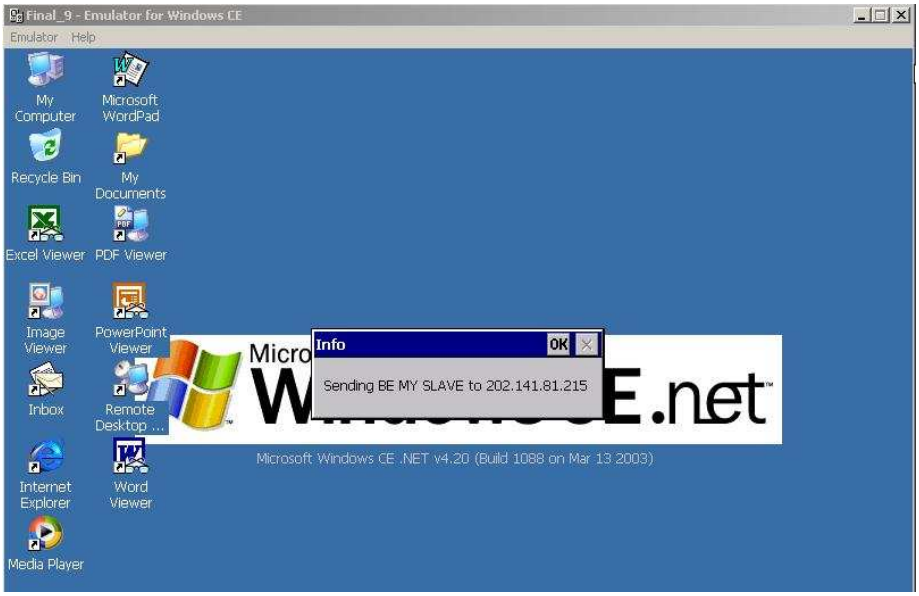

Figure A.1: The node asks a neighbour to become a slave

Figure A.2: Sending route request to the master

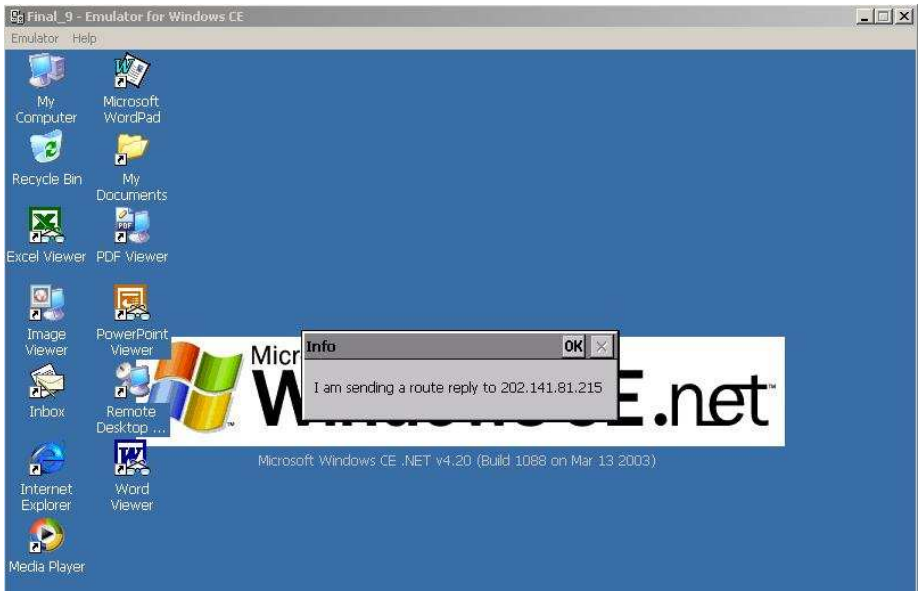

Figure A.3: Master sends route reply

Figure A.4: Reply is received from the master

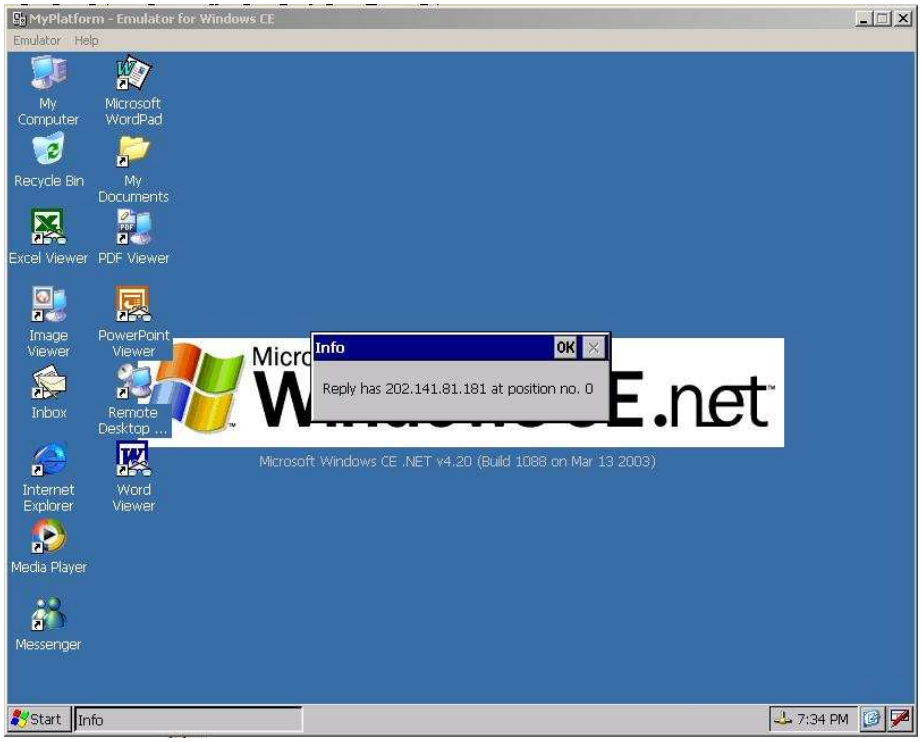

Figure A.5: Showing an entry in the address list of route reply

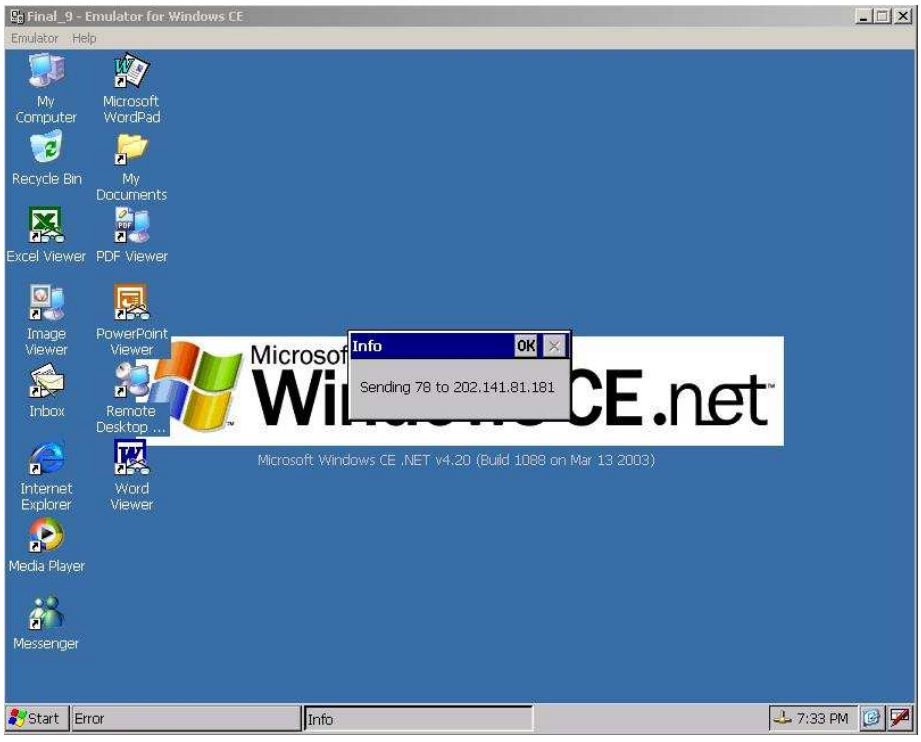

Figure A.6: The intermediate node sends data to the destination

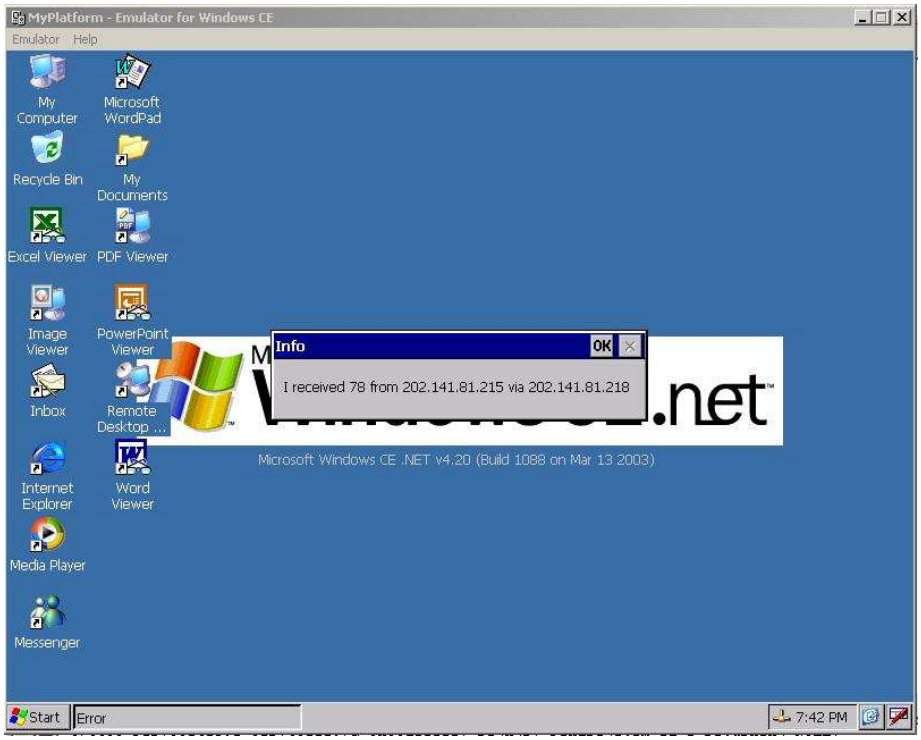

Figure A.7: The destination finally receives the data via the master

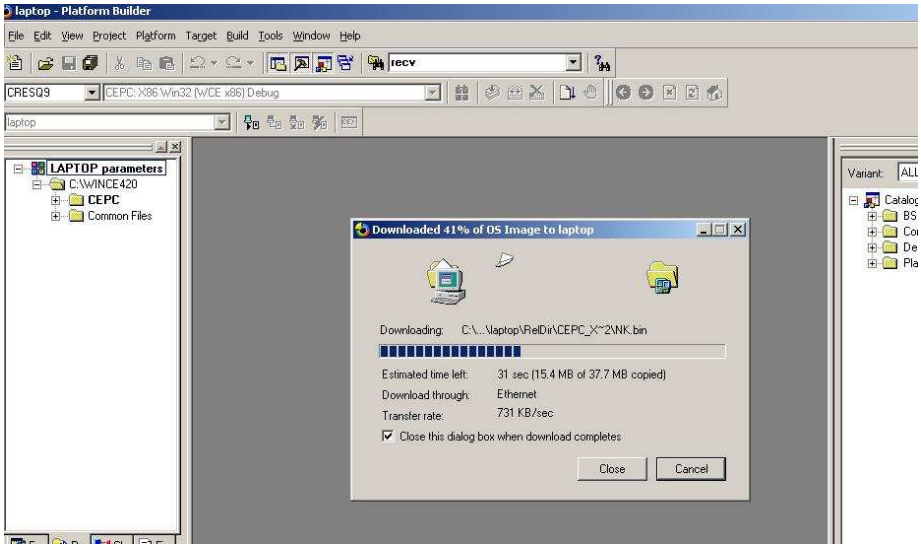

Figure A.8: The operating system getting downloaded on a standalone CEPC

Bibliography

[1] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers. In ACM SIGCOMM'94, London, UK, September 1994.

[2] Punnet Sethi, CRESQ: Providing QoS and Security in Ad hoc Networks, Design of QoS Framework for Ad hoc Networks, BTP Report, IIT Guwahati, April, 2002.

[3] Chakraborty I., Barua G., "ACRPN: A New Routing Protocol for Adhoc Wireless Networks", *Indian Institute of Technology Guwahati, 2001.*

[4] Puneet Sethi, G. Barua, CRESQ: Providing QoS and Security in Ad hoc Networks, Proceedings of IEEE EuroMicro PDP 2003, Italy, February, 2003. (enclosed as Appendix II of this report).

[5] Puneet Sethi, G. Barua, Dynamic Cluster Management in Ad hoc Networks, Proceedings of the Conference on High Performance Computing (HPC 2002), Bangalore, December 2002.

[6] Microsoft Developer Network documentation for Windows® CE .NET.

[7] Extending The Microsoft PassThru NDIS Intermediate Driver –Part 2. Two IP Address Blocking NDIS IM Drivers James Antognini and Thomas F. Divine. http://www.wd-3.com/121503/ExtendingPassThru2.htm

[8] Intermediate Driver Packet Management. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/network/hh/network/301int_955z.asp.

[9] AODV implementation for windows, Master's thesis, David West, Trinity College, Dublin.

[10] The Windows Driver Developer's Digest Archives. http://www.wd-3.com/wd3Archives.htm

[11] Active PassThru NDIS IM Sample Notes. http://www.pcausa.com/pcasim/activepassthru.htm

[12] NDIS Intermediate Driver Samples For Windows NT, Windows 2000 and Higher. http://www.pcausa.com/pcasim/Default.htm

[13] Universal NDIS-Hooking Driver Samples (All Windows Platforms). http://www.pcausa.com/ndispim/NdisHookDefault.htm

[14] Writing an NDIS Intermediate Driver. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/network/hh/network/301int_3mg7.asp

[15] How to make an IM driver that works with NDISWAN over TCP/IP. http://www.pcausa.com/resources/im_ras2.htm

[16] Windows NT 4.0 DDK Samples. http://support.microsoft.com/default.aspx?scid=/support/DDK/Ntddk/NTSamples/default.asp

[17] NDIS Intermediate (IM) Driver Frequently Asked Questions. http://www.pcausa.com/resources/ndisimfaq.htm

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

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

[20] UU AODV homepage. Erik Nordstrm. http://user.it.uu.se/ henrikl/aodv/. September 2003.

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

[22] 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.

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

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