piconet

a wireless ad hoc network for mobile handheld devices

by

Alec Siu

School of Computer Science and Electrical Engineering,

University of Queensland.

Submitted for the degree of

Bachelor of Engineering (Honours)

in the division of Electrical and Electronic Engineering

October 2000

24 Barossa Street
Kippa-Ring
QLD 4021
Tel. (07) 3880 1093

October 20, 2000

The Dean
School of Engineering
The University of Queensland
St Lucia, QLD 4072

Dear Professor Lister

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Electrical and Electronic Engineering, I present the following thesis entitled “Piconet – a Wireless Ad Hoc Network for Mobile Handheld Devices”. This work was performed in partnership with Mr Israel Keys under the supervision of Dr Mark Schulz.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Alec Siu.

Abstract

This thesis documents the design and implementation of a wireless ad hoc network for mobile handheld devices, called Piconet. This product allows the deployment of a lightweight Personal Area Network (PAN) without relying upon the pre-existing infrastructure required by traditional wired and wireless networks such as the Internet. A collection of wireless mobile hosts (in this case, Palm Personal Digital Assistants) forms a temporary network in which each node also functions as a router. Piconet uses source-initiated, on-demand routing algorithms to transparently discover and maintain valid routes for the purpose of packet forwarding even in the face of dynamic network reconfigurations as users enter and leave the network coverage area.

Piconet provides a low-cost, lightweight alternative to wireless PAN and LAN solutions such as Bluetooth and IEEE 802.11, and is suited for low-bitrate data transfers such as messaging. Piconet demonstrates the feasibility of the ad hoc networking concept, and is a promising platform upon which further work in this area can be based.

Acknowledgements

This thesis was the product of many hours of work, frustrating and stimulating in equal measure. It could not have been produced without the help of many people; the author therefore wishes to thank:

Israel Keys, my thesis partner, for his dedication, endless good cheer and our many thought-provoking coffee sessions;

Dr Mark Schulz, for his guidance and insight;

Simon and Yvonne Siu, who tolerate my frequent comings and goings, and for being wonderful parents who have supported me every step of the way in my educational career;

Michael Siu, for being the best brother I could ask for, and whose advice and encouragement has always been of great value;

Rebecca McFadden, Martin Robinson and Nathan Roche, for making 2000 a wonderfully enriching experience that I will never forget;

Juergen Marchard, for teaching me the meaning of true happiness;

Simon Gee, for his sense of humour and proofreading this thesis;

Eng Wei Koo, whose friendship for the last four years made Engineering more than just bearable;

and last but not least, **Nestle**, for making the instant coffee that sustained me many a late night working on this thesis.

Table of Contents

List of Figures

List of Tables

1 Introduction

This thesis details the specification, hardware and software design, and implementation of a Personal Area Network (PAN) for mobile handheld devices.

This chapter introduces the thesis problem, identifies a general solution to that problem which forms the basis of the design outlined in this document, and concludes with a description of the structure of the remainder of the report.

1.1 The Problem

Wireless networking is unquestionably a hot topic within the computing world, and with good reason. Whether it is voice applications, web browsing, e-mail or simply the exchange of information, the ability to communicate without the encumbrance of a cable is a compelling prospect indeed. Unfortunately, the architecture that has driven phenomenal growth in networking – the Internet – is unsuitable for lightweight, wireless personal area networks. The Internet architecture is centralised and predicated on the existence of a wired network of fixed switches and routers. Additionally, its protocols do not account for the additional design constraints of wireless mobile hosts, such as limited connectivity, bandwidth and battery life.

This thesis concentrates on the problem of providing wireless networking services to mobile devices. The target platform is the Handspring Visor Personal Digital Assistant (PDA) [1], one of a growing number of handheld information appliances in the marketplace based on the popular Palm Operating System (Palm OS) [2].

1.2 The Solution

The essence of the lightweight PAN described above is its ability to be deployed without relying upon pre-existing infrastructure. Mobile ad hoc networks (MANETs) [3] meet these requirements perfectly – a collection of wireless mobile hosts forms a temporary network without the aid of centralised administrative or support services. Specialised routing protocols ensure that node roaming does not affect the network's ability to forward packets to the right destination.

This thesis describes the design and development of Piconet, a peer-to-peer, packet-switched, wireless ad hoc personal area networking system that enables lightweight applications such as messaging and browsing.

1.3 Report Structure

This thesis details the design and implementation of a system that addresses the problem of wireless ad hoc networking as outlined in Section 1.1 and whose general solution was discussed in Section 1.2.

Chapter 2 discusses in greater depth the problems associated with ad hoc networking, and the proposed solutions to these problems. Both the current state of academic research in this area and commercial products using ad hoc networking techniques are surveyed.

Chapter 3 specifies requirements and features of an ad hoc networking system for mobile devices. The system is broken down into components, with each component specified in terms of their hardware and software requirements.

Based on these specifications, Chapters 4 and 5 describe the implementation of the major components of the Piconet system, the wireless hardware and protocol stack software, respectively.

Chapter 6 discusses the performance of the Piconet system, and compares the achieved functionality with that derived in Chapter 3.

Chapter 7 presents a comprehensive design review and suggestions for system improvements and future work, with Chapter 8 concluding the thesis with a brief summary.

2 Existing Solutions

This chapter discusses the problems associated with mobile ad hoc networks. These problems are yet to be fully and satisfactorily solved, and hence the current state of research in this area is reviewed; this research supplies the theoretical underpinnings of the protocol design in the Piconet system (the "Picostack").

A number of existing commercial products allow for the creation of ad hoc networks of mobile devices; their functionality is also examined and compared against the Piconet system.

2.1 Ad Hoc Networks

Mobile ad hoc networks (MANETs) are a continuing research area in the computing community [4]. Some basic theory regarding computer networks is reviewed to place into context the challenges that any ad hoc network design must address.

2.1.1 Network Theory

A network can be defined as two or more computers connected by some medium. This medium or *link* (which need not be physically tangible) supports the transfer of bits of data between these computers (called *nodes*) [5].

This greatly simplified characterisation masks the numerous complexities that lie behind the computer networks of today. What services will the network provide to the applications programmer to send and receive data in a timely and reliable manner? How is the data routed to its final destination? And how do we transmit those bits over the medium without error?

The answers to these questions are governed by such things as the physical topology of the network, how the medium is accessed, the physical characteristics of the medium itself (particularly in a wireless context), the envisaged applications of the network, as well as its geographical extent, be it a small office network or something on a global scale such as the Internet.

At this point, we introduce the layering abstraction of network systems. The lowest layer abstracts the services offered by the underlying hardware (for example, encoding and transmitting binary data across the channel); each successive layer adds

more services that are implemented in terms of services provided by layers beneath it in the stack. Each layer in the stack communicates with its peer. A peer is an implementation of the same protocols in the equivalent layer on a remote system. This style of data transfer means that a layer does not concern itself with how layers above and below it function, only on the services that those layers provide. This abstraction makes the understanding of the components that make up a network system easier. It allows a component's implementation to change so long as its functionality and interface remain consistent.

<table>
<tr><th>Layer No.</th><th>OSI model layer description</th><th>TCP/IP equivalent</th></tr>
<tr><td>7</td><td>Application</td><td rowspan="3">Process/Application</td></tr>
<tr><td>6</td><td>Presentation</td></tr>
<tr><td>5</td><td>Session</td></tr>
<tr><td>4</td><td>Transport</td><td rowspan="2">Host-to-Host</td></tr>
<tr><td>3</td><td>Network</td></tr>
<tr><td>2</td><td>Data Link</td><td>Internet</td></tr>
<tr><td>1</td><td>Physical</td><td rowspan="2">Network Access</td></tr>
</table>

Figure 1 The OSI reference model and Internet model side-by-side

The *Open Systems Interconnection* (OSI) reference model is commonly used in describing the structure and function of data communications protocols; it formally partitions network functionality into seven layers.

The *Physical Layer* defines the mechanical and electrical means for data transmission, i.e. cable, connector and signalling specifications.

The *Data Link Layer* deals with framing of data packets, error detection, correction and retransmission. It is itself broken into two sub-layers: *Logical Link Control* (LLC) in the upper half, and *Medium Access Control* (MAC) in the lower half. The MAC layer handles transmitting and receiving data on the channel, while LLC provides a standard way of interfacing with the Network Layer.

The *Network Layer* is responsible for routing packets from one node on the network to another, as well as flow control.

The *Transport Layer* establishes, maintains and terminates connections between two machines. A connection provides a transparent, logical data stream.

The *Session Layer* provides sequencing, synchronisation and full-duplex communications.

The *Presentation Layer* formats data for machine-independence, e.g. translating between ASCII and EBCDIC character sets.

The *Application Layer* is the end-user interface to the system; file access and transfer, terminal emulation, interprocess communication – these are all handled here.

As can be seen from Figure 1, the Internet or TCP/IP model has fewer layers, and collapses the functionality of several OSI layers into one based on real world experience that makes such functional separation illogical. The Internet model is the prevalent model in networking today. As shall be seen later, elements of both models are used in defining the Piconet system.

2.1.2 Challenges

From our understanding of the OSI model, the role of sending, receiving and forwarding packets can be isolated to the Network, Data Link and Physical layers. It is at these layers that a mobile ad hoc network differentiates itself from a conventional wired network, and also where the greatest challenges lie.

At the Network Layer, determining a path between two nodes is fraught with difficulties. The conventional method is to periodically exchange routing information between routers. For a fixed network, this is sufficient since the topology changes relatively infrequently. In an ad hoc network, however, node mobility can rapidly make routing information out of date. Increasing the frequency of routing information updates can solve this problem. However, as the network grows in size and the frequency increases, it can be shown that the entire network bandwidth will be consumed by routing updates [4]! The situation is particularly acute since the network uses a shared wireless medium. This limits the bandwidth available to exchange routing information. Because mobile nodes are usually battery-powered, this also means that reducing the number of transmissions is essential to prolonging battery life.

The consequences of a wireless broadcast channel are also imposed upon the Data Link Layer, which must handle the possibility of collisions as stations attempt to transmit simultaneously, as well as the greater likelihood of data corruption from noise, interference and other factors (as compared with a wired network).

These challenges, then, are areas of research within the computing community. Some of the proposed solutions to these challenges are presented below.

2.2 Current Research

Current research in mobile ad hoc networks concentrates on efficiently managing shared spectrum via Medium Access Control protocols, and handling node mobility using specialised routing protocols.

2.2.1 Medium Access Control

Medium Access Control (MAC) protocols at the Data Link Layer attempt to address the problem of a wireless medium. These include Aloha, Slotted Aloha, Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) and MACA [7]. These schemes use a variety of methods to improve throughput: time-division multiplexing; listening for other transmitting stations before attempting to transmit; signalling an intention to transmit; and the list goes on.

Many of these schemes are flawed; the hidden terminal problem, depicted in Figure 2, is a demonstration of why the collision avoidance as found in CSMA/CA (and used by IEEE 802.11 wireless LANs [8]) is insufficient to prevent collisions. Station B is silent; this leads Station A to believe it is safe to transmit. Station A cannot hear Station C, and hence a collision occurs between packets from A and C.

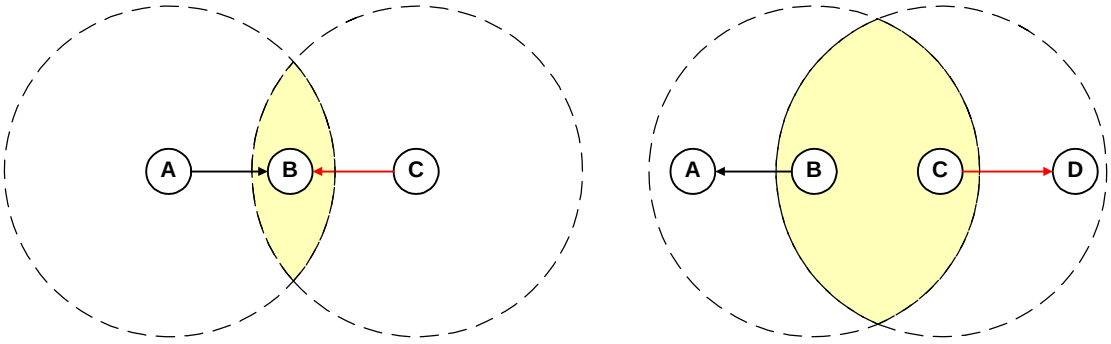


Figure 2 Illustration of the hidden terminal problem (left) and exposed terminal problem (right)

Figure 2 also gives an illustration of the exposed terminal problem, which is the opposite effect but just as undesirable. Here, station B will overhear station C's transmission, and defer sending its own data. This is unnecessary since the intended receiver, station A, is out of range of C and the packet is in no danger of collision.

In both cases, the fundamental flaw is that collision avoidance only takes into account the receiving capabilities of the source, and not the destination.

Schemes to overcome these problems, such as Multi-hop Access Collision Avoidance (MACA) and the Media Access Protocol for Wireless LANS (MACAW) [9], establish a *dialogue* between sender and receiver: before transmitting data, the sending station reserves the channel by transmitting a ready-to-send (RTS) signal; if the target is prepared to receive (i.e. it is not receiving data from another source), it sends back a clear-to-send (CTS) and stations which overhear such a signal inhibit their own communications to avoid collision.

While RTS/CTS dialogue schemes can achieve channel utilisation rates of up to 40% [4], they presume that all potentially interfering nodes can hear these RTS/CTS dialogues – a flawed assumption when dealing with a mobile network.

2.2.2 Routing

Routing is generally defined as the process of finding a path from a source to any destination in a network [10]. This is usually accomplished using routing protocols that maintain information in a routing table at each node indexed by destination, with each entry containing the address of the next hop in the path to that destination.

In a mobile ad hoc network, each node also acts as a router, forwarding packets as necessary. This overcomes a node's limited transmission radius, as well as allowing each node to roam provided it stays within range of at least one other node.

The routing protocol must adapt to this kind of network reconfiguration to ensure packets arrive at the destination when it is physically possible to do so, as well as consider the limitations of a mobile wireless device in terms of available bandwidth and power consumption.

Put simply, there are two opposing demands: to limit the amount (and size) of control messages to preserve bandwidth, while simultaneously maintaining up-to-date routing information to reflect possibly frequent and rapid network reconfigurations.

Routing protocols can be loosely divided into two categories: on-demand (or "reactive" or "source-initiated") protocols, and table-driven or ("proactive") protocols [11].

Examples of the latter include the Distance Vector and Link State families of protocols. There is a continuous evaluation of routes within the network such that when a packet needs to be forwarded, the next hop can be immediately found in the route table. The Internet's RIP and OSPF protocols fall into this category. The main disadvantage is the use of network capacity to keep routing information up to date.

On-demand protocols initiate a route determination procedure (by the source) only when a packet needs to be forwarded (i.e. "on demand"). The problem with reactive protocols is the delay in determining a route between source and destination, in addition to the network bandwidth used in the search procedure. Arguably, they present the best compromise so far between the constraints of limited bandwidth and rapid network changes. The Wireless Andrew Project at Carnegie Mellon University is proof of the effectiveness of on-demand routing, where it is used to allow students to roam the campus with a wireless connection to the University's network [12].

2.3 Commercial Products

A number of commercial products and standards exist for creating wireless local area networks, such as HiperLAN [13] and HomeRF [14]. Few, however, incorporate ad hoc techniques in the way that they connect devices together. The few that do are discussed below.

2.3.1 Bluetooth

Bluetooth is a technology that promises fast, secure, point-to-point wireless communications over short distances (approximately 10 metres) for devices as diverse as mobile phones, consumer electronics appliances and desktop computers [15], [16]. It uses spectrum in the unlicensed Industrial, Scientific and Medical (ISM) band of 2.4 to 2.48GHz. Spread spectrum, frequency-hopping techniques allow it to provide full-duplex data transfers at speeds of up to 720kbps. Bluetooth's emphasis is upon interoperability between products using it, as well as low-cost for system implementers.

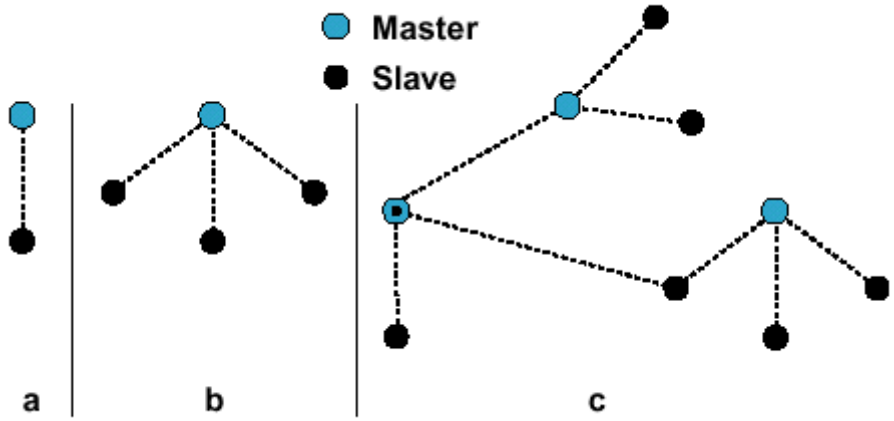


Figure 3 Bluetooth piconets: (a) single slave; (b) multi-slave; and (c) scatternet

Besides being a hardware standard, Bluetooth defines a protocol stack that allows for hierarchical ad hoc networking in the form of “piconets”, in which Bluetooth devices form themselves into point-to-multipoint picocells of seven slaves under the control of one master. Multiple piconets in overlapping coverage areas form “scatternets”. A slave can participate in multiple piconets on a time-division multiplexed basis, while a master in one Piconet can be a slave in another.

Bluetooth can be used as the link layer for a variety of other protocols, including TCP/IP and RFCOMM, as well as a general cable replacement device.

Bluetooth devices are scarce; Figure 4 shows a number of devices from manufacturers such as Ericsson, the original promulgator of the standard (along with Nokia, IBM and Toshiba), and TDK. These demonstrate typical application areas of Bluetooth – as a replacement for wired connections. The TDK device is an adaptor that allows a USB cable to be replaced by a Bluetooth link. The Ericsson products show how a hands-free headset can be connected to a mobile phone via Bluetooth.

Figure 4 A selection of Bluetooth-enabled products: (from left to right) the TDK USB adapter, the Ericsson R520, Ericsson T28 with Bluetooth module, and the Ericsson wireless headset

A small selection of announced (but not shipping) Bluetooth products are available for the Handspring Visor PDA. These include the Xircom Springport (expected to ship in early 2001) [17], and the BlueConnect from Widcomm [18].

While pledges of support for Bluetooth are universal in the computer industry, it is not expected that Bluetooth will become ubiquitous until at least 2003. The originally envisaged price point of $5/chipset in volume is also unlikely to be achieved in the short term.

2.3.2 IEEE 802.11

IEEE 802.11 and its close descendant IEEE 802.11b are wireless local area network communications standards that operate in the 2.4GHz band at data rates of up to 11Mbps and distances of 200 feet [8].

In an IEEE 802.11 network, there are two possible configurations for data transmission: the independent model, in which all nodes in the network must be within range of each other, and the infrastructure model, in which all inter-node communication must pass via a central Access Point (AP). The independent model allows for ad hoc collections of nodes, but communication is point-to-point (still using a shared broadcast channel), with no packet forwarding involved. The infrastructure model can extend the range of IEEE 802.11 networks by stringing together a series of APs, but management of traffic that spans multiple AP coverage areas is not dictated by the standard.

Examples include the Nortel BayStack Wireless LAN series of products, and the Apple AirPort and iBook combination [19]. Typically, IEEE 802.11-compliant

transceiver modules are built into the device (as in the Apple's case), or as an add-on card with a PCI[1] or PC Card interface.

2.4 The Need for Piconet

Piconet differentiates itself from such devices above as a lightweight solution to the need for wireless personal area networking. Table 1 shows that Bluetooth is not widely available, whilst IEEE 802.11 prices itself out of range of the casual user.

	Bluetooth	**IEEE 802.11**
Physical Layer	Frequency-hopped spread spectrum	Frequency-hopped spread spectrum, direct sequence spread spectrum
Ad hoc capabilities	Yes	Limited by wired infrastructure
Range	10m	~300m
Cost	not widely available	~AUS$350 for PC Card ~AUS$2000+ for Access Point

Table 1 Feature comparison of commercial ad hoc networking products

IEEE 802.11 is most useful as a wired LAN replacement or supplement, with access points spread throughout the office environment. This incurs an inconvenience, since it requires infrastructure that precludes impromptu network deployment – wherever a network is needed, an access point needs to accompany it. Without the access point, extending the range of the network by multi-hopping is not possible. Thus, IEEE 802.11's target application is in local area networks and notebook computers rather than a personal area network of small handheld devices whose data transfer needs are less demanding.

Bluetooth is the product most closely aligned with the Piconet philosophy. However, Bluetooth is yet to deliver on its promises, with a paucity of shipping devices and yet-to-be-realised economies of scale in terms of the cost of chipsets.

Unlike Bluetooth, Piconet was created specifically with ad hoc networking in mind. It supports low data rate links, but has ad hoc networking capabilities built into its core. Additionally, it seeks to be a viable solution now at a lower price point than Bluetooth.

[1] Peripheral Component Interconnect

3 System Specifications

This chapter is devoted to specifying the Piconet system. It begins with an overview of the functions the system should be expected to perform; this is used to specify the components that will be required to make up such a system.

3.1 Functional Overview

The Piconet system should provide networking services to mobile handheld devices and more specifically the Handspring Visor[2], a Palm OS-based Personal Digital Assistant (PDA). Two nodes in the Piconet system should be able to exchange messages over a wireless medium, with such an end-to-end data transfer being facilitated by point-to-point packet forwarding at intermediate nodes acting as routers. Piconet should adapt to node mobility, and respond to connectivity disruptions as nodes enter and leave the network.

A few assumptions are made here:

- Only a small number of nodes need to be networked (15 in the current implementation).
- Traffic patterns in the network are generally bursty rather than sustained.
- Movement speed of network nodes is relatively small; typically, such movements are characterised by users taking their Palm from room to room, or entering and exiting the wireless coverage area.

Figure 5 shows the relationship between the OSI model and the protocol stack of the Piconet system. The intermediate node B forwards packets between nodes A and D in the ad hoc network. Hops are via wireless links, with the routing protocol operating at the Network Layer, or Layer 3, of the OSI model.

---

[2] In the remainder of this thesis, the Handspring Visor PDA will be referred to as “the Palm” for reasons of simplicity and likely reader familiarity. When the difference between the two is non-trivial, this will be noted in the text.

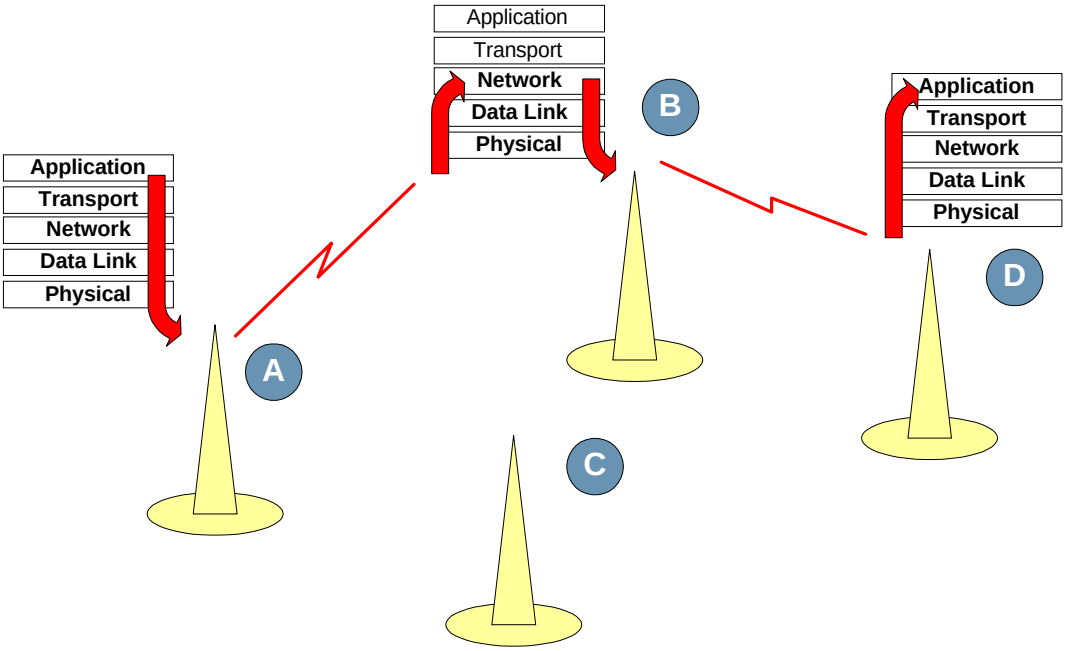


Figure 5 Relationship between the OSI model and Piconet

The high-level components that implement this functionality are shown in Figure 6; an external module provides the wireless link (necessary in this case since the Visor lacks built-in wireless transmission capability), while the software stack resident on the host device itself implements the protocols necessary for networked communication. The controller is shown as a separate logical block to the RF transceiver; it provides the interface to the Palm to make the wireless module appear to be a link for RS232-based serial transmission and reception (although the underlying communications medium is broadcast rather than point-to-point).

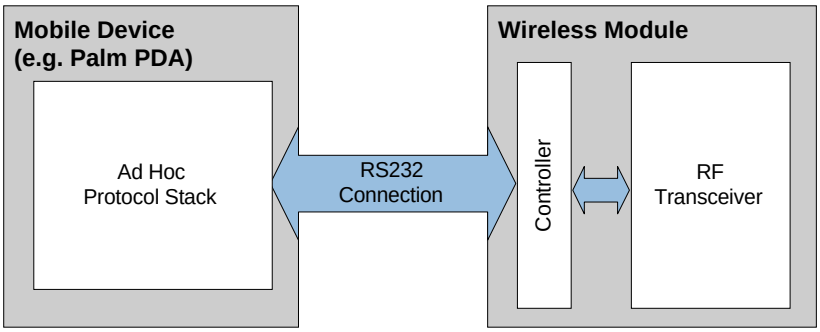


Figure 6 Block diagram of an individual node's system components

The following sections describe in more detail the requirements of the wireless module and the protocol stack.

3.2 Wireless Module Requirements

The wireless module attaches to the Palm PDA via its TTL-level RS232 interface. This not only ensures that the module operates across all devices on the Palm Computing Platform (Palm's PDAs, the Handspring Visor, the Symbol Technologies TRG Pro, the IBM WorkPad etc.), but also on any device that has an RS232-compatible port.

The module embodies the Physical Layer of the Piconet stack, as well as performing many of the functions of the Data Link Layer. This thesis limits itself to the routing problem, and hence the wireless module is not required to have any notion of link-layer addressing; its only purpose is to facilitate the transmission of a stream of bits from one node to another (however, the receiver may not be the ultimate destination). In that sense, the wireless module replaces a cable and nothing more – Piconet relies on the Network Layer to route packets and ensure that data gets to the right destination.

The lack of link layer addressing prevents effective use of some Medium Access Control protocols, such as the RTS/CTS-based schemes mentioned earlier. However, it is also advantageous in some routing protocols that the Network Layer at each node processes all packets received over the medium (in a so-called "promiscuous listening" mode) and not just those that are addressed explicitly for that node. Filtering at the link level would deny such advantages.

Having said that, some form of Medium Access Control protocol should be employed by the module. The wireless channel is inherently a shared, broadcast medium; hence some form of contention-based protocol (however simple) such as those discussed in 2.2.1 would be beneficial to avoid collisions (which waste bandwidth) and reduce the number of consequent retransmissions when dropped packets are detected by higher layers in the protocol stack.

Range and bandwidth are other key specification parameters. It is expected that users will roam within a small office building, or possibly in a campus environment. As such, a communications radius of approximately 10 metres within a closed environment is reasonable. This automatically precludes the use of non-RF wireless products, specifically infrared. The bandwidth of the module should be sufficient to support users' typical data transfer requirements. Although the efficiency of the

network protocols' design will have an impact here, for the envisaged application space of handheld devices, anything above 9600bps should be adequate. A related parameter is the frequency of operation. Australian spectrum regulations must be obeyed; frequency bands permitted for general use include 433.05-434.79MHz, and the international Industrial, Scientific and Medical (ISM) bands of 915-928MHz and 2.45-2.483GHz [20].

Last but not least, the module should be battery-powered and hence exhibit low power consumption. The Palm PDA (and most handheld devices) lack the ability to supply the kind of power required of a wireless transceiver in addition to its own needs. Thus, the module should be powered independent of the host device, and display a life of 12 hours' continuous operation. This should be sufficient for typical bursty usage patterns of perhaps one to two weeks. Quantitative requirements are difficult, since much is dependent upon current and voltage requirements, as well as the type and number of batteries used. To compare, the WidComm BlueConnect, an announced Bluetooth-based solution for the Handspring Visor, uses three AAA alkaline batteries to consume 85mA (with no figure on actual battery life) [18]. It is proposed that we use two AAA alkaline batteries as a reasonable compromise between weight, size and capacity. Based on these figures, then, a current consumption of between 20 to 30mA at 5V DC is reasonable.

3.3 Network Protocol Requirements

A number of fundamental parameters are used to discuss the performance of a network [10]. Again these parameters can be qualitative or quantitative:

- **Throughput** measures how much data can be transmitted over a link taking into account the inefficiencies of the network. Frequently, this is referred to as *bandwidth***;** this is often confused with a network's capacity. For example, Ethernet networks are often claimed to have a bandwidth of 10Mbps. In reality, the contention-based medium access scheme means throughput will rarely exceed a few Mbps.
- **Reliability** describes the confidence of an applications programmer that the sent data will be received in tact at the destination. This can also be a quantitative measure in terms of the ratio of corrupted to uncorrupted packets, etc.

- **Latency** is the time it takes for a message to travel from one end of a network to the other. The round trip time (RTT) will also be important here; if the routing protocol is demand-based, a long RTT will adversely affect how quickly a sending node can discover routes to the destination. From an end-user perspective, high latency also makes applications appear sluggish if they are waiting for a response from a remote node.

The addressing scheme (which allows network devices to be uniquely identified) should support up to 15 distinct nodes, together with a broadcast address that ensures nodes can receive messages intended for the entire network. The network's physical diameter is partly determined by the performance of the wireless link, but the network protocol should employ ad hoc routing (i.e. multi-hopping) to extend this range.

The routing protocol plays a large part in providing the ability for nodes to create networks on-the-fly. Its performance can be judged against a variety of qualitative and quantitative criteria. The following are desirable qualitative properties of an ad hoc routing protocol [3]:

- **Distributed operation**: an obvious but essential property.
- **Loop-freedom**: avoids undesirable situations such as a packet infinitely circulating around the network. Time-to-live (TTL) values can alleviate such problems, although a preventive approach is preferred.
- **Demand-based operation**: although not an essential quality, it is desirable since exchanging routing information is costly in terms of bandwidth.

The routing protocol clearly will have an impact on the fundamental network performance parameters of throughput, reliability and latency. All routing incurs an overhead (header bytes, control messages, etc.) and hence this overhead needs to be minimised.

For the convenience of application developers, the protocol stack should provide communication primitives that support reliable peer-to-peer connections between nodes. These should facilitate applications development without requiring a deep understanding of the implementation of the network itself.

Some form of name/service resolution is desirable to accommodate the dynamic nature of an ad hoc network. This allows nodes to discover which devices are present

on the network and what sort of services these devices can provide. For example, a node may wish to connect to the Internet; lacking a connection on its own, it should be able to query the network to discover if other nodes can provide this service. A typical scenario is as follows:

1. Device A queries the network to discover if service X is available.
2. Device B offers service X, and sends an offer response to A.
3. Device A receives the offer response from B, and connects to B.
4. Device A and Device B communicate on a client/server basis.

Thus, the OSI model layers that need to be implemented span from the Application Layer down to the Network Layer. Of course, this does not imply explicit software entities for each layer should be produced – rather it covers what functionality needs to be present in the stack.

3.4 System Diagram

The requirements outlined above can be summarised in a system block diagram that demonstrates how all of the pieces, both software and hardware, fit together.

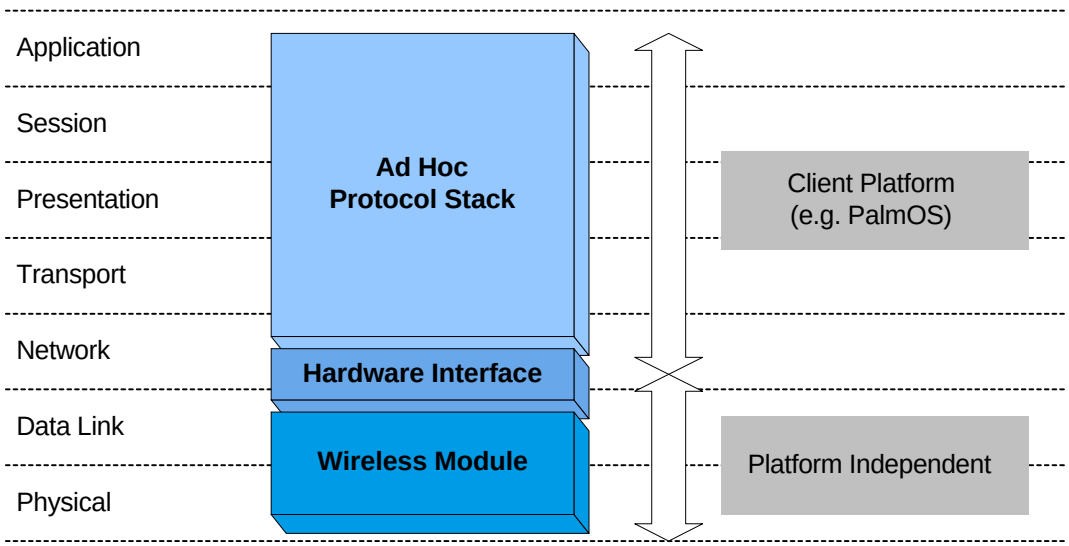


Figure 7 Piconet system design with respect to OSI Reference Model

The protocol stack will be software-based and will sit on top of the host platform, which in this case is the Palm OS. The wireless module is platform independent, and can be considered to be a self-contained transceiver that sends and delivers data serially.

The hardware interface sits between these two components, and is software/firmware that resides on both to allow them to communicate with each other over the RS232 link. This would involve creating some form of driver on the Palm, and its counterpart on the controller sub-component of the wireless module, to asynchronously transfer network packets which are then delivered over the wireless medium.

4 Wireless Module Implementation

This chapter discusses the implementation of the wireless module attachment required for wireless communications to be enabled on the Palm PDA. In implementing the module, consideration is made to the various parameters discussed in Chapter 2.3.2, including bandwidth, battery life and range.

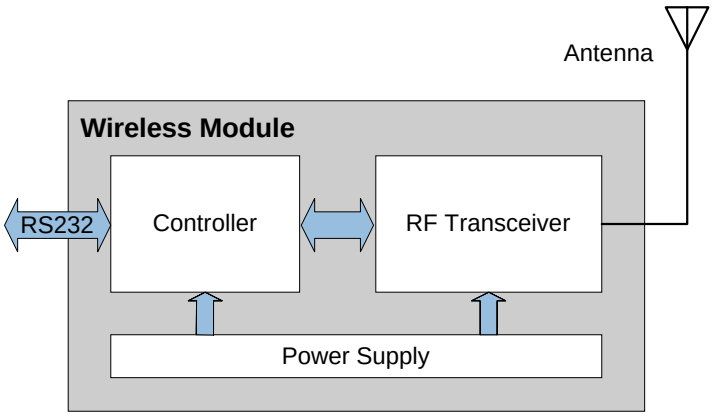


Figure 8 Wireless module block diagram

4.1 Wireless Link Selection

The wireless link should be non-line-of-sight, and allow communications within a reasonable distance. As pointed out in the specifications, this immediately rules out the large range of infrared devices which are generally line-of-sight and point-to-point, have limited range, and restrict the movement of the communicating parties during transmission. Therefore, only radio frequency (RF) devices were considered.

Piconet is intended for use by mobile devices, hence the chosen RF chipset must be easily integrated into a custom hardware design that is also portable and power efficient. A combined transceiver design is preferred, rather than having to design a separate receiver and transmitter module. The portability requirement rules out the many RF boards available that only require a power source and a parallel/serial port connection for I/O (these are typically used for prototyping purposes or in non-mobile applications).

Time constraints also favour chipsets that reduce the design load in terms of RF component tuning, and mundane tasks such as data encoding and framing. The former relates to the radio frequency transmission and reception capabilities of the chipset (Physical Layer functions), while the latter relates to the data transfer capabilities of the system (roles contained within the Data Link Layer).

A selection of chipsets satisfying the criteria of Section 3.2 (keeping in mind the above constraints) were reviewed; their respective merits and shortcomings are detailed below.

4.1.1 Radiometrix Radio Packet Controller

The Radiometrix Radio Packet Controller (RPC) [21] is a 40kbps half-duplex transceiver module that operates in the 433MHz spectrum and at distances of up to 30 metres in-building and 120m over open ground. It provides all the RF circuitry, processor-intensive low-level formatting and error recovery functions for transmitting frames of up to 27 bytes in size. Receiving stations only see error-free frames – corrupted ones are discarded. All that is required is a 5V supply, an antenna and a byte-wide I/O port for data transfer and control purposes.

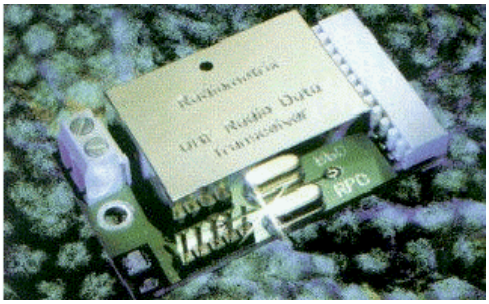


Figure 9 The Radiometrix Radio Packet Controller (RPC)

The RPC also provides some additional useful functions such as sleep mode for power conservation, and a simple collision avoidance mechanism that listens to the channel before transmitting, and backing off a random amount of time if the channel is occupied.

4.1.2 Radiometrix BiM

The Radiometrix BiM is the RF transceiver module at the heart of the RPC of 4.1.1. This means the RF performance is identical – 40kbps half-duplex transmission at 433MHz over distances of up to 30 metres “in-building” and 120 metres over open

ground. Like the RPC, no trimming of components for tuning purposes is necessary. No other functionality is provided, however, such as data encoding, integrity checks, etc; the user must implement these.

4.1.3 Linx SC

The Linx SC [23] is a half-duplex transceiver module that operates in the 916MHz frequency range. It provides data rates of up to 33.6kbps over distances of 60-150 metres depending upon the environment. The Linx SC's major advantage is its "data in, data out" nature – data is transferred via an RS232-compatible UART[3] serial interface. The user does not need to worry about data encoding or synchronisation details, effectively making the Linx SC a replacement for a serial cable.

Unfortunately, when the wireless module was conceived, this part was in the preliminary design stages and pricing was unavailable.

4.1.4 RF Monolithics DR3000

The RF Monolithics DR3000 [24] is a short-range 916MHz transceiver module that uses RFM's amplifier-sequenced hybrid (ASH) architecture for small size and power consumption.

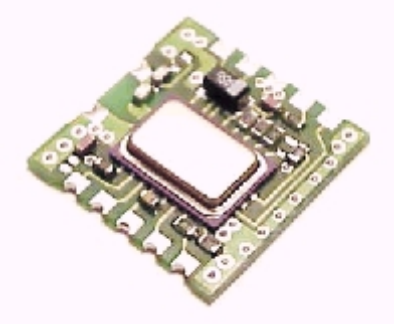


Figure 10 RF Monolithics DR3000 transceiver module

No data encoding, modulation or synchronization is performed. However, no tuning of components is required, and provision is made for both on-off keyed (OOK) and amplitude-shift keyed (ASK) modulation. Its small size is also an advantage.

[3] UART – Universal Asynchronous Receiver/Transmitter

4.1.5 Wireless Link Choice

	Radiometrix RPC	Radiometrix BiM	Linx SC	RF Monolithics DR3000
frequency of operation	433MHz	433MHz	916MHz	916MHz
range	30m (closed) 120m (open)	30m (closed) 120m (open)	60m (closed) 150m (open)	n/a
data rate	40kbps (half-duplex)	40kbps (half-duplex)	33.6kbps (half-duplex)	2.4 – 19.2kbps (half-duplex)
power consumption	4.5 – 5.5V DC, 20mA typ.	4.5 – 5.5V DC, 16mA max.	2.7 – 16V DC, 29mA max.	2.7 – 3.5V DC, 12mA max.
onboard encoding?	Yes	No	Yes	No
interface	5V CMOS	5V CMOS	RS232/485 UART-compatible	3V peak modulated input
dimensions (LWH in mm)	54x32x16	33x23x10	38x28x11	18x18x10
price	A$159.00	A$119	unavaiable at design time	US$350.00

Table 2 Wireless link comparison chart

Selection of the RF chipset was greatly influenced by time constraints. The wireless module is one component of a larger design in which the time-to-prototype of one year is quite short.

All of the devices listed in Table 2 meet the data rate, power consumption and range requirements.

Cost immediately ruled out the DR3000: the paucity of features compared with competing products was its downfall (in addition to requiring shipment from the USA).

Of the remaining products, cost was not a significant differentiating factor. However, only the Radiometrix RPC and Linx SC provided the convenience of built-in data link layer functions such as error detection and correction, synchronisation and framing (but not addressing). Clearly, the BiM and DR3000 products would incur a burden on development time in implementing these functions. Having these functions in hardware also frees the host device for other processing tasks – particularly important given the Palm's weak CPU. The RPC was a clear choice over the BiM since more functionality is present at negligible extra cost relative to the unit price.

Arguably, the Linx SC had the most attractive features for the problem at hand, but the product's preliminary nature and consequent unavailability meant that the Radiometrix RPC was chosen as the chipset for the wireless module in the PicoNet system.

4.2 Microcontroller Selection

The microcontroller represents the wireless module's Controller component found in Figure 6; it provides the interface between the Palm's RS232 serial port and the RF transceiver, the Radiometrix RPC from above. The RPC's method of I/O imposes a number of requirements on the microcontroller.

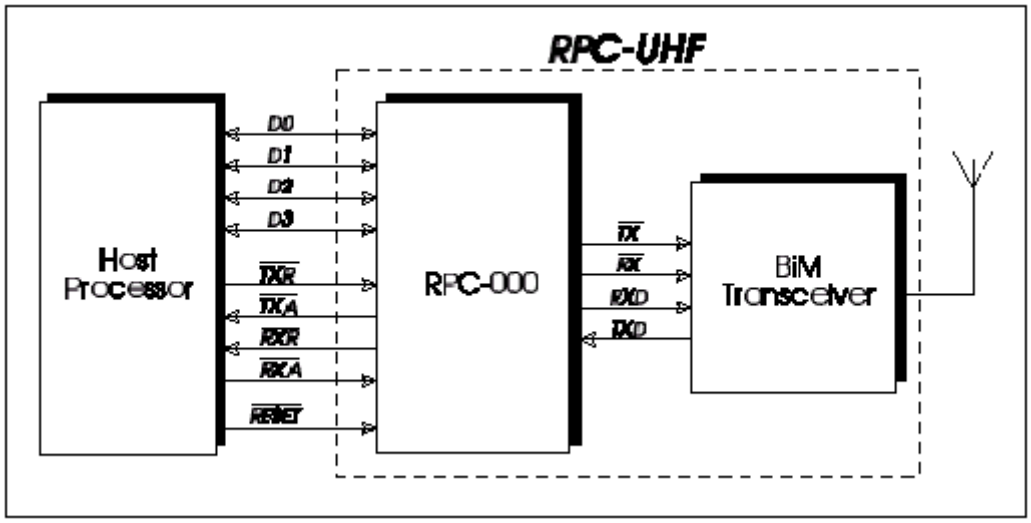


Figure 11 How the RPC interfaces with the host processor [21]

Figure 11 shows the 9 I/O lines to the RPC for uploading and downloading data from the transceiver. An asynchronous handshaking protocol is used which further influences microcontroller selection.

If the host processor wishes to transmit data, it must first request permission to send by pulling the TXR* line low (transmit request). TXA* (transmit accept) will be lowered if permission is granted, and the host processor can proceed to upload the data a nibble at a time via lines D0..D3. The module will have occasion to refuse transmit requests if it is busy decoding a packet; no transmit requests will be granted until the resulting packet (if it was received error-free) is downloaded from the RPC's receive buffer.

The reverse procedure occurs when data is received by the RPC, which signals the host processor that data needs to be emptied from the receive buffer by pulling RXR* low (receive request), and the host processor responding by lowering RXA* (receive accept) when ready. It is vital that such receive requests are honoured as quickly as possible –packets cannot be transmitted until this is done, nor can more data be received from the channel.

This combination of requirements means that the microcontroller needs:

- at least 11 I/O ports – 9 for interfacing with the RPC, and a minimum of 2 (TXD and RXD) for the serial port interface with the Palm;
- and preferably an independent, on-board RS232-compatible UART; this frees the processor to service the RPC with as low latency as possible. Software-based serial I/O is based on cycle timings that could be interrupted by an RX Request from the RPC, which as just described needs to have a higher priority to prevent unnecessary data loss.

Only 8-bit devices are considered since no complex processing is performed – only buffering and data translation from a serial byte stream into nibbles and vice versa.

4.2.1 Motorola 68HC11E1

The Motorola 68HC11E1 [25] is an 8-bit CMOS microcontroller with a 16-bit address bus, 512 bytes of RAM, 512 bytes of EEPROM, 38 ports for I/O, and a serial communications interface (SCI). At an operating frequency of 3MHz, power consumption is 27mA at 5V DC. It is packaged in a 52-pin plastic-leaded chip carrier (PLCC) form factor.

4.2.2 Microchip PIC16F873

The Microchip PIC16F873 [26] is a flash-programmable RISC microcontroller with a Harvard architecture that allows instruction words to be optimised for any size (14 bits in this case) without affecting the data word size (8 bits). This device has enough flash memory to hold 4000 14-bit instructions. It also has 192 bytes of RAM, 22 I/O ports and a full duplex UART. Power consumption is quite low: 2mA at 5V DC at an operating frequency of 4MHz. One of the greater advantages of this device is its ability to be prototyped on breadboards without too much difficulty, especially in its 28-pin narrow plastic DIP packaging. It is in-circuit programmable.

4.2.3 Atmel AVR 90S2313

The Atmel AVR90S2313 [27] is a low-power CMOS RISC architecture microcontroller with 16-bit instruction words. It has 2000 bytes (1000 instruction words) of Flash memory for programs, and can execute most instructions in a single cycle. It also features a full duplex UART and in-circuit programmability. It is quite similar to the PIC16F873 in functionality. It comes packaged in a 20-pin DIP.

4.2.4 Microcontroller Choice

	Motorola 68HC11E1	**Microchip PIC16F873**	**Atmel AVR 90S2313**
price[4]	$27	$20	$27
memory	512 bytes RAM; 512 bytes EEPROM	192 bytes RAM; 128 bytes EEPROM 4k Flash	128 bytes RAM; 128 bytes EEPROM; 2k Flash
data bus width	8 bits	8 bits	8 bits
address bus width	16 bits	14 bits	16 bits
I/O ports	38	22	15
power consumption	27mA @ 5V, 3MHz	2mA @ 5V, 4MHz	2.8mA @ 3V, 4MHz
form factor	52-pin PLCC	28-pin PDIP	20-pin PDIP
frequency of operation	3MHz	4MHz	10MHz max.
interrupt support	yes	yes	yes
on-board UART	yes	yes	yes

Table 3 Microcontroller comparison chart

Development tools are available for all of the above parts – C compilers, assemblers, and programmers – hence this was not a deciding factor. Microcontroller selection, then, was largely driven by cost, the need to prototype quickly, and low power consumption.

The 68HC11 was immediately discarded on the power consumption criterion alone; the RF transceiver itself already consumes a significant amount of power and using a power-hungry microcontroller would impact on battery life considerably. Lowering the clock goes some way towards alleviating power consumption at the expense of performance, but is unlikely to achieve the low levels of the PIC and AVR.

[4] Per-unit price quoted from the Farnell Components Catalogue 2000 for quantities of 1-25.

Of the remaining two choices, the PIC was preferred as a result of its lower cost, and larger memory for programs (leaving the way open for future enhancements in the firmware for extra frame processing functions).

The Palm connects to the wireless module via the microcontroller's USART[5], which can send and receive data independently of the microcontroller's main program flow.

4.3 Power Supply

A step-up regulator is used to provide the 5V DC supply required by the PIC microcontroller and RPC transceiver. A pair of 1.5V alkaline AAA batteries provides the input to the regulator. A simple charge pump converter was chosen over more complex switching regulators since the latter require inductive components, which have negative consequences when used with RF hardware.

The Maxim MAX619 [28] part was used here – no significant advantages in using one product over another could be found, with most devices offering shutdown-mode options and roughly the same conversion efficiency ratings. The MAX619 is capable of delivering a regulated 5V ± 4% output at 50mA for input voltages between 2V and 3.6V DC.

4.4 Antenna

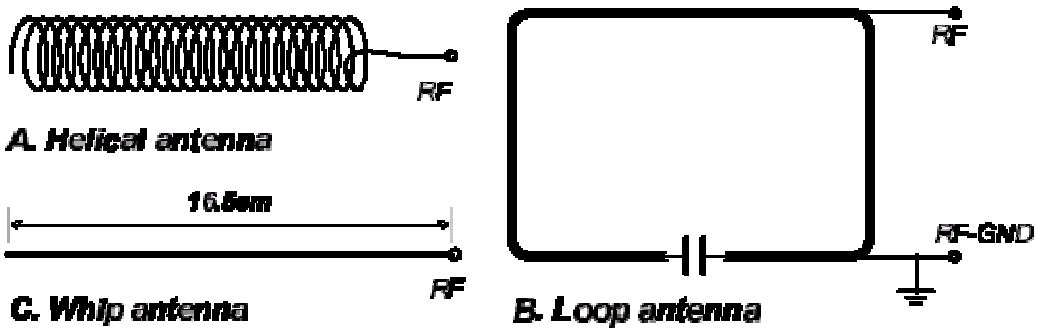


Figure 12 Antenna types suitable for portable applications

Figure 12 shows the different antenna types suitable for portable applications [22]. The quarter-wavelength whip antenna is easy to design (being a straight piece of wire or PCB trace at one quarter of a wavelength at the frequency of operation). It offers

[5] Universal Synchronous Asynchronous Receiver Transmitter

the best performance but can be quite long (16.5cm for 433MHz). A loop antenna can be implemented as a PCB trace; it has the best immunity to proximity effects from non-RF components in the system, but has the poorest performance overall in terms of range and ease of design.

The wireless module uses the best compromise in physical size and range terms: the helical antenna. This is implemented as a simple piece of wire formed into a helical loop similar to that shown above. It is inserted directly into the terminal block provided by the RPC for antenna attachment.

4.5 Controller Firmware

The firmware of the wireless module is based around two finite state machines (FSMs); one which buffers data from the Palm sent via the PIC's UART, and one for transmitting this buffered data, servicing the RPC and receiving data from the wireless channel.

The latter, the RPC Controller FSM, is shown in Figure 13; the TXR*, TXA*, RXR* and RXA* lines perform the asynchronous handshaking protocol described previously for downloading and uploading data to the RPC.

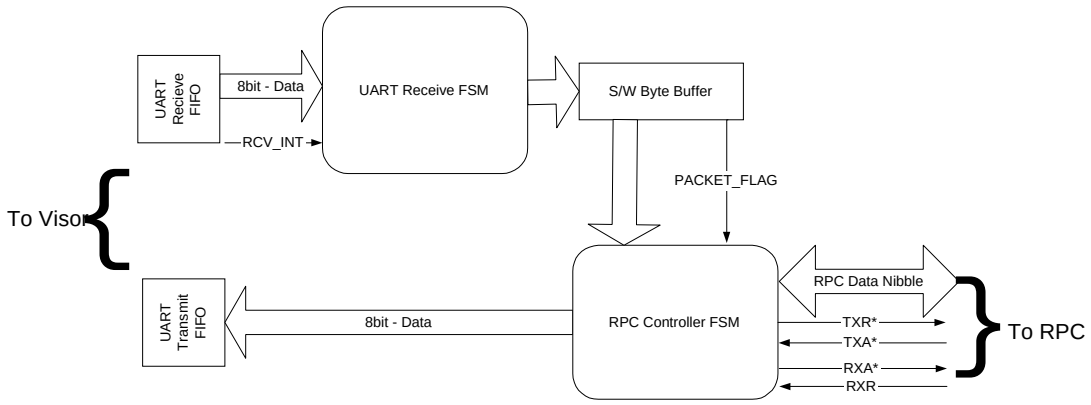


Figure 13 Wireless module firmware block diagram

As discussed previously, the receive buffer of the RPC transceiver must be flushed as quickly as possible, since the RPC will not accept further packets from the medium until this buffer is emptied. The RPC Controller FSM constantly polls RXR* to ensure this is done with a minimum of latency.

The UART Receive FSM is triggered by the interrupt that occurs when data is received from the Palm on the UART's receive FIFO, and transfers the 8-bit data to a software byte buffer. When a full packet is ready to be transmitted, the packet flag is

set. Transmission of the packet by the main controller FSM is deferred until the RPC is free to transmit (i.e. it is not receiving data from the channel).

Further details of the state machines in the wireless module can be found in the thesis document of Mr. Israel Keys [29].

The module sends and receives packets to and from the Palm framed by DLE-STX and DLE-ETX[6] byte pairs. Byte stuffing is employed such that in the event of a DLE appearing in the byte stream, an extra DLE is inserted immediately after. The final byte within the frame is a checksum value to make the sum of all the bytes contained within the frame equal to zero. If the checksum value does not match, the frame is discarded. These measures ensure robustness with minimal overhead.

Figure 14 Link Layer packet framing

Note that this framing is purely to synchronise communications between the Palm and the wireless module. The delimiter fields and checksum are not transmitted over the channel – the RPC adds its own preamble sequence, framing and checksum values.

The wireless module uses a modified CSMA/CA scheme as its medium access protocol, with a random (but not exponential) delay backoff if a station is already occupying the channel. As a consequence, the Piconet system suffers from the hidden and exposed terminal problems discussed previously. Implementing a more effective scheme could be a subject for further study as noted in Chapter 7 on Future Developments.

Source code for the wireless module's firmware can be found in Appendix B.

[6] In ASCII, DLE=Data Link Escape (0x10), STX=Start of Text (0x02), ETX=End of Text (0x03)

5 Network Protocol Implementation

This chapter describes the protocols for enabling ad hoc networked communications in the Piconet system. The structure of the protocol stack is examined, followed by system-wide protocol implementation issues impacting on protocol design and performance.

There is also a detailed discussion of the Piconet system's routing protocol and how it provides Piconet with ad hoc capabilities – how routes are discovered and maintained – and how services can be delivered in a dynamic, changing network environment.

The chapter closes by describing the general code design of each layer.

5.1 Protocol Stack

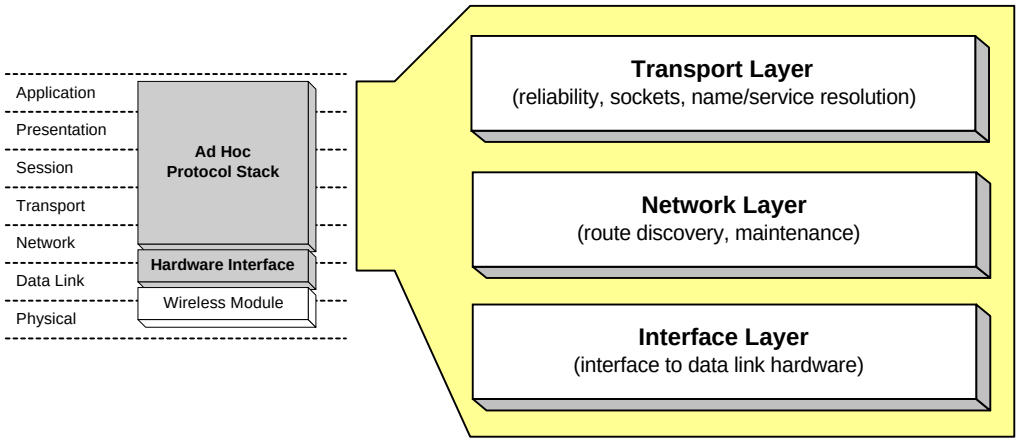


Figure 15 Protocol Stack Structure

The stack has three major layers, as shown in Figure 15. It is roughly equivalent to the Internet model in terms of what functions each layer performs.

The *Transport Layer* provides reliable end-to-end communications using a familiar Berkeley sockets-style syntax (with slightly differing semantics). It also has facilities via a broadcast mechanism to resolve the identity of nodes, and what services are being offered by nodes present in the network.

The *Network Layer* is responsible for routing – delivering packets from A to B. This complex process is described in more detail in Section 5.3. The Network Layer

presumes that packets sent via the Interface Layer will be transmitted to all neighbouring nodes; it also presumes that the packets the Interface Layer delivers to it originate from neighbouring nodes.

The *Interface Layer* takes data from the Network Layer, frames it appropriately for the wireless module according to the format described in Section 4.5, and passes it to the physical hardware of the host device, i.e. the serial port of the Palm. The serial port sends the frame to the wireless module, which then takes care of modulation and RF transmission. This layer is also responsible for receiving data from the wireless module, stripping out the Link Layer framing and passing it onto the Network Layer for processing.

Together, these three layers form the core of the ad hoc networking protocols of Piconet.

5.2 System Design Choices

This section covers general system-wide design choices that affect how the stack interacts with the operating system and the way it is implemented both as a whole and on a per-layer basis. These decisions are justified in terms of necessity or performance benefits.

5.2.1 Stack Partitioning

Stack partitioning refers to the manner in which the stack operates with respect to the underlying operating system [10].

For example, Palm's OS offers networking services via its Net Library, and allows a Palm device to attach to the Internet using TCP/IP [30]. Adding ad hoc functionality to this stack is not possible since the source code is unavailable, meaning that a separate software stack must be created.

Unfortunately, the manner in which Palm has created the Net Library is not available to the casual programmer. The TCP/IP stack is a system library executing as a separate task; however, it enjoys full privileges in the operating system that are not given to user applications, such as threading facilities – a vital component in an effective stack implementation.

Hence, we are left with two options: (i) to use a different operating system, or (ii) write the stack in user-space rather than in the kernel and link the ad hoc networking library into each application that uses it.

The first option brings to mind uCLinux [31], a Linux derivative available under the GNU General Public License (GPL) modified specifically for microcontrollers without Memory Management Units[7] (MMUs). The lack of an MMU results in less elegant multitasking and no memory protection to prevent an application from writing over another's memory space.

uCLinux is based on the Linux 2.0 kernel, is equipped with a full TCP/IP stack, supports file systems as diverse as NFS, ext2 and FAT16/32, and has been ported to a number of microcontrollers, including the 16-bit Motorola Dragonball as used in all Palm Computing Platform devices.

The design of the Palm Platform device used to prototype the system, the Handspring Visor, ruled out the use of uCLinux. Unfortunately, the Visor holds its operating system in a non-reprogrammable EPROM rather than Flash memory as in other Palm devices, and hence cannot be modified to accommodate the uCLinux operating system [32]. In addition, the attractiveness of the system is much reduced from an end-user perspective if it requires erasure of the existing operating system and user data!

In lieu of replacing or modifying the operating system, the only course of action that remains is to implement the stack in user space. This has some bearing on the choice of programming language, particularly with regard to multithreading facilities, and is discussed in more detail in the next section.

[7] A Memory Management Unit (MMU) is a hardware device to support virtual memory and paging by translating virtual addresses into physical addresses. It is useful in multitasking systems for memory protection purposes and also allows multiple physical mappings of the same virtual address space.

5.2.2 Implementation Language

5.2.2.1 C/C++

C is the most popular language for development on the Palm Platform. C's advantages are that it is fast, generally portable, powerful and flexible. Indeed, the Palm OS's libraries are accessed via a native C interface [30].

However, C programs are restricted to a single thread of execution. Threads are important to implementers of network protocols; elementary operations such as timed-out retransmissions without threads are next to impossible. It is worthwhile noting that, while Palm OS itself is threaded, only one is made available to an application developer when writing in C/C++. Palm applications use an event-based model that is adequate for user interface and database tasks since the delay between say, looking up the user's address book and then refreshing the screen, is negligible. This represents a serious disadvantage.

Another restriction is the lack of interrupts or signalling within the Palm OS. This is particularly detrimental when accessing the Palm's serial port. Polling is the only option, which distracts the CPU from other tasks.

5.2.2.2 Java

Java is an object-oriented programming language and runtime environment created by Sun Microsystems [33]. From humble beginnings as a way to animate web pages with "applets", it is now employed as a general purpose programming language for both client- and server-centric applications, and more recently in embedded devices.

Typically, Java programs are compiled into bytecodes that are executed on the host in a Java Virtual Machine (JVM) that abstracts platform-dependent details such as microprocessor word size, number of registers etc. As a pseudo-interpreted language, such an approach understandably incurs an overhead that until now made Java unsuitable for resource-constrained embedded devices. The advent of the IBM J9 Virtual Machine has changed this [34]. IBM's JVM is tailored specifically and automatically to the target platform, taking advantage of any architectural features to increase speed wherever possible. Java programs compiled using IBM's class libraries can execute on a variety of platforms, including the Palm, as long as the J9 runtime system is present.

In contrast with C, Java offers thread facilities in the language itself. The IBM J9 VM circumvents the lack of threading in the Palm OS by offering "green" threads[8], thus offering a significant advantage to the protocol programmer.

Additionally, Java enjoys the same advantages of object-orientation offered by C++, but without the disadvantages of C/C++ such as memory leaks and the classic compile-link-debug cycle.

Admittedly, Java will always be slower than compiled languages by its very nature as a bytecode-interpreted system. However, J9 further acts to minimize the performance penalty by exposing the entire Palm OS API through Java wrapper functions. These can be used if and when the Java-equivalent is inefficient or not supported. For example, the serial port can be accessed via a native OS call using Java.

These features make Java a compelling alternative to C.

5.2.2.3 Language Choice

Implementing a protocol stack requires facilities for multithreading to be truly practical. Ultimately, this made the choice of implementation language moot, since C programming on the Palm Platform restricts the user to a single thread of execution and a blocking, loop-based, event-driven model. This reduces "multitasking" to an inflexible round-robin polling model, which makes programs difficult to design and debug. Java was the only real choice.

Java and IBM's J9 VM gives access to threads, object-oriented programming techniques, and the side-benefit of easy portability to other processor architectures and operating systems for which this virtual machine is available. Currently, the list extends to the Intel x86, PowerPC, ARM and Hitachi SH-3 and SH-4 processors for the former, and QNX/Neutrino, Windows and Linux for the latter.

Notably, the only platform-dependent part of the stack is the Interface Layer. This can be easily adapted to the varying paradigms of accessing physical hardware on other platforms, whilst leaving the remainder of the stack in tact as "Pure Java".

---

[8] "Green" threads are threads scheduled and executed by a separate library or layer within a virtual machine rather than by the underlying operating system.

This was demonstrated by porting the Interface Layer to the Windows/x86 platform. Simulation of the protocols was performed on the PC before transferring to the Palm – a process that made design, development and debugging much easier. It also meant that the breadth of mobile devices enabled under Piconet could be extended easily to notebook computers, for example.

5.2.3 Layer Interfacing

Layer interfacing refers to how the layers themselves communicate with each other [10]. As expected, there is a variety of ways of doing this.

5.2.3.1 Single Context

In the single context approach, protocol layers share a single uninterrupted thread of execution. When an application write occurs, each layer adds its own protocol headers before passing it on to the layer below it. Similarly, when a packet is received from the Data Link Layer, each layer strips its protocol headers before passing it to the layer above it. Only one packet is ever being processed by the stack at a given point in time, meaning shared resources need not be locked.

5.2.3.2 Task Model

The task model involves writing each layer as a task. When a packet needs to be sent or received, a central task scheduler schedules an instance of that task with a pointer to the packet as a parameter. When the task completes its processing, it can exit or schedule another task – for example, once a transport layer task has finished adding its control information to the packet, it can schedule a network layer task for the same packet.

5.2.3.3 Upcall

In the upcall architecture, a thread is associated with each packet. This thread is responsible for shuffling the packet up or down the stack. Each layer registers entry points (or function pointers) with the layer above and below it for when data needs to be sent or received. The packet thread invokes this chain of procedure calls. Such a scheme allows multiple packets to be processed simultaneously (as in the task model) and removes the need for expensive context switches when a packet moves from layer to layer since it is all executed in the same context.

The limited processing power and memory available on the Palm influenced the choice of interfacing method. The main disadvantage of the task and upcall models is the possibly large number of simultaneous threads. Resource constraints dictate that we try to minimize this number.

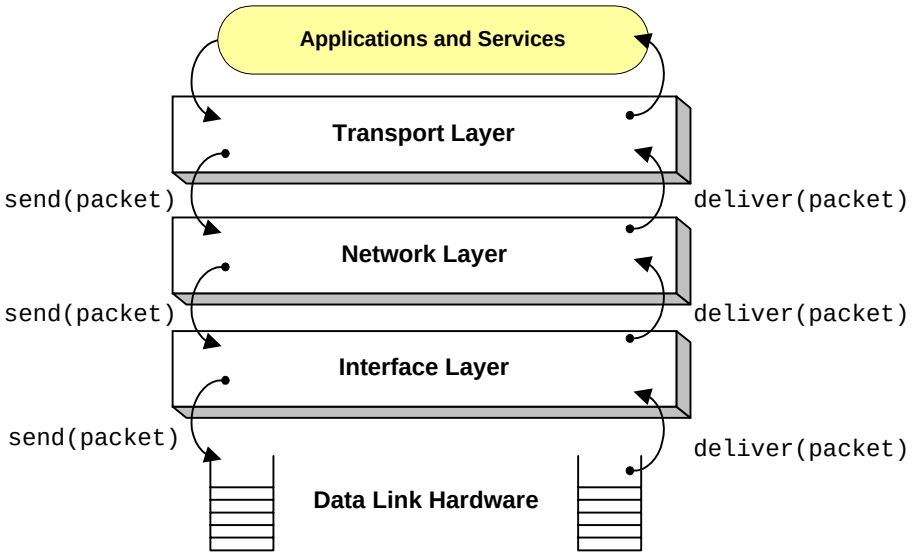


Figure 16 The hybrid single context upcall mechanism for shuffling packets through the stack

5.2.3.4 Layer Interface Choice

The chosen solution is a hybrid that combines the speed and simplicity of single context execution with the elegance of the entry-point registration scheme of the upcall architecture, as shown in Figure 16.

In Java, this is accomplished by defining an abstract PiconetLayer class, which has the following member variables and method definitions:

```
protected PiconetLayer ul;
protected PiconetLayer ll;
public void ulRegister (PiconetLayer l);
public void llRegister (PiconetLayer l);
public int deliver (Packet p);
public int send (Packet p);
```

Figure 17 The PiconetLayer abstract class definition

Each layer in the Picostack extends the `PiconetLayer` class, and must implement the `send()`method, called by its immediate-upper layer when a packet needs to be sent, and `deliver()`, called by its immediate-lower layer when a packet is received. The `ulRegister()` and `llRegister()` methods register references to these upper and lower layers.

By calling `send()`, each layer adds its own header to the data being sent. Similarly, `deliver()` allows the layer to strip its header and perform whatever processing is required before passing it up to the next higher layer (if any).

5.2.4 Optimisations

A number of performance optimisations have been implemented within the protocol stack.

The most notable of these is the Packet class, which captures knowledge about each layer's protocol data units (PDUs) from the Transport Layer down to the Interface Layer. As can be seen from the PiconetLayer abstract class definition, `send()` and `deliver()` both take a single Packet reference as a parameter. A Packet is instantiated once by the source of the data; thenceforth, only the reference is passed between layers, removing the need for expensive memory-to-memory copying. Utility functions within the class can strip and add headers to the message in-place without creating extra, redundant copies.

Memory-to-memory copying will always be a performance bottleneck since the rate of performance increase in memory bandwidth is markedly outstripped by the rate of performance increase in processing power. Thus, minimising memory usage is a simple but valuable optimisation technique [10].

The Packet class achieves this by immediately allocating 27 bytes, the maximum data that can be sent in one Link Layer frame as dictated by the RPC transceiver. Encapsulation means that the user's data together with each layer's header sections must fit within this 27-byte limit (see Figure 18).

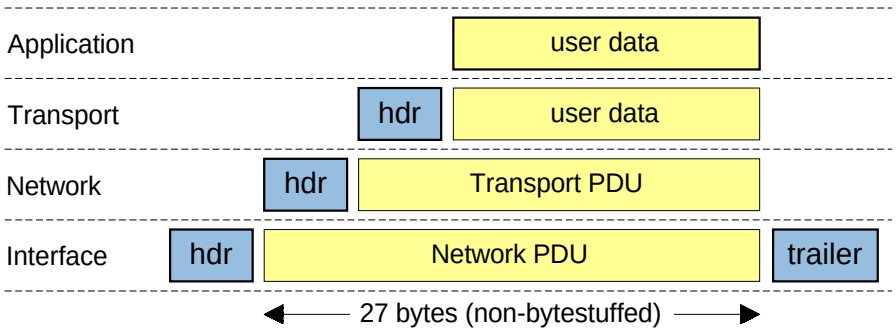


Figure 18 Encapsulation of PDUs

Thus, once the PDU format at each layer has been designed, the Packet class can determine where to place their respective header fields and data within the 27-byte

frame. These formats are discussed in greater detail in the section on Layer Implementation.

5.3 Routing Protocol

This section discusses at length the routing protocol of the Piconet system that is central to its ad hoc capabilities. A selection of routing protocols are described and evaluated before a decision is made on which protocol the Piconet system will implement.

5.3.1 Routing Protocol Comparison

5.3.1.1 Clusterhead Gateway Switch Routing (CGSR)

Clusterhead Gateway Switch Routing (CGSR) is a table-driven protocol, and a close relative of the Destination-Sequenced Distance-Vector (DSDV) routing algorithm [35]. Each network node maintains a route table containing all possible destinations and the number of hops to each; each entry has a sequence number assigned by the destination node to distinguish old routes from new ones and to prevent routing loops. Route table updates are broadcast periodically throughout the network; upon receipt, only the route with the most recent sequence number is used.

When a node needs to send a packet, it first forwards it to the cluster head, which propagates the packet to another cluster head via a gateway (an intermediate node that is within communication range of two or more cluster heads). This process is repeated until the destination is reached.

CGSR requires that each node maintain a cluster membership table, as well as some form of cluster head selection algorithm when membership changes. Unfortunately, mobility support is not as good as for other algorithms because frequent cluster head changes can mean nodes are busy selecting new cluster heads rather than relaying packets, thus affecting throughput.

5.3.1.2 Ad Hoc On-Demand Distance Vector Routing (AODV)

Ad Hoc On-Demand Distance Vector Routing (AODV), like CGSR, builds on DSDV but uses on-demand route acquisition [35]. When a node needs to send a packet to a destination for which a valid route is not present in its route table, a path discovery process is initiated. A route request (RREQ) packet is broadcast to its neighbours,

which forward the request to their neighbours, and so on, until either the destination or an intermediate node with a sufficiently fresh route is found. During the process of forwarding, intermediate nodes record the neighbour from which the RREQ was received, thus establishing a reverse path. Assuming a valid path to the destination is found, the resulting route reply packet (RREP) is sent back to the originator of the RREQ along this reverse path. Forward route entries are then established in the route tables along the path of the RREQ to the destination. Route timers ensure that table entries that remain unused within a fixed period of time will expire and force a fresh path discovery process. As a result of using the reverse path to return RREPs, only symmetric links are supported under AODV – problematic if a node's receiving capability does not match its transmitter power.

5.3.1.3 Dynamic Source Routing

Dynamic Source Routing (DSR) is another on-demand routing protocol [36]. Packets to be routed contain in their header an ordered list of nodes representing a path to the destination. This is DSR's key advantage because intermediate nodes do not need to maintain routing information in order to route the packets they receive, since the packets themselves contain all the necessary routing information.

Each node maintains a route cache of recently learned routes. When a packet needs to be sent and a valid entry to the destination cannot be found in the cache, a route request (RREQ) packet is broadcast to its neighbours. As in AODV, these neighbour nodes forward on the RREQ packet to their neighbours, but they do not record the immediate source of the RREQ. Rather, prior to forwarding, the intermediate node appends its own address to the packet's route record, an ordered list of node addresses along which the RREQ has already been forwarded. Loop-freedom is assured by the node inspecting the route record for prior entries of its own address.

A route reply (RREP) is generated when either the RREQ reaches the destination itself, or when an intermediate node has a sufficiently fresh route to the destination in its cache. The route record is copied into the RREP packet, and either forwarded back along the reverse path if symmetric links are supported, or a new route discovery process is initiated for the reverse direction, with the RREP possibly piggybacked on the RREQ.

DSR's main advantage and disadvantage is its storage of the entire route in the header. This affects DSR's scalability to larger network diameters. DSR has other advantages, however, such as the lack of keep-alive messages (which conserves bandwidth), allowance for multiple routes in case of link failure, and its support of asymmetric links.

5.3.1.4 Routing Protocol Choice

	CGSR	AODV	DSR
type	table-driven	on-demand	on-demand
addressing	hierarchical	flat	flat
network size	small-medium	small-large	small-medium
mobility support	poor-fair	good	good
asymmetric link support	yes	no	yes
hello messages	no	no	no
loop-freedom	yes	yes	yes

Table 4 Feature comparison of various ad hoc routing protocols

Table 4 summarises the features of the ad hoc routing protocols discussed above.

CGSR is not preferred, since it has poor mobility support, and additional complexity due to its hierarchical nature. For the purposes of conserving bandwidth and battery power, it is preferable to use an on-demand routing protocol rather one which broadcasts periodic routing updates. The frequency of these updates plays a large part in the mobility support of this protocol; on-demand mechanisms are more suitable for the problem at hand.

The qualities of AODV and DSR appear similar, but studies have shown that DSR has the edge over AODV in terms of number of packets successfully delivered under conditions of high node mobility and movement speed without significant expense in routing overhead bytes resulting from storing the entire route in the packet header [37].

DSR satisfies the criteria outlined in the requirements section by using a source-initiated routing procedure that minimises the number of control messages for routing, and consequently lessens bandwidth demands, increases battery life, and contributes to increased throughput via reducing the probability of packet collisions and

decreasing routing overhead (as compared to table-driven protocols). Another desirable quality is that routing in DSR is assured to be loop-free.

5.3.2 Implementation of DSR

DSR consists of two major phases: (i) route discovery, and (ii) route maintenance. This section covers in more detail how each of these phases is implemented in the Piconet system.

5.3.2.1 Route Discovery

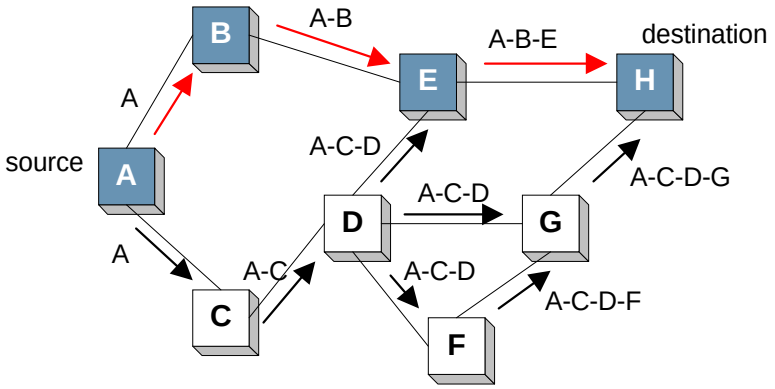


Figure 19 Building of the route record during the route discovery process

Route discovery is implemented as described previously. A route cache indexed on destination address is maintained; if a valid route for a particular destination cannot be found, a RREQ packet is sent across the network. Figure 19 shows graphically how the RREQ packet propagates. Nodes may receive the same RREQ packet more than once as a result of forwarding; the node discards any packets besides the first by examining and caching the unique packet identifier (UPI) header field of RREQ packets. The resulting route record is likely but not guaranteed to be the shortest path – the time for a RREQ packet to propagate to the destination depends not only on the number of hops but also on the speed of processing of the intermediate nodes.

The route reply process is slightly different from DSR. Links are assumed to be asymmetric, but rather than initiate a route request procedure to route the reply back to the source, the RREP packet uses the same controlled flooding mechanism by which the RREQ packet was sent, thus reducing the delay between when the route request was sent and its reply.

5.3.2.2 Route Maintenance

Routes are maintained by a number of means. Before expounding on this, however, it is valuable to understand how data packets are routed.

Building on the example of Figure 19, Figure 20 shows how the route record obtained from route discovery is copied into the header of each data packet, and used by intermediate nodes to forward packets on to the destination. Here, node E can see that it is part of the route , and hence forwards the packet to H.

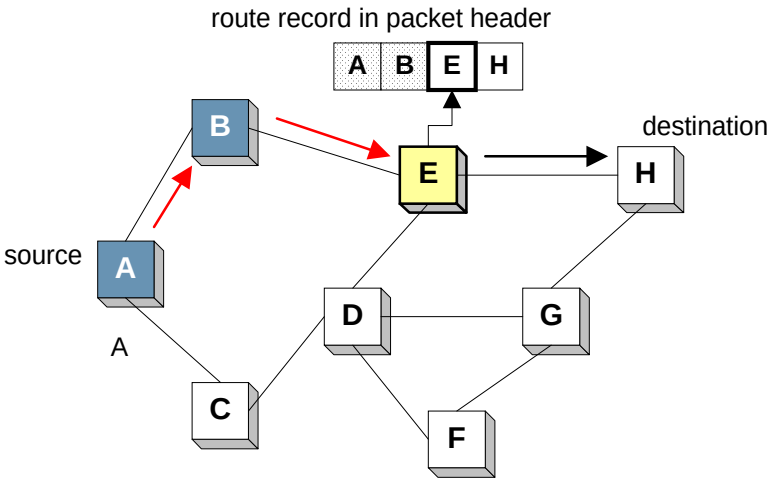


Figure 20 A packet being source-routed from A to H

The route record is stored as a contiguous sequence of network addresses. A field within the header indexes the next address in the sequence for which the packet is destined. The source initialises the index to 1, with each subsequent node in the path incrementing this index prior to forwarding. By consulting the index, a node can discern whether: (i) the packet has been received in order (i.e. if the position of its address matches the value of the index field); (ii) the packet has been received earlier than expected (meaning a shorter route exists); or (iii) the packet is received again (usually as a result of receiving the next hop's forward transmission).

The latter is used as a passive acknowledgement mechanism for route maintenance purposes. For example, when node B forwards the packet on to node E, it starts a timer and listens for the same packet to be forwarded by E onto H. If the timer expires, this indicates that the link between B and E has failed. B responds to this by broadcasting a route error packet back to the source – a signal to the source that a new route discovery process should be initiated.

5.3.2.3 Optimisations

If a node receives a packet earlier than the order of the route record indicates, a shorter route between the source and destination must exist. By examining the current index into the route record, the node can infer which nodes were skipped, and generates an unsolicited route reply containing the shortened route.

5.4 Layer Implementation

This section discusses the functionality and code design of each of the individual layers that make up the Piconet protocol stack.

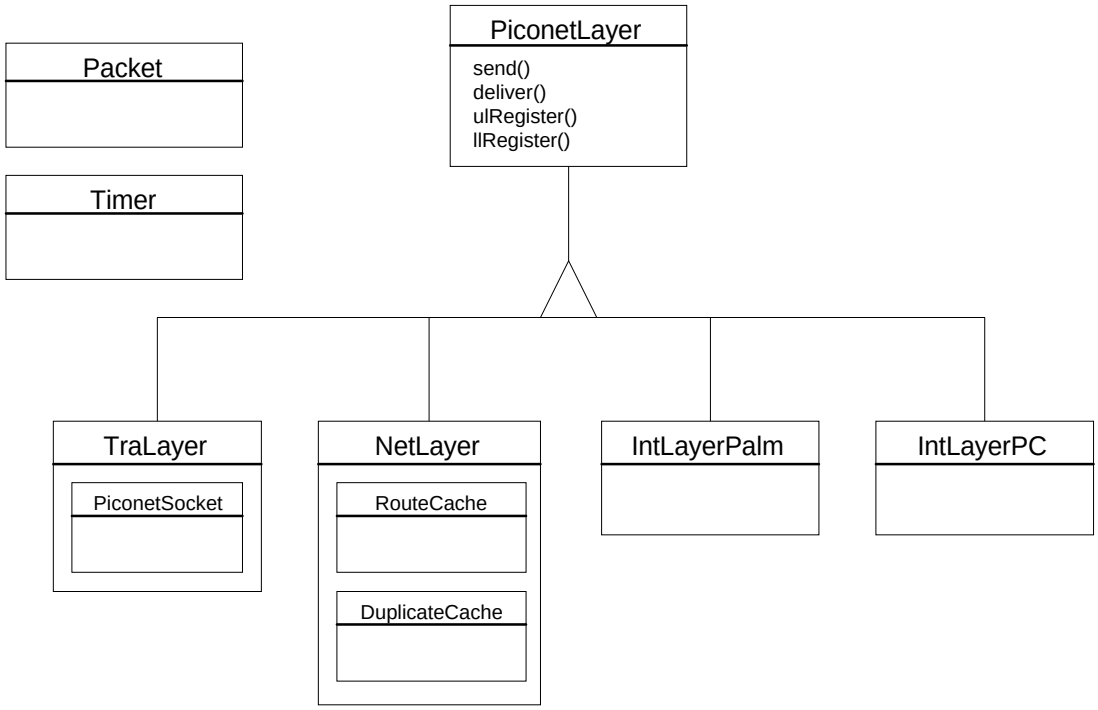


Figure 21 Java class hierarchy in the Piconet protocol stack

Figure 21 shows the class hierarchy of the protocol stack, together with significant utility classes and data structures such as the Packet and Timer classes, which are used throughout the stack, and the Network Layer's route and duplicate caches. Code listings for a selection of these files can be found in Appendix B.

5.4.1 Interface Layer

The Interface Layer performs framing and un-framing of Network Layer packets as described earlier, and sends and receives data to and from the Palm's serial port. OS-specific functions are used to periodically poll the serial port using a daemon thread. An interrupt function triggered upon receipt of data on the serial port would be more

efficient; unfortunately, Palm OS has no such facility (and resultantly nor does the J9 JVM). The serial port API blocks unless a timeout is specified. This timeout value is currently set at 10 ms to maintain program responsiveness.

When a valid packet is received, it is shuffled up the stack via a call to the Network Layer's `deliver()` method, which in turn calls the Transport Layer's `deliver()` method, and so on.

The Interface Layer is the only platform-specific code in the protocol stack. As a demonstration of the advantages of writing in Java, replacing the Palm-specific serial port functions with their PC equivalents results in a PC-based version of the ad hoc protocol stack.

5.4.2 Network Layer

The Network Layer provides a best-effort delivery service on a per-packet basis between nodes in the Piconet system. The routing protocol described in Section 5.3 is implemented at this layer.

The format of the Network Layer PDU is quite simple, as can be seen in Figure 22.

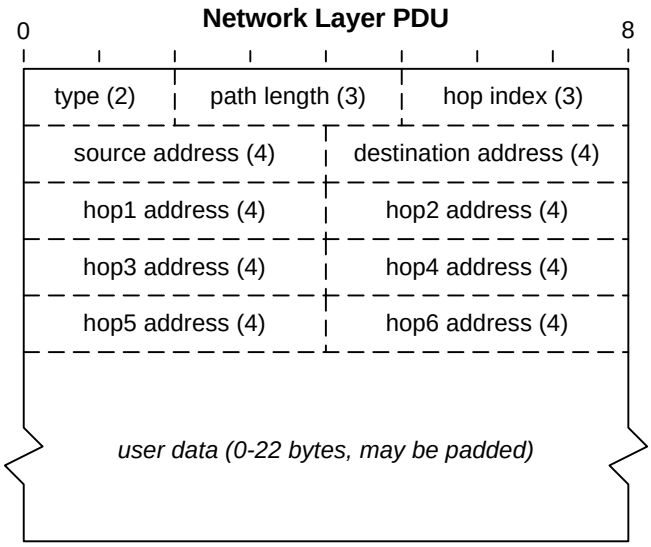


Figure 22 Format of the Network Layer PDU

The type specifier is two bits long to distinguish amongst the four packet types required by DSR, namely route requests (RREQ), route replies (RREP), route errors (RERR) and data (DATA).

The path length field indicates the length in hops of the route record. In the current implementation, 3 bits are provided for this purpose, limiting the number of hops to 7.

Although the header size is fixed at 5 bytes (for containing the 8 possible path addresses), this field could be used in the future to accommodate varying header lengths.

The hop index is used as described previously as an index into the route record of the network address of the next hop to which the packet should be forwarded.

Following this are up to 8 network addresses, of which the first two are always the end-to-end source and destination addresses of the packet. Each network address is 4 bits long, allowing up to 15 distinct nodes to be present in the network as described in the requirements. One address (“1111”) is reserved for broadcast purposes.

Larger networks and more hops can be accommodated at the expense of payload size. The tradeoff was made in view of the somewhat limited 27-byte packet size of the wireless module. Currently, a maximum of 22 bytes is available for user data.

Header fields conspicuous by their absence include checksum and length fields. The former is not required here since RPC frames must pass a checksum before arriving at the Interface Layer; this ensures that the data passed up by the Interface Layer (i.e. a Network Layer PDU) will be valid. The latter is also not required, since packets are assumed to be of uniform length, and are padded if necessary. These fields have been omitted and the responsibility of further error checking and message length determination is passed on to higher layers. The tradeoff occurs in minimising the header size against the wasted bandwidth of transmitting redundant pad bytes. If one assumes that the majority of packets will have a full payload (a not unreasonable assumption given the space available in each packet for user data), this wastage is minimal. It may be useful to introduce such fields if the link hardware changes and frame sizes increase.

The format of the RREQ and RREP packets used in the route discovery/route maintenance process is similar to that used for carrying data. The path length field is used to determine where in the route record the forwarding node should append its own address; the forwarding node then increments the value of the path length field. The route record is also inspected for prior entries of its own address – this enforces the loop freedom property. The hop index field is unused.

A route error packet only uses the packet type, source address and destination address fields. Hence, only the source and destination addresses of the invalid route are

identified, not the broken link itself. This is not of consequence since the RERR packet should trigger a new route discovery process anyway.

The Network Layer maintains a number of data structures:

- A duplicate RREQ, RREP and RERR cache. This is necessary by virtue of the manner in which such packets are broadcast across the network. Each node forwards the first instance of each type of packet it receives. Duplicate filtering is achieved by entering the time the first instance was processed into a hashtable keyed on the `<packet type, source address, destination address>` tuple of the packet. This hashtable is searched upon receipt of any one of the RREQ, RREP and RERR packet types; if the difference between the resulting entry (if there is one) and the current time does not exceed some preset threshold, the packet is dropped.
- A route cache of learned routes. This is simply an array of route records indexed by destination address. Although multiple routes are possible in DSR, it is optional and not supported in this implementation.

The Network Layer is stateless; packets to be sent and received are handled purely based on its type: DATA, RREQ, RREP or RERR. Shared information is gleaned from the duplicate and route caches; these enforce mutual exclusion via Java's `synchronize` keyword to ensure concurrent modification does not occur (the Interface Layer executes as a thread, which means packets may be delivered to the Network Layer for processing at any time, resulting in concurrency issues).

When transmitting packets, the route cache is consulted for a sufficiently fresh route. If one is found, it is copied into the route record area and transmitted. A timer is initialised for passive acknowledgement purposes; if the timer expires before a passive acknowledge is received, a RERR packet is generated and flooded back to the source of the data packet.

If the route cache does not contain a valid route, the data packet is dropped and a route discovery process (RREQ) is initiated. It is the responsibility of the upper layers to retransmit the dropped packet.

Piconet differs from DSR in these respects. DSR specifies that send and retransmit buffers be maintained. The former buffers packets for which a route cannot be found

in the route cache; these are transmitted when the route discovery process returns with a result. Similarly, the latter buffers and retransmits packets that have not been passively acknowledged. It was felt maintaining such buffers would be too costly in terms of resources, and responsibility for retransmission left to an upper layer.

When receiving a data packet, the route record is immediately examined. If the destination address matches the receiving node's address or the broadcast address, the packet is `deliver()`ed to the Transport Layer. Otherwise, the hop index field is used as described in Section 5.3.2 to determine whether the packet needs to be forwarded, and if an unsolicited route reply should be sent back to the source to indicate a shorter path has been found. A timer to wait for passive acknowledgement is also initialised in the same manner as described above.

The semantics of the hop index field is different in the case of forwarding packets whose destination is the broadcast address. In all cases, the hop index is incremented prior to forwarding. However, rather than acting as an index into the route record, here it is used to limit the number of times the packet is rebroadcast by neighbour nodes and prevent broadcast packets endlessly circulating the network.

Received RREQ, RREP and RERR packets are forwarded as described previously. The duplicate cache is consulted and the packet forwarded if this node has not already processed it recently. A form of promiscuous listening is used, such that intermediate nodes whose addresses are contained in the route reply will use the route record and insert it into their own route caches. This saves them from having to initiate route discovery processes for routes they already know about.

5.4.3 Transport Layer

The Transport Layer provides a reliable, pseudo-connection-oriented service for applications programmers. It is based on the Berkeley sockets style of API [38] which gives the impression that a network connection is actually a local stream of data.

The Transport Layer creates a PiconetSocket object for each connection request by an application. The application must specify the address of the remote node, and is given a reference to the socket in response. Unlike in TCP/IP, there is no concept of ports – there is only one application per `<source address, destination address>` pair, although one device may have more than one connection open (each with different nodes).

The PiconetSocket object implements reliability by buffering and retransmitting packets using a sliding window protocol and the Karn/Partridge algorithm for calculating round-trip times and retransmission time-outs [39]. Sliding window is preferred over simpler acknowledgement schemes as stop-and-wait

The Transport Layer acts as a simple multiplexer, diverting packets delivered to it by the Network Layer to the appropriate PiconetSocket object associated with the source of the packet. The PiconetSocket object then buffers the packet for the application, performing sequencing and reassembly as required.

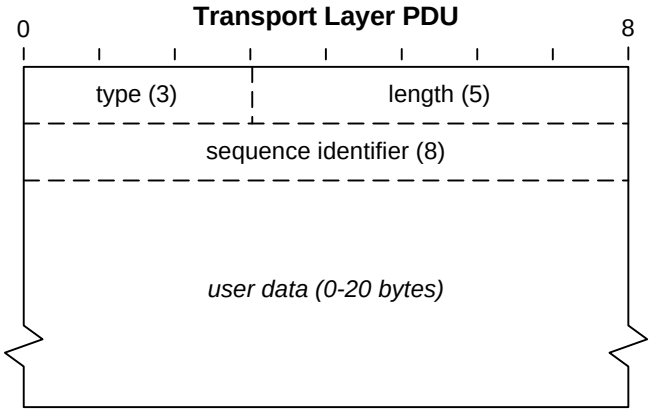


Figure 23 Format of the Transport Layer PDU

The basic Transport Layer PDU (shown in Figure 23) has type, length and sequence identifier fields. The type specifies whether the PDU is for connection, data or acknowledgement purposes; the length field indicates the size of the user data area in bytes, while the sequence identifier is used in the sliding window protocol.

The Transport Layer also provides a form of name/service resolution. This is achieved by using the broadcast address to propagate name or service requests across the entire network. The type of request is contained within the broadcast message (whether it is a name that is sought, or a particular service), and the reply is an echo of the request, but with the address of the node offering the service. Although the facilities for this feature are present, time constraints prevented name/service resolution to be tested.

A more detailed description of the workings of the Transport Layer can be found in Mr Israel Keys' thesis document [29].

6 System Performance

This section outlines the performance of the wireless module and network protocols. Reference is made to the original specification of the Piconet system of Section 3.

This section concludes with a personal evaluation that reflects on the design and development process, and how it could have been improved to make a better product.

6.1 Wireless Module Performance

6.1.1 Physical Characteristics

The wireless module was initially developed using breadboard techniques before moving to a proper PCB. For the Piconet prototype, the RPC transceiver is mounted vertically on a 10-pin header on the PCB. Batteries are held in an external plastic case that could in future be held flush against the RPC. A custom HotSync connector from ATL Technology [40], a company specialising in cables and connectors, links the wireless module to the Visor's serial port.

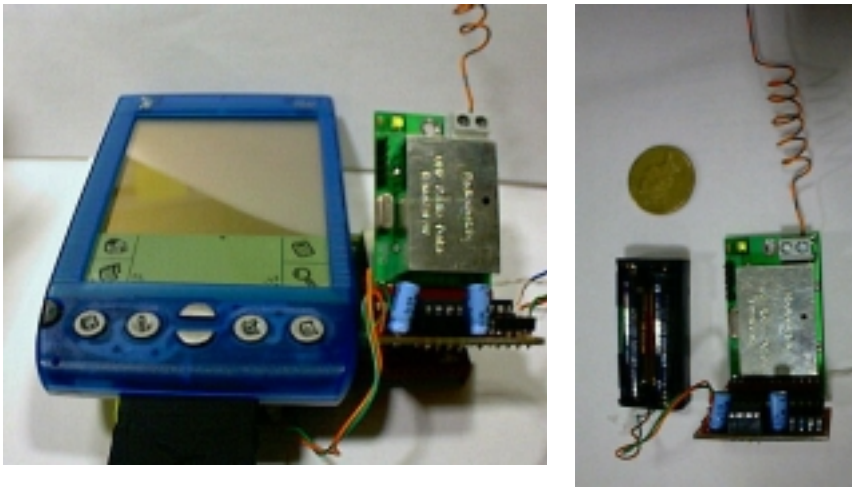


Figure 24 Photos of the Visor with the prototype wireless module (left) and individually for size comparison purposes (right)

The photos of Figure 24 give an idea of the size of the module and how it is attached. A shipping product would have a more elegant means of attaching itself to the Palm, most likely using a custom-designed sleeve that wraps around the bottom of the case and encloses the serial port pins.

power consumption	20mA (less in sleep mode)
battery life	32hrs continuous operation*
weight	50g (excl. batteries)
dimensions	75mm x 40mm x 36mm (LxWxH)

* based on 82% conversion efficiency from 2x AAA alkaline battery cells

Table 5 Physical characteristics of the wireless module

Table 5 certainly demonstrates that the wireless module fulfils the specifications for power consumption, battery life and portability. Under "real world" conditions, it is likely that the battery life could extend to weeks depending upon usage. Additional power saving techniques, such as placing the wireless transceiver into sleep mode when the channel is inactive [21], could improve this figure even further.

6.1.2 Data Link Performance

	range (m)	data rate (bps)
open air	50	3200
closed environment	8	3200

Table 6 Range and data rate of the wireless module

A number of raw performance measures were made on the wireless module. A data rate of approximately 3.2kbps can be achieved between two nodes barring the presence of interference effects. At distances beyond those listed above, packets begin to be corrupted; the RPC filters these and hence are not processed by the module firmware. This explains the recurring data rate figure regardless of environment.

Antenna choice has an impact on the performance of the link. As described previously, a simple helical loop antenna was used (simply a piece of wire wound on a 3.2mm diameter former). A whip antenna appropriately matched to the antenna port would increase the range at the expense of portability.

Currently, the serial link between the wireless module and the Palm represents a bottleneck in the system. Connection speed is limited to 9600bps. At higher data rates, the buffer over-runs because the PIC is too busy servicing the RPC to empty the transmit buffer filled by the UART, causing packets to be lost. Using some form of software flow control mechanism between the Palm and the wireless module may rectify this. There is insufficient RAM on the PIC to enlarge the buffer significantly without resorting to external memory. A flow control protocol between the Palm and

the module was implemented, but without success. It was felt that 9600bps was adequate for our purposes, and further work was stopped to devote time to Network Protocols development.

6.2 Network Protocols Performance

6.2.1 Physical Network Characteristics

Network size	15 nodes
Network span	7 hops
Link range	8m closed, 50m open
Maximum network range (end-to-end)	750m*

* based on 7 hops in an open environment

Table 7 Physical characteristics of the network

Physical network characteristics (Table 7) are a result of the conscious design decision of only allowing 15 nodes for the prototype (easily extensible by increasing the bit length of network addresses), and the combination of multi-hopping with the link radius of the wireless module. Thus, the use of nodes as routers extends the range of Piconet to a theoretical limit of 750 metres. This could not be tested due to a lack of available node hardware, but limited multi-hopping tests have not revealed any obstacle to this potential being realised.

6.2.2 Network Performance Measures

Stack delay (top to bottom)	10- ms
Stack delay (bottom to top)	10- ms
Transport Layer percentage header	9%
Network Layer percentage header	18.5%
Nearest neighbour round trip time	~500 ms
Route acquisition time	500+ ms
Control bytes transmitted/Total bytes transmitted*	19.8%

* typical 512-byte data transfer requiring 24 packets and accounting for route acquisition

Table 8 Quantitative protocol performance measures

Table 8 provides a quantitative view of the performance of Piconet. A few comments need to be made regarding these figures.

The first two entries of the table measure stack overhead; this is useful to understand the penalty that the protocol implementation incurs disregarding the speed of the

wireless link. At the maximum data link speed of 3200bps quoted for the wireless module, this means that the protocol stack introduces a 4-byte delay per packet, or roughly 15% of the length of a packet.

Protocol Data Units each impose an overhead and restrict the number of bytes that can be transmitted with each packet. At the Network Layer, the header consumes 5 bytes out of the total 27, or just over 18%. The Transport Layer consumes another 2 bytes, leaving approximately 75% for user data. This is quite low, but fragmenting larger-sized packets into multiple RPC frames to improve these ratios is not practical (as might occur in a wired IP network).

Route acquisition time in conjunction with nearest neighbour round trip time demonstrate how well Piconet implements the throughput and latency specification of Section 3.3. These two figures combine to measure end-to-end throughput, which is hampered by time to acquire a route if it is not already in the route cache, and the time it takes for each link to be traversed in the route.

Route acquisition time refers to the latency imposed by the on-demand route discovery process. The quoted figure of 500 milliseconds is an average based on the small scale testing performed for this thesis. For larger network diameters, this number will only increase. Unfortunately, this large latency is the main disadvantage displayed by all on-demand, source-initiated routing protocols. Once a route is acquired, however, there is no performance penalty on subsequent sends, and route maintenance processes can help to reduce the number of route requests by learning new routes as packets are forwarded (e.g. DSR's "promiscuous listening" mode).

The nearest neighbour round trip time of 500ms needs to be considered on top of the effect of the Java VM, the Palm OS, and the lack of reliable time slicing, and the fact that the nearest neighbour could be busy processing other packets as well as avoiding collisions – the round trip time can be highly variable.

It is worthwhile to also take into account the number of control bits transmitted versus total bits transmitted. The table indicates that close to 20% of the bandwidth used by a sending node when transferring 512 bytes is taken up with control bits (RREQ, RREP and route records in data packet headers). This figure could improve if the native frame size of the RPC was larger than just 27 bytes.

Clearly, the biggest restriction in the performance of the protocols lies with the Data Link Layer and the small frame size that can be transmitted wirelessly – suggesting an alternative to the RPC would be worthwhile investigating.

It is also worth noting that a number of variables at the Network and Transport Layers affect the performance of the network. These are listed in table, along with their current settings in Piconet.

DUPLICATECACHE_THRESHOLD	2 seconds
ROUTECACHE_THRESHOLD	30 seconds
PASVACK_THRESHOLD	5 seconds
MAX_REBROADCASTS	8
RESEND_TIMEOUT	250 ms*
MAX_RESENDS	60

* init. value only, changes dynamically

Table 9 System variables and their settings

The duplicate cache threshold (DUPLICATECACHE_THRESHOLD) gives the minimum time allowed between successive forwards of RREQ, RREP or RERR packets for a particular `<source, destination>` address pair. As its name implies, this enables duplicate packet detection resulting from nearest neighbour retransmission.

The route cache expiry time (ROUTECACHE_THRESHOLD) governs how long a route may remain unused in the cache; if the route has expired, the next time a request is made to send a packet to that destination, a new route discovery process must be initiated, causing a delay while the route is acquired. This time is currently set at 30 seconds, a somewhat arbitrary figure that estimates the network configuration to be quite dynamic. Extending this would be beneficial when it is known that the network is fairly stable.

Another variable is the time permitted between a transmission and its passive acknowledgement (PASVACK_THRESHOLD). This figure was honed through an iterative process to account for the delay caused by traversing up and down the protocol stack on the local and remote nodes, as well as the speed of the wireless link. If the time allowed for a passive acknowledgement was too short, erroneous route errors would be signalled.

Testing also demonstrated how to optimise the length of the time-out between retransmissions at the Transport Layer (RESEND_TIMEOUT). Leaving it as a fixed

value impacts on the performance of the network by performing premature retransmissions. A more sophisticated algorithm was required to gauge the round trip time of the connection and hence allow time for the end-to-end acknowledgement to arrive.

The values of these variables are not fixed – indeed, for optimum performance they should be tuned to the prevailing conditions. This could be the source of future work.

6.2.3 Ad Hoc Capabilities

A limited number of nodes was available for testing purposes. As a result, only 2- and 3-node configurations could be tested. Some of these configurations (including transitions to demonstrate the routing protocol's ability to adapt to node mobility) are shown in Figure 25. In each case, the protocol was found to be effective in discovering routes, and maintaining a connection even in the face of positional changes.

Node movement speed was not of concern here. A traditional interested party of ad hoc networking, the military, pays particular attention to this parameter since likely users would include potentially fast moving objects such as tanks, trucks, and mobile infantry [4]. For our target application area – home and office users – the detection and repair requirements are less stringent.

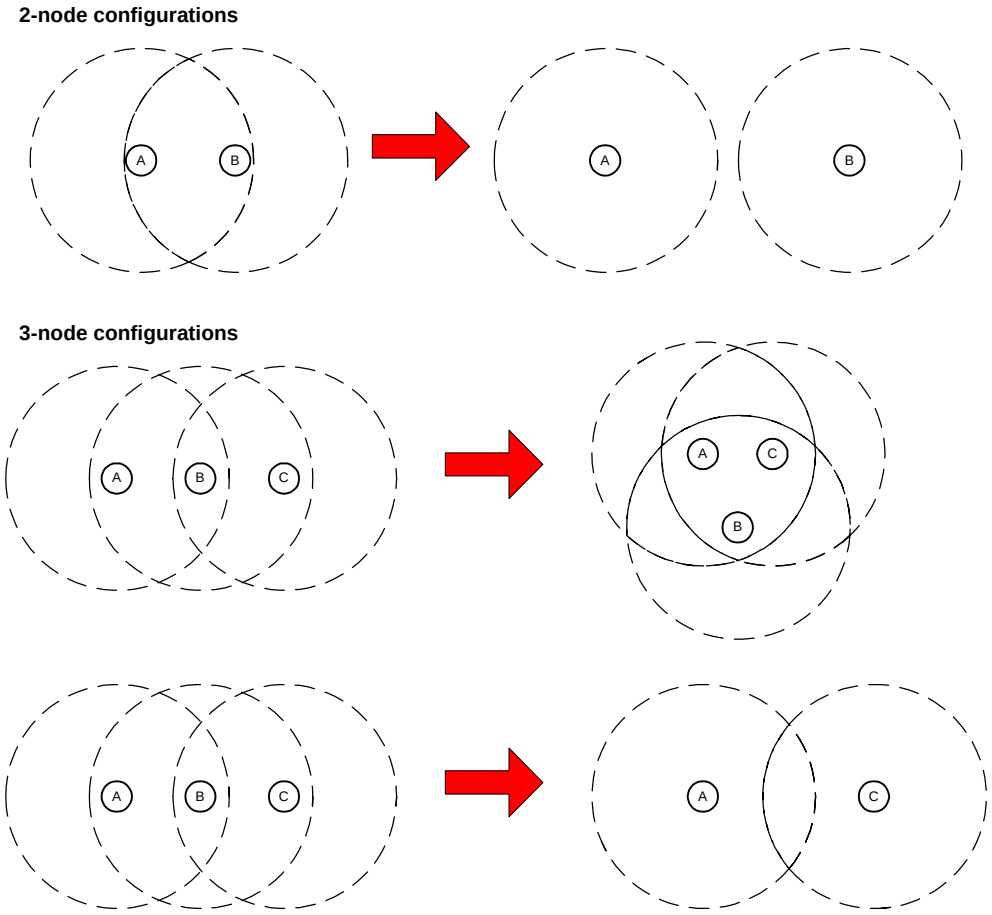


Figure 25 Some of the tested network configurations and transitions

Beyond 3-node configurations, a simple simulator was introduced to verify the Network Layer's correct operation. This simulator is described below.

6.2.3.1 Simulator Description

The simulator was written in Java, based on a client/server model. The Interface Layer of the stack was replaced with a TCP client. A TCP server replaced the Data Link and Physical Layers, and acted as the arbitrator of a virtual broadcast channel. Network nodes (the clients) register with the server, which maintains a record of which nodes are present in the network, and which nodes' wireless link radii overlap.

All inter-nodal communication occurs via the server. It is the server's responsibility to forward data to the appropriate nodes (i.e. the nodes that are "in range").

Discrete event simulators are available that are targeted at networking research. The Network Simulator, or ns, supports simulation of TCP, routing and multicast protocols over both wired and wireless networks [41]. However, for our purposes, the Java-based simulator was sufficient. A significant barrier to using ns was that the

protocols and event scripting would need to be re-written in the scripting language, Tcl.

6.2.3.2 Results from the Simulator

As in real world testing, the simulator verified the correct operation of the ad hoc networking protocol. Multiple nodes (i.e. more than 3) were tested in various configurations. The system was able to discover and maintain routes robustly, although obviously the characteristics of a wireless medium cannot be fully captured by the simulator's rather primitive broadcast model – issues such as medium access control, packet collision, data loss and varying node mobility are difficult to simulate effectively. System variables such as retransmission time-outs, and duplicate and route cache expiry times, must be tuned to the environment, and hence further testing with multiple nodes would be necessary in real world deployment.

6.2.4 Proof-of-Concept Applications

A simple messaging application demonstrated the feasibility of the Piconet system. Two Palm PDAs in conjunction with a notebook computer formed an ad hoc network to support the application. One node performed solely in the routing function, leaving the remaining two nodes in a connected chat session.

Figure 26 A screen dump of the Java-based chat application running on Piconet

The user specifies the node's local address as well as the address of the remote destination. The Piconet stack is then initialised, and a PiconetSocket is instantiated to allow reliable communication between the two devices. The user then enters a string

into the text field, and transmits the data to the remote device for display by pressing the "!" button widget. In essence, the application is a simplified form of IRC[9]. This application was also used as the testbed from which some of the measurements of Table 8 were made.

6.3 Comparison with Specifications

The specifications for the wireless module were met. It consumes relatively little current and meets the battery life specification, with room for further power improvements if sleep mode was used (time-to-wakeup was one concern which discouraged its use, however).

The range is also satisfactory, with 8 metres in a closed environment a reasonable figure – although performance will obviously vary according to the conditions.

Portability is a minor concern. The use of the HotSync connector to attach the wireless module to the Palm is not a production-quality feature, and it is envisioned that further development aesthetically would be needed.

The Network Protocols largely implemented what was required of them in the qualitative sense – Piconet can discover routes and use them for packet forwarding, and deliver data in a reliable fashion – but overall performance is sluggish as a result of the overhead incurred at each layer. In some ways, this is a hardware problem (e.g. the limited speed of the Palm's serial port, the relatively slow Dragonball CPU), while in others it is a software problem (interpreted Java, the effect of system variables for time-outs etc.). Removing some of these bottlenecks is the subject of Future Developments.

Unfortunately, the name/service resolution feature of the Transport Layer could not be satisfactorily tested. However, the underlying structures for such a facility are present in the code.

6.4 Personal Evaluation

Many lessons were learned in designing and developing the Piconet system.

[9] Internet Relay Chat

Ambition is an admirable quality to possess, but in retrospect, the Piconet concept and our estimations of the timeframe required to realise it were overly optimistic.

On the hardware side, developing an external wireless module that could be battery-powered and interface robustly via an RS232 connection had its own difficulties. While we had experience with the Motorola 68HC11, developing for the PIC had a steeper learning curve than expected. In general, it is a simple device to use, but a few quirks that are only really learned by experience delayed our plans somewhat.

On the software side, we were faced with the resource constraints of the Palm device – in terms of processor speed, operating system and general ease of programming. The Palm OS was never designed for complex processing tasks, and effectively turning it into a router was always going to be difficult. These constraints were compounded by our having to grasp the theoretical aspects of computer networking, the difficulties of ad hoc routing, and how we could solve these problems and simultaneously implement an efficient networking system within these constraints.

Overall, the experience has proved valuable beyond belief. An important lesson was how each component of the system can affect the others – design of each cannot be done entirely in isolation. For example, the choice of the RPC transceiver determined what kind of microcontroller would be used in the wireless module, and restricted the Network Layer PDU size to the 27-byte limit of the frames that the RPC accepts. This was one of the most difficult aspects of the development work: making design decisions with system-wide ramifications must be done with plenty of consultation, even over seemingly minor design details.

Personally, I am satisfied with what we have achieved in Piconet – a working prototype – given the timeframe and the numerous obstacles we faced.

Besides being personally enriched by my thesis partnership, I believe the most valuable skill I have gained is an understanding of system design: being able to break down a system into components, how to design each component to minimally affect the others, but at the same time optimise the interfaces to maximise performance.

7 Future Developments

This thesis has documented how Piconet implemented a wireless ad hoc network for mobile devices. It has verified Dynamic Source Routing as a viable ad hoc routing protocol. This chapter outlines improvements and possible extensions to the Piconet system.

7.1 Wireless Module

A severe limitation is the Handspring Visor's serial port. Presently, the wireless module communicates at a maximum rate of 9600bps; any attempt at going higher than this results in buffer overruns in the microcontroller. The RPC is capable of transmitting at up to 40kbps, so a sizeable proportion of its capability is not being exploited. The Visor's Springboard slot may be an option. This proprietary expansion interface uses the same 68-pin header as PCMCIA[10] cards (although the electrical specification differs). As an expansion of the CPU bus and its support for plug-and-play, Springboard modules have great potential. However, this is at the risk of tying the wireless module to a proprietary but freely circulated standard that has yet to reach wide acceptance in industry. This could, however, result in faster data transfers since the bottleneck is the serial port interface, which is currently limited to just 9600bps.

Some form of medium access control would also be desirable. The RPC transceiver may not be the best for this purpose as it does not allow such fine-grained control of transmission parameters.

The RPC itself has proved problematic – the small frame size does not lend itself well to designing an efficient network protocol, particularly using the DSR routing scheme which, while robust, already has a relatively high overhead in header bytes compared to other schemes.

[10] Personal Computer Memory Card International Association – an industry standards body defining an I/O interface for Integrated Circuit cards for mobile devices

7.2 Network Protocols

The system variables regarding time-outs and the like are some of the most problematic in setting up Piconet. Unless these are adjusted, there is potential for useless packet transmissions and congestion. A proposed future development would be to incorporate a learning algorithm that adapts these timeouts to prevailing network conditions, increasing and decreasing them as necessary to maximise channel efficiency and utilisation.

One instant way of improving performance would be to port the Java code to C. However, doing so for the Palm is difficult beyond the time penalty incurred in moving from an object-oriented language (Java) to a procedurally-oriented one (i.e. C). As mentioned previously, the Palm OS is the crippling factor here, together with its relatively slow Motorola Dragonball CPU.

Using a Palm rather than a Handspring Visor could allow experimentation with uCLinux, which may yield better performance.

One extreme would be to move the entire ad hoc stack into hardware (or more likely to execute on a separate, dedicated microprocessor) and bypass the Palm CPU and OS altogether. This is entirely possible, given the number of companies who are releasing or have released microprocessors that can natively execute Java bytecodes. This would make porting the present stack not particularly difficult. Combined with the Springboard slot option mentioned above, this has the greatest potential for performance improvement.

From an applications perspective, TCP/IP integration is a possibility. In much the same way that Bluetooth can function as the Link Layer protocol in a TCP/IP stack, Piconet could be used in this capacity (although this would need to be combined with the improvements listed above to be truly practical). Theoretically, an alternative approach is to tunnel Internet Application Layer message streams using the Piconet transport protocol, but this would be ambitious and the overhead significant. It would remain to be seen whether Piconet could cope with sustained data transfers.

8 Conclusion

Piconet was designed to be a lightweight, wireless ad hoc network system for mobile handheld devices. Such a system's virtues include being able to establish networked connections in the absence of fixed infrastructure, encouraging communications regardless of locale.

This thesis has demonstrated that by using the source-initiated routing protocol, DSR, it is possible to create a network for mobile devices that can adapt to dynamic reconfiguration – a typical scenario amongst the target market of office or home users as they move from room to room and office to office.

Although there are many challenges in designing for handheld devices, the current state of Piconet presents a promising step towards a robust ad hoc networking system that embraces the concept of mobile computing.

References

[1] Handspring, *Welcome to Handspring*, http://www.handspring.com/, October 2000.

[2] Palm OS, *Welcome to Palm OS*, http://www.palmos.com/, October 2000.

[3] Internet RFC 2501, Network Working Group, S. Corson, J. Macker, *Mobile Ad Hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations*, January 1999. Available from http://www.cis.ohio-state.edu/htbin/rfc/rfc2501.html.

[4] Z. Haas and S. Tabrizi, On Some Challenges and Design Choices in Ad-hoc Communications, *IEEE Milcom '98,* Bedford, MA, October 18-21, 1998.

[5] Larry L. Peterson and Bruce S. Davie, Computer Networks, A Systems Approach, 2nd ed. San Francisco, Morgan Kaufmann, 2000.

[6] IETF MANET Working Group, Internet-Draft, J. Broch, D. Johnson, et al., *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks*, 22 October 1999. Available from http://www.monarch.cs.cmu.edu/ietf.html, October 2000.

[7] P. Karn, MACA – A New Channel Access Method for Packet Radio, *ARRL/CRRL Amateur Radio 9th Computer Networking Conference*, pp. 134-140, ARRL, 1990.

[8] The Wireless LAN Association, *The IEEE 802.11 Wireless LAN Standard*, http://www.wlana.com/intro/standard/index.html, October 2000.

[9] V. Bharghavan, et al., *MACAW: a media access protocol for wireless LAN's*, Proceedings of the conference on Communications architectures, protocols and applications, pp.212-225, 1994.

[10] S. Keshav, *An Engineering Approach to Computer Networking: ATM networks, the Internet, and the Telephone Network*. Reading, Mass. Addison-Wesley, 1997.

[11] R. Maatta, *Wireless Ad Hoc Routing Protocols, a Taxonomy*, http://personal.eunet.fi/pp/sahkotl/seminar/adhoc.htm, September 2000.

[12] Carnegie Mellon University, *Wireless Andrew*, http://www.cmu.edu/computing/wireless/index.html, October 2000.

[13] ETSI, *ETSI HiperLAN/1 Standard*, http://www.etsi.org/technicalactiv/hiperlan1.htm, October 2000.

[14] HomeRF, *Technical & Spec. Info*, http://www.homerf.org/tech/index.html, October 2000.

[15] Bluetooth SIG, *The Bluetooth Specification, Vol. 1, Core*, available in PDF format at http://www.bluetooth.com/developer/specification/core_10_b.pdf, October 2000.

[16] Bluetooth SIG, *The Bluetooth Specification, Vol. 2, Profiles*, available in PDF format at http://www.bluetooth.com/developer/specification/profile_10_b.pdf, October 2000.

[17] Xircom, *Handheld Access Solutions*, http://www.xircom.com/cda/page/1,1298,0-0-1_1-704,00.html, October 2000.

[18] WIDCOMM, *BlueConnect Product Summary Notes*, available in PDF format at http://www.widcomm.com/pdfs/BlueConnect.pdf, October 2000.

[19] Apple, *Airport Wireless Technology*, http://www.apple.com/airport/, October 2000.

[20] Australian Communications Authority, *The Australian Radiofrequency Spectrum Plan*, http://www.austel.gov.au/frequency/spectrum.htm, October 2000.

[21] Radiometrix, *Radio Packet Controller Datasheet*, available in PDF format from http://www.radiometrix.co.uk/dsheets/rpc.pdf, October 2000.

[22] Radiometrix, *BiM Datasheet*, available in PDF format from http://www.radiometrix.co.uk/dsheets/bim.pdf, October 2000.

[23] Linx Technologies, *Linx SC Datasheet (Preliminary)*, available in PDF format from http://www.linxtechnologies.com/lpdfs/scdata.pdf, October 2000.

[24] RF Monolithics, *RFM DR3000 Datasheet*, available in PDF format from http://www.rfm.com/products/data/dr3000.pdf, October 2000.

[25] Motorola Semiconductor Products Sector, *Motorola 68HC11 E-series Databook*, available in PDF format from http://ebus.motorola.com/brdata/PDFDB/MICROCONTROLLERS/8-BIT/68HC11_FAMILY/DATABOOK/M68HC11E.pdf, October 2000.

[26] Microchip Technology Inc, PIC16F873 Datasheet, available in PDF format at http://www.microchip.com/download/lit/pline/picmicro/families/16f87x/datasheet/30292b.pdf, October 2000.

[27] Atmel, AVR 90S2313 Datasheet, available in PDF format at http://www.atmel.com/atmel/acrobat/0839s.pdf, October 2000.

[28] Maxim Integrated Products, MAX619 Datasheet, available in PDF format from http://pdfserv.maxim-ic.com/arpdf/1917.pdf, October 2000.

[29] I. Keys, *Piconet: A Wireless Ad Hoc Network for Mobile Handheld Devices*, Bachelor Thesis, University of Queensland, St Lucia, School of Computer Science and Electrical Engineering, 2000.

[30] Palm, *Palm OS Software Documentation*, available in PDF format at http://www.palmos.com/dev/tech/docs/palmos35docs.zip, October 2000.

[31] Embedded Linux/Microcontroller Project, http://www.uclinux.org/, October 2000.

[32] Handspring, *Springboard Development Guide for Handspring Handheld Computers*, available in PDF format from http://www.handspring.com/developers/Devkit2/Springboard_Development_Guide.pdf, October 2000.

[33] Sun Microsystems, *The Source for Java Technology*, http://java.sun.com/, October 2000.

[34] IBM VisualAge Micro Edition, http://www.embedded.oti.com/, October 2000.

[35] E. Royer and C. Toh, A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks, *IEEE Personal Communications*, pp. 46-55, April 1999.

[36] D. Johnson and D. Maltz, *Protocols for Adaptive Wireless and Mobile Networking*, IEEE Personal Communications, Vol 3, No. 1, pp. 34-42, February 1996.

[37] Josh Broch, David A. Maltz et al., *A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols*, Proc. of the ACM/IEEE MobiCom, Oct. 1998.

[38] W. Stevens, *UNIX Network Programming*, New Jersey, Prentice Hall, 1990.

[39] D. Comer, D. Stevens, *Internetworking with TCP/IP Volume II, Design, Implementation, and Internals, 2nd ed.*, New Jersey, Prentice Hall, 1994

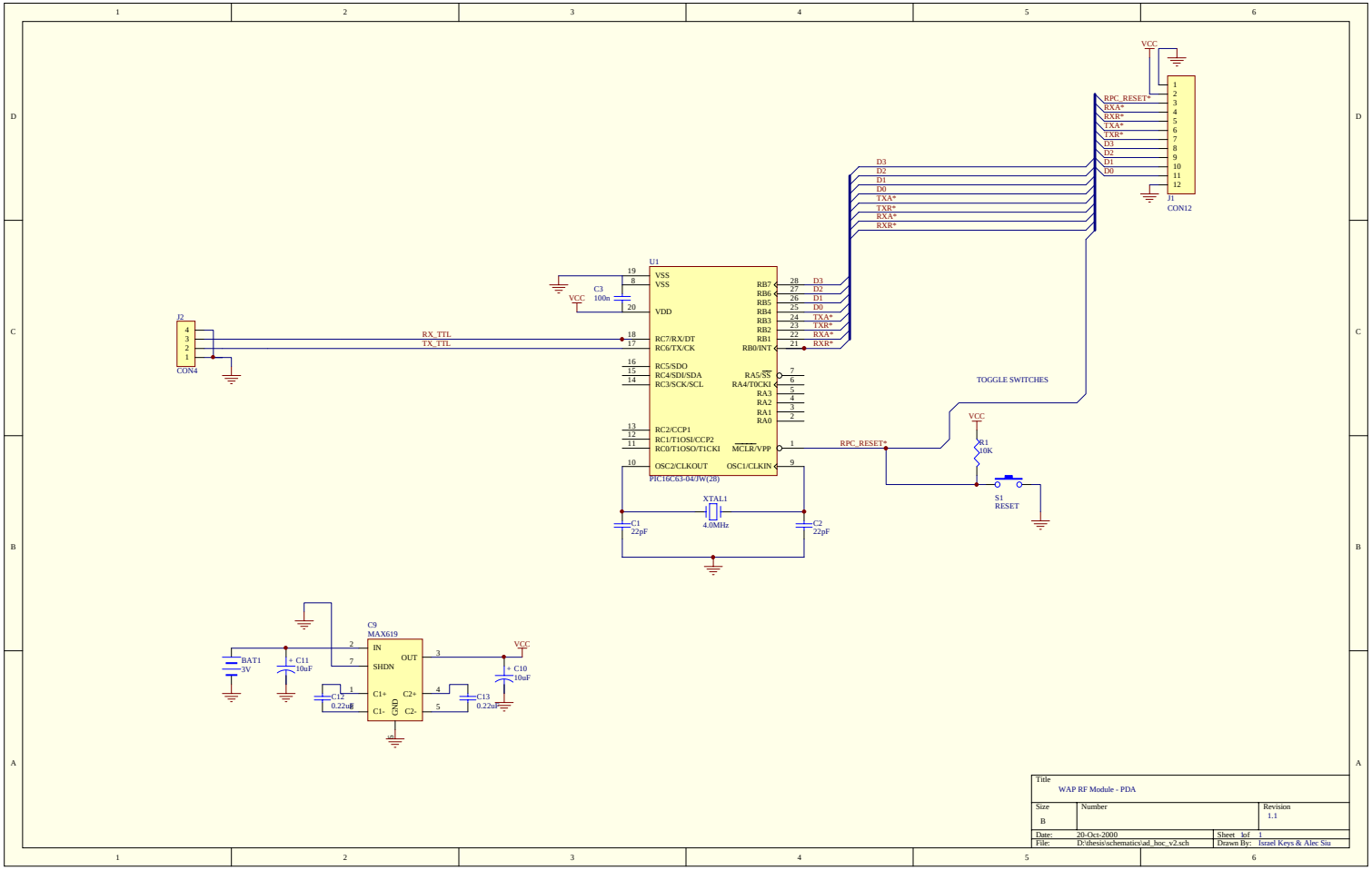
[40] ATL Technology, *Handspring Third Party Developer Support*, http://www.atlconnect.com/handspringsource.html, October 2000.

[41] *The Network Simulator – ns-2*, http://www.isi.edu/nsnam/ns/.

[42] IETF MANET Working Group, Internet-Draft, J. Broch, D. Johnson, et al., *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks*, available from http://www.ietf.org/proceedings/99mar/I-D/draft-ietf-manet-dsr-01.txt, October 2000.

Appendix A: Schematics

The following schematic is for the wireless module (ad_hoc_v2.sch).


VCC
RPC_RESET*
RXA*
RXR*
TXA*
TXR*
D3
D2
D1
D0
J1
CON12
U1
VSS
VSS
VDD
RC7/RX/DT
RC6/TX/CK
RC5/SDO
RC4/SDI/SDA
RC3/SCK/SCL
RC2/CCP1
RC1/T1OSI/CCP2
RC0/T1OSO/T1CKI
OSC2/CLKOUT
RB7
RB6
RB5
RB4
RB3
RB2
RB1
RB0/INT
RA5/SS
RA4/T0CKI
RA3
RA2
RA1
RA0
MCLR/VPP
OSC1/CLKIN
PIC16C63-04/JW(28)
C3
100n
J2
CON4
RX_TTL
TX_TTL
TOGGLE SWITCHES
R1
10K
S1
RESET
XTAL1
4.0MHz
C1
22pF
C2
22pF
C9
MAX619
IN
SHDN
OUT
C1+
C1-
C2+
C2-
GND
BAT1
3V
C11
10uF
C12
0.22uF
C13
0.22uF
C10
10uF
Title
WAP RF Module - PDA
Size
B
Number
Revision
1.1
Date: 20-Oct-2000
Sheet lof 1
File: D:\thesis\schematics\ad_hoc_v2.sch
Drawn By: Israel Keys & Alec Siu

Appendix B: Source Code Listings

Wireless module firmware C source code is provided for the following files only (full listings for other files are available in soft copy format):

- main.c – main program
- rpc.c – Radio Packet Controller routines

Network Protocol Java source code is provided for the following files only (full listings for other files are available in soft copy format):

- PiconetLayer.java
- NetLayer.java
- TraLayer.java
- IntLayer.java
- Packet.java
- PiconetSocket.java

```c
/*
 * File: main.c
 *
 * Description:
 *
 * Author: Israel Keys
 *
 * Last Modified: 18/10/2000
 */
#include <pic.h>
#include "utildefs.h"
#include "buf_def.h"
#include "usart.h"
#include "rpc.h"

/* state names for controller */
typedef enum statetype {
    IDLE, SEND_CTRL, SEND_DATA, RECV_CTRL, RECV_DATA
} main_statetype;

typedef enum isr_statetype {
    WAIT, START_PENDING, DLE_PENDING, GET_DATA
} isr_statetype;

/* serial control signals */
#define DLE 0x10
#define STX 0x02
#define ETX 0x03

/* buffer for rpc transmitting*/
buftype txbuf;
uchar pktflag;

/* isr variables */
uchar send_chksum;
uchar datacnt;
isr_statetype isrstate;

/*
 * isr -
 *  recieve service routine for usart
 */
static void interrupt isr(void)
{
    uchar d;

    /* buf producer: usart recieve */
    if (RCIF) {

        /* process incomming data according to state */
        switch(isrstate) {
            case WAIT:
                if(usartRX() == DLE)
                    isrstate = START_PENDING;
                break;
            case START_PENDING:
                if(usartRX() == STX) {
                    datacnt = 0;
                    send_chksum = 0;
                    isrstate = GET_DATA;
                }
                break;
            case GET_DATA:
                d = usartRX();
                if(d == DLE) {
                    isrstate = DLE_PENDING;
                } else {
                    send_chksum += d;
                    datacnt++;
                    bufWrite(txbuf, d);
                }
                break;
            case DLE_PENDING:
                switch(usartRX()) {
                    case ETX:
                        if(send_chksum) {

                            /* checksum error */
                            while(datacnt--)
                                bufRead(txbuf, d);
                        } else {
                            pktflag++;
                        }
                        isrstate = WAIT;
                        break;
                    case DLE:
                        isrstate = GET_DATA;
                        send_chksum += DLE;
                        datacnt++;
                        bufWrite(txbuf, DLE);
                        break;
                    case STX:
                        /* an error has occured
                         * flush buffer from last packet */
                        while(datacnt--)
                                bufRead(txbuf, d);
                        isrstate = GET_DATA;
                        break;
                    default:
                        isrstate = WAIT;
                        break;
                }
                break;
            default:
                isrstate = WAIT;
                break;
        }
    }
}

/*
 * main -
 *
 */
void main(void)
{
    uchar count, d;
    main_statetype state;
    uchar recv_sum;

    /*
     * initialise buffers, usart and rpc
     */
    datacnt = 0;
    pktflag = 0;
    isrstate = WAIT;
    bufInit(txbuf);
    usartInit(9600, SCI_EIGHT);
    TRISA = 0x00;           /* all output, RA0 = RPC reset*/
    rpcReset();
    rpcMemWrite (0x00, 0x03);   /* LBT, DBT */

    /* initialise interupts */
    PIR1 = 0;   /* clear pending interrupts */
    PEIE = 1;   /* perhipheral interrupts enabled */
    GIE  = 1;   /* global interrupts enabled */
```

```
137     RCIE = 1;   /* usart receiver interrupts enabled */
138
139     state = IDLE;
140     for(;;)
141         switch(state) {
142             case IDLE:
143                 RCIE = 0;
144                 TXR = !pktflag;
145                 RCIE = 1;
146                 if(!RXR)
147                     state = RECV_CTRL;
148                 else if(!TXA)
149                     state = SEND_CTRL;
150                 break;
151             case SEND_CTRL:
152                 count = MAX_PACKET_SIZE + 1;
153                 rpcTX(0x1F & count);
154                 count--;
155                 state = SEND_DATA;
156                 break;
157             case SEND_DATA:
158                 RCIE = 0;
159                 bufRead(txbuf, d)
160                 rpcTX(d);
161                 RCIE = 1;
162                 if(--count == 0) {
163                     pktflag--;
164                     bufRead(txbuf, d) /* check sum */
165                     state = IDLE;
166                 }
167                 break;
168             case RECV_CTRL:
169                 count = (0x1F & rpcRX()) - 1;
170                 usartTX(DLE);
171                 usartTX(STX);
172                 recv_sum = 0;
173                 state = RECV_DATA;
174                 break;
175             case RECV_DATA:
176                 recv_sum += (d = rpcRX());
177                 usartTX(d);
178                 if(d == DLE)
179                     usartTX(DLE);
180                 if(--count == 0) {
181                     usartTX(~recv_sum + 1);
182                     usartTX(DLE);
183                     usartTX(ETX);
184                     state = IDLE;
185                 }
186                 break;
187             default:
188                 rpcReset();
189                 state = IDLE;
190                 break;
191         }
192 }
193
194
195
```

```c
/*
 * File: rpc.c
 *
 * Description:
 *
 * Author: Israel Keys and Alec Siu
 *
 * Last Modified: 8/5/2000
 */
#include "rpc.h"

void rpcTX_nibble(uchar d)
{
    RPC = (0xF0 & (d << 4)) | 0x02; /* also set TXR = 0 */
    while(TXA)
        ;
    RPC_DDR = DATA_OUT;
    TXR = 1;
    while(!TXA)
        ;
}

void rpcTX(uchar d)
{
    rpcTX_nibble(0x0F & d);
    rpcTX_nibble(0x0F & (d >> 4));
    LISTENBUS;
}

uchar rpcRX_nibble(void)
{
    uchar data;

    while(RXR)
        ;
    RXA = 0;
    while(!RXR)
        ;
    asm("nop");
    data = 0xF0 & RPC;
    RXA = 1;
    return data >> 4;
}

uchar rpcRX(void)
{
    uchar data;
    LISTENBUS;
    data = 0x0F & rpcRX_nibble();
    data |= 0xF0 & (rpcRX_nibble() << 4);
    return data;
}

uchar rpcMemRead (uchar addr)
{
    uchar control = 0;
    bitset(control, PT);
    bitclr(control, WR);
    control |= (0x3F & addr);
    rpcTX(control);
    if (control != rpcRX()) {
        // do something!
    }
    return rpcRX();
}

void rpcMemWrite (uchar addr, uchar config)
{
```



```c
    uchar control = 0;
    bitset(control, PT);
    bitset(control, WR);
    control |= (0x3F & addr);
    rpcTX(control);
    rpcTX(config);
}

void rpcReset (void)
{
    TXR = 1;
    RXA = 1;
    RESET = 0;
    asm("nop"); asm("nop");
    RESET = 1;
    LISTENBUS;
}
```

```java
import java.io.*;
import java.util.*;

/**
 * <pre>
 * Implements a communication endpoint for a mobile host in a piconet.
 *
 * History:
 * - 09/10/00 IK, implemented sliding window alg.
 * - 31/09/00 IK, improved stop-n-wait, fixed problems found
 * - 27/09/00 IK, -
 * - 26/09/00 IK, implemented stop-and-wait algorithm.
 * - 24/09/00 IK, minor modifications
 * - 22/09/00 IK, changed to use new packet pool
 * - 19/09/00 IK, re-written to suit new transport layer.
 * - 16/09/00 IK, minor modifications
 * - 12/09/00 IK, piconet layer communication,
 *                unreliable datagram service.
 * - 09/09/00 IK, added support for comms to transport layer
 * - 06/09/00 IK, created
 * </pre>
 */
public class PiconetSocket implements TraLayerConstants {

    private PiconetLayer tl;
    private byte remoteaddr, localaddr;
    private DebugOut debugOut;

    private byteBuffer bufReceive = new byteBuffer(SOCKET_BUFFER_SIZE);
    private boolean isConnected = false;

    // for connection state machine (sender and receiver)
    private final int STATE_CLOSED = 0;
    private final int STATE_CONNECTING = 1;
    private final int STATE_ESTABLISHED = 2;

    // for sender
    private int sender_state;
    private final int SWS = 8;  // send window size
    private int LPS;            // last packet sent
    private int LAR;            // last ack recv.
    private TimedPacket[] SendWindow = new TimedPacket[SWS];
    private Object sw_mutex = new int[]{};
    private Semaphore semSpaces = new Semaphore(SWS);

    // for receiver
    private int receiver_state;
    private final int RWS = 1;  // recv. window size
    private int NPE;            // next packet expected

    // for both
    private final int MAX_SEQNO = 15; /* acctually 14 */

    /**
     * Constructs a new communication end point.
     * @param tl transport layer to communicate with
     * @param localaddr local address of the socket
     * @param remoteaddr remote address the socket is to communicate to.
     */
    public PiconetSocket(PiconetLayer tl, byte localaddr,
                         byte remoteaddr, DebugOut debugOut) {
        this.tl = tl;
        this.localaddr = localaddr;
        this.remoteaddr = remoteaddr;
        this.debugOut = debugOut;
        isConnected = false;
    }

```



```java
    public void connect() {
        if(!isConnected) {
            isConnected = true;

            // initilise sender
            sender_state = STATE_CONNECTING;
            LAR = 0;
            LPS = 0;

            // initialise receiver
            receiver_state = STATE_CLOSED;

            debug("Started");
        }

    }

    public void disconnect() {
        isConnected = false;
        sender_state = STATE_CLOSED;
        receiver_state = STATE_CLOSED;
        debug("Disconnected");
    }

    /**
     * Returns the remote address the socket is connected to
     */
    public byte getRemoteAddress()  {
        return remoteaddr;
    }


    /**
     * Returns the local address of the socket
     */
    public byte getLocalAddress() {
        return localaddr;
    }


     /**
      * Returns a string representation of the socket as
      * (localaddr, remoteaddr) address pair.
      */
     public String toString() {
        return "(" + localaddr + "," + remoteaddr + ")";
     }


     /**
      * Converts the given byte array to packets and sends them
      * to the transport layer.
      * Returns the number of bytes sent.
      */
     public int send(byte[] b, int off, int len) {
        Packet p;
        int packetlen, bytes_sent = len;

        while(len > 0) {

            // initialise new packet
            p = new Packet();
            p.setDestAddr(remoteaddr);
            p.setSrcAddr(localaddr);
            packetlen = (len > DATA_SIZE) ? (DATA_SIZE) : (len);
            p.setTraLength((byte)packetlen);
            p.setTraData(b, off, packetlen);

```

```
137             // set sequence number and add packet to send window
138             semSpaces.Wait();
139             LPS = (LPS + 1) % MAX_SEQNO;
140             p.setTraId((byte)LPS);
141             synchronized(sw_mutex) {
142                 SendWindow[LPS % SWS] =
143                     new TimedPacket(p, RESEND_TIMEOUT, MAX_RESEND);
144             }
145             send(p);
146
147             // calculate how mush is left to send
148             len -= packetlen;
149             off += packetlen;
150         }
151         return bytes_sent;
152     }
153
154     /**
155      *  Receive len bytes from the remote address
156      *  This method blocks until len bytes are
157      *  available. Returns the number of bytes received.
158      */
159     public int receive(byte[] b, int off, int len) {
160         for(int i = 0; i < len; i++)
161             b[off + i] = bufReceive.del();
162         return len;
163     }
164
165     /**
166      * Returns the number of bytes which can be
167      * received without blocking.
168      */
169     public int available() {
170         return bufReceive.available_del();;
171     }
172
173
174    /**
175     * Called by the transport layer when a packet is
176     * to be received by this socket.
177     */
178     public int deliver(Packet p) {
179
180         // read packet type and pass it to the appropriate fsm for processing
181         switch(p.getTraType()) {
182             case TYPE_CONNECT:
183                 return recv_connect(p);
184             case TYPE_DATA:
185                 return recv_data(p);
186             case TYPE_ACK:
187                 return recv_ack(p);
188         }
189         return 0;
190     }
191
192     /**
193      * Private function for formating debug output
194      */
195     private void debug(String str) {
196         debugOut.debugPrint("ps("+ localaddr + "," + remoteaddr + "): " +str);
197     }
198
199
200     private boolean inWindow(int seqnum, int min, int max) {
201         int pos = seqnum - min;
202         int maxpos = max - min + 1;
203         return pos < maxpos;
204     }
```



```
205
206
207     /**
208      * to be called when an ACK packet is recieved
209      */
210     public int recv_ack(Packet p) {
211         int seqnum = p.getTraId(); // sequence no. of ack
212
213         debug("ack packet received " + p);
214         if(inWindow(seqnum, LAR + 1, LPS)) {
215
216             // remove all timed packets in window less than the seqnum
217             // and cancel their resend timers
218             do {
219                 LAR = (LAR + 1) % MAX_SEQNO;
220                 synchronized(sw_mutex) {
221                     if(SendWindow[LAR % SWS] != null) {
222                         SendWindow[LAR % SWS].cancel();
223                         SendWindow[LAR % SWS] = null;
224                         semSpaces.Signal();
225                     }
226                 }
227             } while(LAR != seqnum);
228             sender_state = STATE_ESTABLISHED;
229
230         }
231         return p.packet.length;
232     }
233
234
235     /**
236      * Send a packet to the transport lower layer
237      */
238     private void send(Packet p) {
239         p.setTraType((sender_state==STATE_CONNECTING)
240             ? TYPE_CONNECT : TYPE_DATA);
241         debug("sending packet " + p);
242         tl.send(p);
243     }
244
245
246     /**
247      * to be called when a connection packet is received
248      */
249     private synchronized int recv_connect(Packet p) {
250         debug("connect packet received " + p);
251
252         // reset next packet expected
253         NPE = p.getTraId() + 1;
254
255         switch(receiver_state) {
256             case STATE_ESTABLISHED:
257             case STATE_CLOSED:
258                 if(take_data(p)) {
259                     debug("take data, return ack");
260                     send_ack(p.getTraId());
261                     receiver_state = STATE_CONNECTING;
262                 } else {
263                     debug("no space for data, dropping packet w/o ack");
264                 }
265                 break;
266             case STATE_CONNECTING:
267                 debug("ignore data, return ack");
268                 send_ack(p.getTraId());
269                 return p.packet.length;
270         }
271         return 0;
272     }
```

```
273
274     /**
275      * to be called when a data (TYPE_DATA) packet is
276      * recieved
277      */
278     private synchronized int recv_data(Packet p) {
279         debug("data packet received " + p);
280
281         // check that we are in a valid state for data
282         if(receiver_state == STATE_CLOSED) {
283             debug("not connected, dropping packet w/o ack");
284             return 0;
285         }
286         receiver_state = STATE_ESTABLISHED;
287
288         // only take the data if its the packet we expect
289         // and we have space
290         if(p.getTraId() == NPE) {
291             if(!take_data(p)) {
292
293                 // no data could be taken since no space
294                 debug("no space, dropping packet w/o ack");
295                 return 0;
296             }
297             debug("took data");
298             NPE = (NPE + 1) % MAX_SEQNO;
299         }
300
301         // return ack of highest in-order seqnum recieved
302         debug("returning ack");
303         int hseqnum = (NPE > 0) ? (NPE - 1) : (MAX_SEQNO - 1); //!!!
304         send_ack(hseqnum);
305         return p.packet.length;
306     }
307
308     /**
309      * takes all data from the packet.
310      * returns true if successful, false otherwise
311      */
312     private boolean take_data(Packet p) {
313         int len = p.getTraLength();
314         if(bufReceive.available_add() < len)
315             return false; // no space
316
317         // space available
318         bufReceive.add(p.packet, Packet.TRADATA_OFFSET, len);
319         return true;
320     }
321
322     /**
323      * Sends an ACK packet with the specified sequence number
324      */
325     private void send_ack(int seqnum) {
326         Packet p = new Packet();
327         p.setDestAddr(remoteaddr);
328         p.setSrcAddr(localaddr);
329         p.setTraType(TYPE_ACK);
330         p.setTraId((byte)seqnum);
331         tl.send(p);
332     }
333
334
335     /**
336      * Represents a packet which has been sent at least
337      * once and is awaiting an acknoledgement. The timed
338      * packet will continue sending itself periodically
339      * until the cancel method is called.
340      */
```



```
341     private class TimedPacket implements TimerUser {
342         private Packet p;
343         private Timer timer;
344         private int resend_cnt;
345         private int maxresend;
346
347         /**
348          * Constructs a new timed packet with a specified
349          * resend delay.
350          * @param p packet to be resent
351          * @delay time in milliseconds between resends
352          */
353         public TimedPacket(Packet p, long delay, int maxresend) {
354             this.p = p;
355             this.maxresend = maxresend;
356             resend_cnt = 0;
357             timer = new Timer(this, delay);
358             timer.start();
359         }
360
361         /**
362          * Cancel this object from any more resends
363          */
364         public synchronized void cancel() {
365             debug("timer canceled for " + p);
366             timer.cancel();
367         }
368
369         /**
370          * Called by the timer every time a resend
371          * is required.
372          */
373         public synchronized void timeout() {
374             debug("timeout " + p);
375             send(this.p);
376             if(++resend_cnt >= maxresend) {
377                 this.cancel();
378                 if(sender_state == STATE_CONNECTING)
379                     debugOut.error(ErrorCodeConstants.CONNECTION_FAILED);
380                 else
381                     debugOut.error(ErrorCodeConstants.CONNECTION_LOST);
382             }
383         }
384     }
385
386 }
387
```

```java
/**
 * <pre>
 * Provides a common interface for communication with any piconet
 * layer.
 *
 * History:
 * - 06/9/00 IK, creation
 * - 19/9/00 IK, changed from interface to abstact class
 * </pre>
 */
public abstract class PiconetLayer {

    /**
     * Should be set to true if the Layer starts successfully
     */
    protected boolean alive = false;

    /**
     * upper PiconetLayer
     */
    protected PiconetLayer ul = null;

    /**
     * lower PiconetLayer
     */
    protected PiconetLayer ll = null;

    /**
     * Check if the Layer is operational.
     * @return true if the Layer was booted successfully
     */
    public boolean isAlive () {
        return alive;
    }

    /**
     * Shuts down the Layer.
     */
    public void shutDown () {
        alive = false;
    }

    /**
     * Register upper layer for inter-Piconetlayer communication.
     * @param l upper PiconetLayer
     */
    public void ulRegister(PiconetLayer l) {
        ul = l;
    }

    /**
     * Register lower layer for inter-PiconetLayer communication.
     * @param l lower PiconetLayer
     */
    public void llRegister(PiconetLayer l) {
        ll = l;
    }

    /**
     * Called by upper layer when a packet is to be sent.
     * When Layer finishes processing, the packet is passed to the
     * lower layer if it exists.
     * @param p packet to be sent.
     * @return number of bytes sent.
     */
    public int send (Packet p) {
        if (ll != null)
            return ll.send(p);
        else
            return 0;
    }

    /**
     * Called by the lower layer when a packet is received.
     * When Layer finishes processing, the packet is passed to the
     * upper layer if it exists.
     * @param p the received packet
     * @return number of bytes received.
     */
    public int deliver (Packet p) {
        if (ul != null)
            return ul.deliver(p);
        else
            return 0;
    }

}
```

```java
import java.util.*;

/**
 * <pre>
 * Network Layer of the Ad Hoc Piconet protocol stack ("PicoStack")
 * History:
 * - 06/10: AS, fixed broadcast data not being forwarded, last hop always
 *          passive-acks DATA packets, unsolicited RREPs generated from
 *          passive ack data.
 * - 01/10: AS, changed to use packets containing full routes in header
 * - 19/09: IK, modified to use PacketPool
 * - 17/09: AS, some optimisations to remove redundant memory-to-memory
 *          copies, serial port flushes on errors in read/write.
 *          Reformatted comments for javadoc.
 * - 09/09: IK, modified for PiconetLayer callback interface (ie.
 *          deliver(), ulReceive() etc.)
 * - 04/09: AS, use of Packet objects rather than pure byte arrays.
 * - 31/08: AS, added statistics measures.
 * - 18/08: AS, created.
 * </pre>
 */
public class NetLayer extends PiconetLayer implements NetLayerConstants {

    public long controlBits = 0;
    public long dataBits = 0;
    private DebugOut debugOut = null;   // debugging output
    private byte selfAddr;              // this node's network address
    private RouteCache routeCache;      // cache of valid routes
    private HashCache requestCache;     // time of last route request
    private byte id;                    // sequence id of next packet
    private Timer[][] acks;             // pasvack timers

    /**
     * Constructor for the Network Layer of the PicoStack.
     * @param selfAddr this node's network address
     * @param debugOut any object which implements the DebugOut interface
     */
    public NetLayer (byte selfAddr, DebugOut debugOut) {
        this.debugOut = debugOut;
        this.selfAddr = selfAddr;

        routeCache   = new RouteCache(ROUTETABLE_THRESHOLD);
        requestCache = new HashCache(REQUESTCACHE_THRESHOLD);

        id = (byte) System.currentTimeMillis();

        acks = new Timer[MAX_NODES][MAX_NODES];

        debugOut.debugPrint("NetLayer started");
        alive = true;
    }

    /**
     * Return the local network address of this node
     */
    public byte getLocalAddr() {
        return selfAddr;
    }

    private int getRequestIndex (Packet p) {
        return (p.getPacketType() << 8) | p.packet[1];
    }

    private void debugPrint (String s) {
        if (debugOut != null)
            debugOut.debugPrint("nl(" + selfAddr + "): " + s);
    }

```



```java
    // Sends the Packet p and starts a timer for the passive
    // acknowledgement
    private int ackSend (Packet p) {
        byte srcAddr = p.getSrcAddr();
        byte destAddr = p.getDestAddr();
        if (acks[srcAddr][destAddr] == null) {
            acks[srcAddr][destAddr] = new Timer(
                    new RouteMonitor(srcAddr, destAddr), PASVACK_THRESHOLD
                );
            acks[srcAddr][destAddr].start();
        }
        return ll.send(p);
    }

    // Cancels the specified pasvack timer
    private void ackReceive (byte srcAddr, byte destAddr) {
        Timer t = acks[srcAddr][destAddr];
        if (t != null) {
            RouteMonitor rm = (RouteMonitor) t.user;
            t.cancel();
            t = null;
            debugPrint("Hop2Hop RTT = " + (System.currentTimeMillis() - rm.startTime));
        }

    }


    // Inner class for generating RERR packets when a passive ack
    // is not received in time.
    class RouteMonitor implements TimerUser {

        public long startTime = 0;
        byte srcAddr, destAddr;

        public RouteMonitor (byte srcAddr, byte destAddr) {
            this.srcAddr = srcAddr;
            this.destAddr = destAddr;
            this.startTime = System.currentTimeMillis();
        }

        public void timeout () {
            acks[srcAddr][destAddr].cancel();
            acks[srcAddr][destAddr] = null;
            // generate route error packet
            Packet p = new Packet();
            p.setPacketType(RERR_PKT);
            p.setSrcAddr(srcAddr);
            p.setDestAddr(destAddr);
            p.setId(++id);
            if (srcAddr == selfAddr)
                deliver(p);
            else {
                controlBits += 16;
                send(p);
            }
        }
    }

    /**
     * Send a packet. Called by upper layer when a packet needs to
     * be sent.
     * @param p Packet to be sent by upper layer
     * @return number of bytes written
     */
    public int send (Packet p) {
        debugPrint("send request");
        byte destAddr = p.getDestAddr();
        // handle loop-back
```

```
137         if (destAddr == selfAddr) {
138             deliver(p);
139         }
140         // if the route table contains a route to the destination,
141         // send it directly to the serial port
142         byte[] path = routeCache.get(destAddr);
143         if (path != null) {
144             p.setPath(path); // , routeCache.index(path, selfAddr), path.length - 1);
145             p.setNextOrd(1);
146             p.setPacketType(DATA_PKT);      // set packet type to DATA
147             return ackSend(p);
148         }
149         // no route currently exists, quietly drop the packet;
150         // send a route request for the destination (provided we haven't
151         // sent one in the recent past)
152         int index = getRequestIndex(p);
153         if (!requestCache.recent(index)) {
154             requestCache.touch(index);
155             Packet p2 = new Packet();
156             p2.setRouteRequest(p.getSrcAddr(), destAddr, ++id);
157             controlBits += 16;
158             debugPrint("RREQ time " + p2 + " " + System.currentTimeMillis());
159             ll.send(p2);
160         }
161         return 0;
162     }
163
164     private int handleData (Packet p) {
165         byte srcAddr = p.getSrcAddr(), destAddr = p.getDestAddr();
166         int ord = p.getOrd(selfAddr), nextOrd = p.getNextOrd();
167         // pass packet to upper layer (if it exists) if packet's
168         // destination address is self or the broadcast address
169         if (destAddr == selfAddr || destAddr == BROADCAST_ADDR) {
170             // forward broadcast messages up to certain limit, also
171             // force retransmission for pasv ack purposes even if we are
172             // the destination
173             if (destAddr == selfAddr || nextOrd < MAX_REBROADCASTS) {
174                 p.setNextOrd(ord + 1);
175                 controlBits += 40;
176                 dataBits += 176;
177                 ll.send(p);
178             }
179             if (ul != null)
180                 return ul.deliver(p);
181         }
182         // the packet is not destined for us, but we're in the path;
183         // increment the hop index and forward the packet
184         else if (ord >= 0) {
185             // we received the packet in the expected order
186             if (nextOrd == ord) {
187                 routeCache.touch(destAddr);
188                 p.setNextOrd(ord + 1);
189                 return ackSend(p);
190             }
191             // we received the packet earlier than expected;
192             // generate an unsolicited RREP indicating a shorter
193             // path is possible
194             else if (nextOrd < ord) {
195                 Packet p2 = new Packet();
196                 p2.setPacketType(RREP_PKT);
197                 p2.setPathLength(1);
198                 p2.setSrcAddr(srcAddr);
199                 p2.setDestAddr(destAddr);
200                 for (int i = 1; i < nextOrd; i++)
201                     p2.appendAddr(p.getAddr(i));
202                 p2.appendAddr(selfAddr);
203                 for (int i = ord + 1; i < p.getPathLength(); i++)
204                     p2.appendAddr(p.getAddr(i));
```



```
205                 controlBits += 40;
206                 return ll.send(p);
207             }
208             // we received the packet from the next hop
209             // (i.e. nextOrd == ord + 2) => passive ack
210             else if (nextOrd == ord + 2) {
211                 routeCache.touch(destAddr);
212                 ackReceive(srcAddr, destAddr);
213             }
214         }
215         // we're not in the path, so drop it (ie. do nothing)
216 //        else {
217 //            ;
218 //        }
219         return 0;
220     }
221
222     // TBD: check if in the route table and send a reply without
223     // having to append self to path
224     private int handleRouteRequest (Packet p) {
225         byte srcAddr = p.getSrcAddr(), destAddr = p.getDestAddr();
226         int index = getRequestIndex(p);
227         if (requestCache.recent(index)) {
228             debugPrint("");
229         }
230         // we are the target of the route request -> reply
231         // with packet contents (route) unmodified
232         else {
233             requestCache.touch(index);
234             if (destAddr == selfAddr) {
235                 p.setPacketType(RREP_PKT);
236                 controlBits += 40;
237                 return ll.send(p);
238             }
239             // not in the route table, and self not in the path,
240             // append self to path and pass it on
241             else if (p.getOrd(selfAddr) < 0) {
242                 p.appendAddr(selfAddr);
243                 controlBits += 40;
244                 return ll.send(p);
245             }
246         }
247         return 0;
248     }
249
250     private int handleRouteReply (Packet p) {
251         debugPrint("RREP time = " + p + " " + System.currentTimeMillis());
252         byte srcAddr = p.getSrcAddr(), destAddr = p.getDestAddr();
253         int index = getRequestIndex(p);
254         if (requestCache.recent(index)) {
255             debugPrint("");
256         }
257         // if self is in the path
258         else {
259             requestCache.touch(index);
260             byte[] path = p.getPath(selfAddr);
261             // if there's a path starting from self in the RREP packet
262             if (path != null && path.length > 1) {
263                 routeCache.insert(destAddr, path);
264             }
265             if (srcAddr != selfAddr) {
266                 controlBits += 40;
267                 return ll.send(p);
268             }
269         }
270         return 0;
271     }
272
```

```
273     private int handleRouteError (Packet p) {
274         byte srcAddr = p.getSrcAddr(), destAddr = p.getDestAddr();
275         int index = getRequestIndex(p);
276         if (requestCache.recent(index)) {
277             debugPrint("");
278         }
279         else {
280             requestCache.touch(index);
281             // we are the target of the route error; create a new
282             // route request for the specified route
283             if (srcAddr == selfAddr) {
284                 //routeCache.del(destAddr);
285                 Packet p2 = new Packet();
286                 p2.setRouteRequest(srcAddr, destAddr, ++id);
287                 controlBits += 40;
288                 ll.send(p2);
289             }
290             else {
291                 controlBits += 40;
292                 ll.send(p);
293             }
294         }
295         return 0;
296     }
297
298     /**
299      * Receive a packet. Called by lower layer when a packet
300      * is received. Stateless design removes need for buffering
301      * and retransmission etc.
302      * @param p Packet received by lower layer
303      * @return number of bytes received
304      */
305     public int deliver (Packet p) {
306         switch (p.getPacketType()) {
307             case DATA_PKT:
308                 return handleData(p);
309             case RREQ_PKT:
310                 return handleRouteRequest(p);
311             case RREP_PKT:
312                 return handleRouteReply(p);
313             case RERR_PKT:
314                 return handleRouteError(p);
315         }
316         return 0;
317     }
318
319 }
320
```

```java
import java.io.*;
import java.util.*;

/**
 * <pre>
 * Transport Layer for Piconet Stack. Provides Multiplexing and Demultiplexing
 * of Packets between the lower PiconetLayer and a set of registered
 * PiconetSockets.
 *
 *
 * History:
 * - 09/10/00 IK, added accept method/functinality.
 * - 27/09/00 IK, minor modifications
 * - 24/09/00 IK, changed list of sockets from hashtable to array,
 * - 22/09/00 IK, modified to use new packet pool
 * - 19/09/00 IK, re-written to support mux/demux functionality only.
 * - 12/09/00 IK, connection establishment via accept and connect.
 *                basic PiconetSocket communication.
 *                unreliable datagram service.
 * - 09/09/00 IK, minor modifications
 * - 06/08/00 IK, added Piconet Layer support
 * - 14/08/00 IK, created.
 * </pre>
 */
public class TraLayer extends PiconetLayer implements TraLayerConstants {

    private PiconetSocket[] socks;
    private String[] names;
    private byte localaddr;
    private String name;
    private DebugOut debugOut;

    private boolean accepting;
    private byte accepting_addr;
    private Object accepting_mutex = new int[]{};
    private Semaphore semAccept;

    /**
     * Constructs a Transport Layer without a name.
     * @param localaddr address of the transport layer (nibble)
     * @param debugOut interface for debug output
     */
    public TraLayer(byte localaddr, DebugOut debugOut) {
        this(localaddr, "", debugOut);
    }

    /**
     * Constructs a named Transport Layer.
     * @param localaddr address of the transport layer (nibble)
     * @param name name of transport layer
     * @param debugOut interface for debug output
     */
    public TraLayer(byte localaddr, String name, DebugOut debugOut) {
        this.socks = new PiconetSocket[MAX_NODES];
        for(int i = 0; i < MAX_NODES; i++)
            socks[i] = null;
        this.names = new String[MAX_NODES];
        this.localaddr = localaddr;
        this.name = name;
        this.debugOut = debugOut;
        accepting = false;
        semAccept = new Semaphore();
        super.alive = true;
        debug("started");
    }

    /**
     *  Returns the local address of the Transport Layer
```



```java
     */
    public byte getLocalAddress() {
        return localaddr;
    }

    /**
     *  Returns the name given
     */
    public String getName() {
        return name;
    }


    /**
     * Called by the lower layer when a packet is received.
     * The packet is passed to the appropriate socket if it exists,
     * otherwise it ignored and returned to the packet pool
     */
    public int deliver (Packet p) {
        debug ("received packet " + p);
        // find socket
        PiconetSocket sock = socks[p.getSrcAddr()];

        //if we are accepting, and the packet is a connect packet
        synchronized(accepting_mutex) {
            if((sock == null) && accepting && (p.getTraType() == TYPE_CONNECT)) {

                // add a new socket
                accepting_addr = p.getSrcAddr();
                connect(accepting_addr);

                // tell the accept method that we have a new socket
                semAccept.Signal();
            }
        }

        // if a socket does not exist, ignore packet
        if(sock == null) {
            debug ("socket not available. dropping packet " + p);
            return 0;
        }

        // otherwise, pass packet to socket
        debug ("delivering packet to socket " + p);
        return sock.deliver(p);
    }

    public int send(Packet p) {
        debug ("sending packet " + p);
        return super.send(p);
    }

    /**
     * Registers a PiconetSocket as a communication end point.
     * @param sock PiconetSocket to be registered.
     */
    public void socketRegister(PiconetSocket sock) {
        socks[sock.getRemoteAddress()] = sock;
    }

    /**
     * UnRegisters a PiconetSocket
     * @param sock PiconetSocket to be unregistered.
     */
    public void socketUnRegister(PiconetSocket sock) {
        socks[sock.getRemoteAddress()] = null;
    }

```

```
137     /**
138      * Returns a PiconetSocket connected to a remote host.
139      * @param remoteaddr address of remote host
140      */
141     public PiconetSocket connect(byte remoteaddr) {
142         if(socks[remoteaddr] == null)
143             socks[remoteaddr] = new PiconetSocket(this, localaddr,
144                                     remoteaddr, debugOut);
145         socks[remoteaddr].connect();
146         return socks[remoteaddr];
147     }
148
149     /**
150      * Returns a PiconetSocket connected to a remote host.
151      * @param remoteaddr name of remote host
152      */
153     public PiconetSocket connect(String remotename) {
154         byte remoteaddr = 0;
155         return connect(remoteaddr);
156     }
157
158     /**
159      * Blocks until a connect packet is received from an
160      * unknown remote host and returns a connected PiconetSocket.
161      */
162     public PiconetSocket accept() {
163         synchronized(accepting_mutex) {
164             accepting = true;
165         }
166         debug("accept waiting...");
167         semAccept.Wait();
168         debug("no longer waiting...");
169         synchronized(accepting_mutex) {
170             accepting = false;
171         }
172         return socks[accepting_addr];
173     }
174
175
176     /**
177      * Private function for formating debug output
178      */
179     private void debug(String str) {
180         debugOut.debugPrint("tl(" + localaddr + "): " +str);
181     }
182
183 }
184
185
```

```java
import com.ibm.oti.palmos.*;

/**
 * <pre>
 * Java wrapper for PalmOS serial port functions.
 * History:
 * - 15/10: AS, changed to implement byte-stuffing and framing.
 * - 19/09: AS, changed configuration method. Changed comments for javadoc.
 * - 15/09: AS, added functions to flush read/write buffers; some
 *          optimisations by removing redundant object creation.
 * - 18/08: AS, created.
 * </pre>
 */
public class IntLayerPalm extends PiconetLayer implements NetLayerConstants {

    /* states for the unstuffing procedure */
    private static final int SYNC = 0;
    private static final int WAIT = 1;
    private static final int RXING = 2;
    private static final int UNSTUFF = 3;

    /* framing characters */
    private static final byte DLE = 0x10;
    private static final byte STX = 0x02;
    private static final byte ETX = 0x03;

    private boolean serOpened = false;
    private int serHandle;
    private int timeOut;
    private Int16Ptr errWriteVoidPtr = new Int16Ptr();
    private Int16Ptr errReadVoidPtr = new Int16Ptr();
    private DebugOut debugOut = null;
    private Configuration config = new Configuration();

    /**
     * Create a new Interface Layer for the Piconet stack
     * @param debugOut Object implementing the DebugOut interface.
     */
    public IntLayerPalm (DebugOut debugOut) {
        this.debugOut = debugOut;
        if (open() < 0)
            return;
        if (configure() < 0)
            return;
        readFlush();
        writeFlush();
        alive = true;
        new Listener().start();
        debugOut.debugPrint("IntLayer booted");
    }

    /**
     * Shut down the Interface Layer.
     */
    public synchronized void shutDown () {
        alive = false;
        close();
    }

    /**
     * Open the serial port for reading and writing.
     * @return 0 if the serial port was opened successfully, error
     * code otherwise
     */
    private int open () {
        int err;
        CharPtr serLibName = new CharPtr("Serial Library");
        Int16Ptr serHandlePtr = new Int16Ptr();
```



```java

        if ((err = OS.SysLibFind(serLibName, serHandlePtr)) != 0) {
            return err;
        }
        serHandle = serHandlePtr.getShortAt(0);
        serHandlePtr.dispose();
        serLibName.dispose();

        err = OS.SerOpen(serHandle, (short) 0, config.baudRate);
        if (err == OSConsts.serErrAlreadyOpen) {
            OS.SerClose(serHandle);
        }
        if (err != 0) {
            return err;
        }

        serOpened = true;
        return 0;
    }

    private int configure () {
        int err, flags = 0;
        switch (config.parity) {
            case Configuration.PARITY_EVEN:
                flags |= OSConsts.serSettingsFlagParityOnM;
                flags |= OSConsts.serSettingsFlagParityEvenM;
                break;
            case Configuration.PARITY_ODD:
                flags |= OSConsts.serSettingsFlagParityOnM;
                break;
            case Configuration.PARITY_NONE:
                break;
        }
        switch (config.dataBits) {
            case Configuration.DATABITS_5:
                flags |= OSConsts.serSettingsFlagBitsPerChar5;
                break;
            case Configuration.DATABITS_6:
                flags |= OSConsts.serSettingsFlagBitsPerChar6;
                break;
            case Configuration.DATABITS_7:
                flags |= OSConsts.serSettingsFlagBitsPerChar7;
                break;
            case Configuration.DATABITS_8:
                flags |= OSConsts.serSettingsFlagBitsPerChar8;
                break;
        }
        switch(config.stopBits) {
            case Configuration.STOPBITS_1:
                flags |= OSConsts.serSettingsFlagStopBits1;
                break;
            case Configuration.STOPBITS_2:
                flags |= OSConsts.serSettingsFlagStopBits2;
                break;
        }
        if (config.hardwareFlowControl != 0) {
            flags |= OSConsts.serSettingsFlagCTSAutoM;
            flags |= OSConsts.serSettingsFlagRTSAutoM;
        }

        SerSettingsType serSettings = new SerSettingsType();

        serSettings.setBaudRate(config.baudRate);
        serSettings.setFlags(flags);
        serSettings.setCtsTimeout(OS.SysTicksPerSecond() * 5);

        if ((err = OS.SerSetSettings(serHandle, serSettings)) != 0) {
            return err;
```

```
137         }
138
139         timeOut = config.readIntervalTimeout * OS.SysTicksPerSecond() / 1000;
140
141         serSettings.dispose();
142         return 0;
143     }
144
145     private void close () {
146         if (!serOpened)
147             return;
148         OS.SerClose(serHandle);
149         serOpened = false;
150         errWriteVoidPtr.dispose();
151         errReadVoidPtr.dispose();
152     }
153
154     private int read (CharPtr buf, int len) {
155         int err = 0, bytesRead = 0;
156         // retry up to 10 times if there's a serial line error
157         for (int i = 0; i < 10; i++) {
158             bytesRead = OS.SerReceive(serHandle, buf, len, timeOut, errReadVoidPtr);
159             if ((err = errReadVoidPtr.getShortAt(0)) == OSConsts.serErrLineErr) {
160                 OS.SerClearErr(serHandle);
161             }
162             else {
163                 break;
164             }
165         }
166         return (err == 0 || err == OSConsts.serErrTimeOut) ? bytesRead : -1;
167     }
168
169     private int read (byte[] buf) {
170         if (!serOpened) {
171             return 0;
172         }
173         int bytesRead;
174         CharPtr buffer = new CharPtr(buf);
175         if ((bytesRead = read(buffer, buf.length)) >= 0) {
176             for (int i = 0; i < bytesRead; i++) {
177                 buf[i] = (byte) buffer.getCharAt(i);
178             }
179             buffer.dispose();
180             return bytesRead;
181         }
182         buffer.dispose();
183         return -1;
184     }
185
186     private int write (byte[] buf) {
187         CharPtr buffer = new CharPtr(buf);
188         int bytesWritten = 0;
189         int err = 0;
190
191         for (int tries = 0; tries < 10; tries++) {
192             bytesWritten = OS.SerSend(serHandle, buffer, buf.length, errWriteVoidPtr);
193             if ((err = errWriteVoidPtr.getShortAt(0)) == OSConsts.serErrLineErr) {
194                 OS.SerClearErr(serHandle);
195             }
196             else {
197                 break;
198             }
199         }
200         buffer.dispose();
201
202         if (err == 0)
203             return bytesWritten;
204         if (err == OSConsts.serErrTimeOut)
```



```
205             return 0;
206         return -1;
207     }
208
209     /**
210      * Flush the serial port receive buffer.
211      */
212     public void readFlush () {
213         OS.SerReceiveFlush(serHandle, 0);
214     }
215
216     /**
217      * Flush the serial port's write buffer.
218      */
219     public int writeFlush () {
220         return OS.SerSendFlush(serHandle);
221     }
222
223     public int send (Packet p) {
224         byte[] dummy = new byte[p.packet.length * 2];
225         int index = 2, checksum = 0;
226         if (!serOpened)
227             return 0;
228         dummy[0] = DLE;
229         dummy[1] = STX;
230         for (int i = 0; i < p.packet.length; i++) {
231             checksum += p.packet[i];
232             dummy[index++] = p.packet[i];
233             if (p.packet[i] == DLE)
234                 dummy[index++] = DLE;  // stuff DLE characters
235         }
236         dummy[index++] = (byte) ((~checksum + 1) & 0xFF); // checksum
237         dummy[index++] = DLE;
238         dummy[index++] = ETX;
239         debugOut.debugPrint("Packet delay (down) = " + (System.currentTimeMillis() - p.start'
240         return write(dummy);
241     }
242
243     /**
244      * Poll the serial port for data -- cannot use interrupts since
245      * the Palm OS serial port API blocks. Have to use timeout
246      * mechanism instead to yield to other processes.
247      */
248     class Listener extends Thread {
249
250         public Listener () {
251             setDaemon(true);
252         }
253
254         public void run () {
255             int state = SYNC, index = 0; byte checksum = 0;
256             byte[] dummy = null; //new byte[28];    // magic no.
257             byte[] dummy2 = new byte[1];
258             while (true) {
259                 while (read(dummy2) <= 0)
260                     ;
261                 byte currentByte = dummy2[0];
262                 switch (state) {
263                     case SYNC:
264                         if (currentByte == DLE)
265                             state = WAIT;
266                         break;
267                     case WAIT:
268                         if (currentByte == STX) {
269                             dummy = new byte[28];
270                             index = checksum = 0;
271                             state = RXING;
272                         }
```

```
273                         else
274                             state = SYNC;
275                         break;
276                     case RXING:
277                         if (currentByte == DLE) {
278                             state = UNSTUFF;
279                         }
280                         else {
281                             try {
282                                 dummy[index++] = (byte) currentByte;
283                                 checksum += currentByte;
284                             }
285                             catch (IndexOutOfBoundsException ioobe) {
286                                 state = SYNC;
287                             }
288                         }
289                         break;
290                     case UNSTUFF:
291                         switch (currentByte) {
292                             case ETX:
293                                 if ((checksum & 0xFF) == 0) {
294                                     Packet p = new Packet();
295                                     System.arraycopy(dummy, 0, p.packet, 0, p.packet.length)
296                                     p.startTime = System.currentTimeMillis();
297                                     ul.deliver(p);
298                                 }
299                                 state = SYNC;
300                                 break;
301                             case STX:
302                                 dummy = new byte[28];
303                                 index = checksum = 0;
304                                 state = RXING;
305                                 break;
306                             case DLE:
307                                 try {
308                                     dummy[index++] = (byte) currentByte;
309                                 }
310                                 catch (IndexOutOfBoundsException ioobe) {
311                                     state = SYNC;
312                                 }
313                                 checksum += currentByte;
314                                 state = RXING;
315                                 break;
316                             default:
317                                 state = SYNC;
318                         }
319
320                 }
321             }
322         }
323     }
324
325     class Configuration {
326
327         public static final int PARITY_NONE   = 0;
328         public static final int PARITY_EVEN   = 1;
329         public static final int PARITY_ODD    = 2;
330         public static final int PARITY_MARK   = 3;
331         public static final int PARITY_SPACE  = 4;
332
333         public static final int STOPBITS_1    = 0;
334         public static final int STOPBITS_1_5  = 1;
335         public static final int STOPBITS_2    = 2;
336
337         public static final int DATABITS_5    = 5;
338         public static final int DATABITS_6    = 6;
339         public static final int DATABITS_7    = 7;
340         public static final int DATABITS_8    = 8;
```



```
341
342         public int baudRate             = 9600;
343         public int parity               = PARITY_NONE;
344         public int dataBits             = DATABITS_8;
345         public int stopBits             = STOPBITS_1;
346         public int hardwareFlowControl  = 0;
347         public int softwareFlowControl  = 0;
348         public int readIntervalTimeout  = 10;
349         public int readTotalTimeout     = 10000;
350         public int writeTotalTimeout    = 10000;
351
352     }
353
354 }
355
```

```java
import java.io.*;

/**
 * <pre>
 * Packet object for the PicoStack; used by both the Network and
 * Transport Layers. Encapsulates knowledge of packet formats at
 * both these layers to prevent expensive memory-to-memory copying.
 *
 * History:
 * - 29/09 AS, Changed to include full route in the header.
 * - 22/09 IK, Added built in object pooling functionality. This
 *             allows the sharing of instantiated objects for increased
 *             performance. [Ref 1]
 * - 18/09 AS, re-formatted comments for javadoc. Added utility functions
 *         for RREQ/RREP handling (ie. inRoute(), appendToRoute()).
 * - 04/09 AS, created.
 *
 * References:
 * 1) Sosnoski, D "Java performance programming, Part 1: Smart
 *    object-management saves the day", www.javaworld.com, Nov. '99
 *
 * General notes:
 * - NetLayer packet format
 *   5-byte header, 22-byte data
 *   type (2 bits), path length in hops (3 bits), next hop offset (3 bits),
 *   source address (4 bits), destination address (4 bits),
 *   hop 1 up to hop 6 (1 - 3 bytes)
 * - TraLayer packet format (falls in data section of NetLayer packet)
 *   2-byte header, 20-byte data
 *   type (3 bits), length (5 bits), id (8 bits), data (20 bytes)
 * </pre>
 */
public class Packet implements NetLayerConstants, TraLayerConstants {

    public long startTime = 0;
    public static final int NETDATA_OFFSET = 5;
    public static final int TRADATA_OFFSET = 7;

    private static final String[] PKT_TYPE = new String[]
            {"DATA", "RREQ", "RREP", "RERR"};

    //------------------------------------------------------------------------
    // Packet Constuctors
    //------------------------------------------------------------------------

    /**
     * Byte array containing raw packet data. This member is made
     * public for convenience purposes, thus modifying this member
     * directly should be done with caution.
     */
    public byte[] packet;

    /**
     * Length of the packet (for variable length packet sizes)
     */
    public int length;

    /**
     * Constructs a Packet with the default Network Layer PDU size
     * of 27 bytes.
     */
    public Packet () {
        packet = new byte[NetLayerConstants.PDU_SIZE];
        length = packet.length; /* to be changed !! */
    }

    //------------------------------------------------------------------------
    // Network Layer setters and getters
```



```java
    //------------------------------------------------------------------------

    /**
     * Sets the data section of the Network Layer PDU from the specified
     * buffer. No error checking for exceeding the packet length!
     * Also assumes a DATA_PKT.
     * @param data buffer containing the data
     * @param offset offset into the buffer from which to begin copying
     * @param len number of bytes to copy
     */
    public void setData (byte[] data, int offset, int len) {
        System.arraycopy(data, offset, packet, NETDATA_OFFSET, len);
    }

    /**
     * Set the packet's (NetLayer) sequence id.
     * @param id sequence id to set to
     */
    public void setId (byte id) {
        packet[0] = (byte) ((packet[0] & 0xF8) | (id & 0x07));
    }

    /**
     * Get the packet's (NetLayer) sequence id.
     @return the packet's sequence id
     */
    public byte getId () {
        return (byte) (packet[0] & 0x07);
    }

    //------------------------------------------------------------------------
    // Transport Layer header getter and setters
    //------------------------------------------------------------------------

    /**
     * Set the packet's (TraLayer) type.
     */
    public final void setTraType (byte traType) {
        packet[NETDATA_OFFSET] = (byte) ((packet[NETDATA_OFFSET] & 0x1F) | (traType << 5));
    }

    /**
     * Get the packet's (TraLayer) type.
     * @return the packet's type
     */
    public final byte getTraType () {
        return (byte) ((packet[NETDATA_OFFSET] >> 5) & 0x07);
    }

    /**
     * Set the packet's (TraLayer) length.
     * @param length length value to set to
     */
    public final void setTraLength (byte length) {
        packet[NETDATA_OFFSET] = (byte) ((packet[NETDATA_OFFSET] & 0xE0) | (length & 0x1F));
    }

    /**
     * Get the packet's (TraLayer) length.
     * @return the packet's length in bytes
     */
    public final byte getTraLength () {
        return (byte) (packet[NETDATA_OFFSET] & 0x1F);
    }

    /**
     * Set the packet's (TraLayer) id.
     * @param id id value to set to
```

```
137      */
138     public final void setTraId (byte id) {
139         packet[NETDATA_OFFSET+1] = id;
140     }
141
142     /**
143      * Get the packet's (TraLayer) id.
144      * @return the packet's id
145      */
146     public final byte getTraId () {
147         return packet[NETDATA_OFFSET+1];
148     }
149
150     /**
151      * Sets the data section of the Transport Layer PDU from the
152      * specified buffer. No error checking for exceeding the packet length!
153      * @param data buffer containing the data
154      * @param offset offset into the buffer from which to begin copying
155      * @param len number of bytes to copy
156      */
157     public final void setTraData (byte[] data, int offset, int len) {
158         System.arraycopy(data, offset, packet, TRADATA_OFFSET, len);
159     }
160
161     /**
162      * String representation of a packet.
163      * @return [<i>packet id</i>, (<i>source address</i>,
164      * <i> destination address</i>)]
165      */
166     public String toString () {
167         String s = "[" + PKT_TYPE[getPacketType()] + ",";
168         if (getPacketType() == DATA_PKT)
169             s += strTYPE[getTraType()] + ",seq=" + getTraId() + ",";
170         s += getId() + ",(" + getSrcAddr() + "," + getDestAddr() + ")]";
171
172         return s;
173     }
174
175     /**
176      * Set the path section of the packet. Used for data packets
177      * generally.
178      * @param path In-order byte array of network addresses
179      * representing the path.
180      */
181     public final void setPath (byte[] path) {
182         setPathLength(path.length-1);
183         setSrcAddr(path[0]);
184         setDestAddr(path[path.length-1]);
185         int index = 2;
186         for (int i = 1; i < path.length-1; i++) {
187             if (i % 2 != 0)
188                 setUpperNibble(index, path[i]);
189             else {
190                 setLowerNibble(index, path[i]);
191                 index++;
192             }
193         }
194     }
195
196     /**
197      * Extracts the path from the Packet, beginning from selfAddr,
198      * terminating at the destination.
199      * @param selfAddr Network address from which to start extracting
200      * the path (used in promiscuous listening).
201      * @return In-order byte array of network addresses representing
202      * the path.
203      */
204     public final byte[] getPath (byte selfAddr) {
```



```
205         int selfAddrOffset = getOrd(selfAddr), pathLength = getPathLength();
206         if (selfAddrOffset == -1)
207             return null;
208         byte[] path = new byte[pathLength - selfAddrOffset + 1];
209         for (int i = 0; i < path.length; i++)
210             path[i] = getAddr(i+selfAddrOffset);
211         return path;
212     }
213
214     /**
215      * Set the path length to the specified value.
216      * @param len The length value to set.
217      */
218     public void setPathLength (int len) {
219         packet[0] = (byte) (((len & 0x07) << 3) | (packet[0] & 0xC7));
220     }
221
222     private int getPathBytes () {
223         return 1 + (getPathLength() / 2); // + (getPathLength() % 2);
224     }
225
226     /**
227      * Return the path length.
228      * @return the length of the path including source and destination.
229      */
230     public final int getPathLength () {
231         return (packet[0] >> 3) & 0x07;
232     }
233
234     /**
235      * Return the hop index.
236      * @return the hop index.
237      */
238     public final int getNextOrd () {
239         return packet[0] & 0x07;
240     }
241
242     /**
243      * Set the hop index to the specified value.
244      * @param the value of the hop index to set.
245      */
246     public void setNextOrd (int ord) {
247         packet[0] = (byte) ((packet[0] & 0xF8) | (ord & 0x07));
248     }
249
250     /**
251      * Get the length of the data section of the Network Layer PDU
252      * @return the length in bytes.
253      */
254     public final int getDataLength () {
255         return NetLayerConstants.PDU_SIZE - 1 - getPathBytes();
256     }
257
258     /**
259      * Get the index of the specified address within the route record.
260      * @param addr the address to look for
261      * @return the index within the route record of addr
262      */
263     public final int getOrd (byte addr) {
264         if (addr == getSrcAddr())
265             return 0;
266         else if (addr == getDestAddr())
267             return getPathLength();
268         for (int i = 1; i < getPathLength(); i++) {
269             if (getAddr(i) == addr)
270                 return i;
271         }
272          return -1;
```

```
273     }
274
275     /**
276      * Get the network address at the specified index within the route record.
277      * @return the network address
278      */
279     public final byte getAddr (int ord) {
280         if (ord == 0)
281             return getSrcAddr();
282         else if (ord == getPathLength())
283             return getDestAddr();
284         int offset = 1 + (ord / 2) + (ord % 2);
285         //System.out.println("ord == " + ord + " offset == " + offset);
286         if (ord % 2 == 0)
287             return getLowerNibble(offset);
288         return getUpperNibble(offset);
289     }
290
291     /**
292      * Appends the specified address to the route record.
293      * @param addr the address to append
294      */
295     public void appendAddr (byte addr) {
296         int pathOffset = getPathBytes(), pathLength = getPathLength();
297         if (pathLength % 2 == 0)
298             setLowerNibble(pathOffset, addr);
299         else
300             setUpperNibble(pathOffset + 1, addr);
301         setPathLength(pathLength + 1);
302     }
303
304     /**
305      * Convenience function to set a RREQ packet for the specified source,
306      * destination and UPI.
307      * @param srcAddr source address
308      * @param destAddr destination address
309      * @param id UPI
310      */
311     public final void setRouteRequest (byte srcAddr, byte destAddr, byte id) {
312         setPacketType(RREQ_PKT);
313         setSrcAddr(srcAddr);
314         setDestAddr(destAddr);
315         setPathLength((byte) 1);
316         setId(id);
317     }
318
319     /**
320      * Return the source address of the Packet.
321      */
322     public final byte getSrcAddr () {
323         return getUpperNibble(1);
324     }
325
326     /**
327      * Return the destination address of the Packet.
328      */
329     public final byte getDestAddr () {
330         return getLowerNibble(1);
331     }
332
333     /**
334      * Set the destination address of the Packet.
335      * @param destination address to set
336      */
337     public final void setDestAddr (byte destAddr) {
338         setLowerNibble(1, destAddr);
339     }
340
```



```
341     /**
342      * Set the source address of the Packet.
343      * @param source address to set
344      */
345     public final void setSrcAddr (byte srcAddr) {
346         setUpperNibble(1, srcAddr);
347     }
348
349     /**
350      * Get the packet's (NetLayer) type.
351      * @return the packet's type
352      */
353     public final byte getPacketType () {
354         return (byte) ((packet[0] >> 6) & 0x03);
355     }
356
357     /**
358      * Set the packet's (NetLayer) type.
359      */
360     public final void setPacketType (byte packetType) {
361         packet[0] = (byte) ((packet[0] & 0x3F) | (packetType << 6));
362     }
363
364     // convenience functions for setting nibbles
365     private byte getUpperNibble (int index) {
366         return (byte) ((packet[index] >> 4) & 0x0F);
367     }
368
369     private void setUpperNibble (int index, byte value) {
370         packet[index] = (byte) ((packet[index] & 0x0F) | (value << 4));
371     }
372
373     private  byte getLowerNibble (int index) {
374         return (byte) (packet[index] & 0x0F);
375     }
376
377     private void setLowerNibble (int index, byte value) {
378         packet[index] = (byte) ((packet[index] & 0xF0) | (value & 0x0F));
379     }
380
381     public String getPathString () {
382         String pathString = "";
383         for (int i = 0; i <= getPathLength(); i++)
384             pathString += ((char) (getAddr(i) + '0')) + ",";
385         return pathString;
386     }
387
388 }
389
```