Enhancing the Windows Network Device Interface Specification for Wireless Networking

Paramvir Bahl
Microsoft Research
bahl@microsoft.com

Abstract

Radical differences in channel characteristics coupled with end-node mobility make wireless networking sufficiently different from wired networking. Unfortunately, under current implementations of most operating systems a local area wireless network is treated as "just-another-wired-network" and is exposed to the higher layer networking protocols, operating system, and applications as an Ethernet technology. This then limits the system and applications ability to adapt to the changing channel conditions and prevents software vendors from developing adaptive and novel applications. In this paper we describe how the network architecture of a popular operating system, Windows, can be enhanced to support wireless-aware and mobility-aware networking and applications. We describe a couple of wireless systems that we have built and deployed, which would not have been possible without these extensions.

***Keywords**: Wireless network interfaces, programming for mobility, experimental architectures and implementation*

1 Introduction

Wireless networks are very different from traditional wired networks. When applications, operating systems, and network protocol stacks ignore this fact performance suffers and potentially useful functionality is not exposed.

Most modern operating systems, like Microsoft's Windows operating system, were designed to support wired networks whose device characteristics are more-or-less unchanging over time. In contrast, wireless networks have a more complex and dynamic environment that requires a richer interface between network devices, network protocol stacks, and applications than that required for wired networks.

An example of this can be seen in the extensions that have been defined in the *Windows Network Device Interface Specification* (NDIS) to support wireless infrared (IrDA) devices [1],[2]. NDIS supports the ability for higher-level network protocol drivers to query and set the state of a network interface card (NIC) by invoking appropriate `IoControl` operations on a set of *management objects*. In order to allow higher-level network protocol layers to interoperate with any IrDA NIC, a set of standard management objects has been defined that all IrDA NIC drivers must support. This set of management objects then define part of the low-level NIC device driver's interface.

A fairly minimal set of standard management objects has also been defined for wireless wide-area networks (wireless WANs) [2]. However, no set of standard management objects has yet been defined for other wireless device classes or for wireless devices in general. Consequently, different wireless hardware vendors have all defined their own proprietary set of device-specific management objects, making the creation of wireless-aware applications and network protocol stacks that can interoperate with arbitrary wireless devices extremely difficult.

Another important issue for operating systems designers has to do with ensuring correct operation of all applications and network protocol stacks in the presence of ones that wish to dynamically change the state of a wireless NIC – or NIC driver – in order to exploit some special aspect of wireless operation. This requires support for dynamic notification of device state changes to interested applications and to higher-level network protocol layers. It also requires that wireless-unaware applications and protocol stacks be insulated from these changes in a transparent manner.

In the Windows operating system, protocol drivers can register IndicateStatus functions on the managed objects of a device driver and NDIS will invoke the registered functions for a particular object whenever a device driver signals a status change on that object. Unfortunately, current network protocol stacks only register an IndicateStatus function for media connect/disconnect events. Outside the kernel, applications using the standard WinSock networking interface [3] don't even have the option of receiving notifications of state changes since WinSock has no notion of registering callbacks.

Dynamically changing device state could be masked by having low-level NIC drivers always return the device to some default state before responding to a request from a higher-level driver. However, some state changes in wireless devices are expensive in terms of time delay and/or power consumption. Hence this approach can result in significantly poorer performance as well as significantly higher power consumption than would be possible if applications and higher-level drivers could take into account the current state of a wireless device. Consequently, a means must be provided to inform NIC drivers when to maintain the illusion of a default state for network connections being handled by "legacy" protocol stacks and when to expose a changed device state for

network connections being handled by wireless-aware protocol stacks and applications.

The point of this paper is to introduce enhanced interactions that are needed to exploit wireless networks and to describe how such enhanced interactions can be enabled within the Windows networking architecture. Section 2 briefly describes the relevant aspects of the NDIS, WinSock and Windows Management Instrumentation (WMI) interfaces. Section 3 characterizes the NDIS specific functionality that we have built as part of an enhanced networking interface. Section 4 describes a networking architecture, based on these extensions combined with WMI support for notifying applications of relevant state changes. Section 5 describes two examples of how we exploit the enhanced functionality to build a location tracking and determination system and mobility support in a network access application. In Section 6, we discuss additional areas of wireless networking that could benefit from this enhanced interface and finally in Section 7 we present a summary and draw conclusions for this work.

The extensions discussed in this paper are based on our development experience from having built several wireless networking infrastructure services that are deployed both in our research lab. and in public buildings (*see* http://www.mschoice.com)

2 Overview of Window's NDIS, WinSock, and WMI Interfaces

In order to explain the enhancements we propose to the Windows networking architecture we first provide an overview of the relevant Windows operating system APIs.

2.1 Network Driver Interface Specification (NDIS)

NDIS describes the interface by which one or more NIC drivers communicate with one or more underlying network interface cards, with one or more overlying protocol drivers, and with the operating system. NDIS supports the following interactions:

- A NIC driver to receive a network packet from any one of several upper layer protocol drivers for transmission on the network through any one of several network adapters.
- A NIC driver to accept a network packet from any one of several underlying network adapters, and pass it up to one or more upper layer protocol drivers that are interested in receiving it.
- An upper layer protocol driver to set specific configuration parameters for a network adapter or a NIC driver by means of an IOControl function.
- An upper layer protocol driver to query a NIC driver for specific configuration and statistics from the underlying network adapter or the NIC driver by means of an IOControl function.
- A NIC driver to asynchronously inform an overlying driver of changes in network status or NIC status by means of an IndicateStatus notification function.

NDIS defines a fully abstracted environment for NIC device driver development. For every external function that a NIC driver needs to perform, it can – and must – rely on NDIS intermediary routines to perform the operation. This includes the entire range of tasks performed by a NIC driver, from communicating with protocol drivers, to registering and intercepting NIC hardware interrupts, and communicating with underlying NICs through manipulating registers, port I/O, and so forth.

Each NDIS driver also contains its own management information block (MIB), an information block in which the driver stores dynamic configuration information and statistical information that higher-level drivers can query or set. Each information element within the MIB is referred to as an *object* and is referred to by means of an *Object Identifier (OID)*. If a higher-level driver wants to query a particular aspect of a NIC or its driver, it does so by invoking a query IoControl function with the relevant OID and receives back appropriate object-specific data. Similarly, to set some aspect of the state of a NIC or its driver, a set IoControl call is made for the relevant OID, which will include a buffer of object-specific data that the NIC driver will act on. Thus, the set of capabilities that can be queried or set for a particular NIC depends on the set of published OIDs and associated object-specific data definitions that the NIC driver exports.

2.2 WinSock

Sockets are a general-purpose networking interface, supported by most operating systems. WinSock is designed to be compatible with the Berkeley Sockets Distribution (BSD) specification while performing well on Windows-based operating systems.

The relevant aspects of the WinSock API for this paper are its *Ioctl* calls, which are used to retrieve and set operating parameters associated with a socket. These calls pass in an integer command argument and a pointer to a data buffer; the data buffer is used to pass in command-specific data as well as receive back command-specific data.

Socket drivers examine the integer command argument in order to decide what action to take for a given *Ioctl* call. In theory, *Ioctl* calls could be used to allow applications to query and set wireless NIC characteristics. However, this would require that the network protocol stacks involved make the appropriate corresponding NDIS IoControl calls on their underlying NDIS device drivers. Currently no Windows network protocol stacks do this, in good part because they would have to understand and translate to a different set of NDIS IoControl operations for each different NIC.

Another important aspect of the *Ioctl* API is that there is no notion of setting a callback function for anything other than notification of address/route changes. Hence applications wishing to learn about any underlying state changes in a socket connection must either poll for them or allocate a separate thread to a blocking call that will return only when a relevant state change has occurred. The latter approach is only possible if the underlying socket driver supports an appropriate *Ioctl* command, which none currently do.

2.3 Windows Management Instrumentation (WMI)

The Windows Management Instrumentation (WMI) technology [4] is an implementation of the Desktop Management Task Force's (DMTF) Web-Based Enterprise Management (WBEM) initiative for Microsoft Windows platforms. It extends the Common Information Model (CIM) to represent management objects in Windows management environments. The Common Information Model, also a DMTF standard, is an extensible data model for logically organizing management objects in a consistent, unified manner in a managed environment.

WMI offers a powerful set of services, including query-based information retrieval and event notification. These services and the management data are accessed through a Component Object Model (COM) programming interface. A scripting interface is also provided.

The WMI architecture consists of following components:

- *A management infrastructure*. This includes the user-level CIM Object Manager, which provides applications with uniform access to management data and a central storage area for management data.
- *WMI providers*. These function as intermediaries between the CIM Object Manager and managed objects. Using the WMI APIs, providers supply the CIM Object Manager with data from managed objects, handle requests on behalf of management applications, and generate event notifications.

WMI extensions to the Windows Driver Model (WDM) provide the basis for hardware instrumentation in Windows environments. The WMI extensions to WDM are kernel-level instrumentation technologies for the Windows platform that provide access to driver data and events. WMI interactions with NDIS device drivers use the existing paradigm of object identifiers. The code provided by the NDIS library class driver translates between WDM extension I/O request packets (IRPs) and corresponding query/set operations on appropriate object identifiers.

Unfortunately, higher network protocol layers in the OS kernel cannot use WMI in order to communicate with lower-level device drivers. This is because WMI queries and event notifications all go through the user-level CIM object manager. Thus, WMI is only useful for allowing applications, but not network protocol stacks, to interact with NIC device drivers.

With the above as background, we now describe the extensions we have built and a device programming architecture that combines our extensions with WMI support for notifying applications of relevant state changes.

3 Functional Enhancements to NDIS

In the previous section we explained that the interactions with a NIC's device driver can be categorized according to whether they involve a static query that will occur at set-up time, a dynamic query that will occur during device operation, and an update command that will occur either during set-up or device operation. In this section we summarize the capabilities that we have built categorized in this fashion. We justify these in Section 5 and 6 where we illustrate their use with examples and discussions.

3.1 Static Query Routines (information usually retrieved at set-up time)

The following information about a wireless NIC's capability is retrievable via an OID call:

- raw bit rate supported (bps)
- support for link-layer ACKs (bi-directional routes)
- capability to control forward error correction (FEC)
- support for handoffs between access points (APs)
- ability to dynamically adapt transmission power
- method of media acquisition used

3.2 Dynamic Query Routines (information retrieved during operation)

The following information characterizing current state of the wireless node's current point of attachment (access point/base station/control point) is retrievable via OIDs:

- signal strength (dbW)
- noise floor at transmitter (dB)
- MAC address of access point
- Access point identifier, and
- frequency of beacon signals (Hz)

In addition we built the ability for a programmer to retrieve a list of all potential points of attachment that the wireless NIC can currently hear. Each list element includes the information enumerated above for all potential points of attachment.

For every specified incoming packet the following state information relating to its arrival is available upon query:

- signal strength (dBW),
- noise floor at transmitter (dB),
- noise floor at receiver (dB),
- MAC address of the transmitter, and

link-layer retransmission count.

For every specified outgoing packet the following state information relating to its transmission is available via an API call:

- transmission power
- retransmission count, and
- delay involved in acquiring the medium for transmission.

In addition to per-attachment point and per-packet data, the following running average statistics are retrievable from the wireless NIC:

- throughput (bps)
- bit error rate (BER)
- medium acquisition delay (in milli-seconds), and
- percentage of beacons received.

So that applications and higher layers of network protocol stacks can avoid the need to poll, the ability to register callbacks for the following events is provided:

- Failure to acquire the medium for transmission of a packet
- Receipt of an ACK or failure to receive an ACK from the receiving node after a timeout period
- Attachment to a specified or automatic new point of attachment

3.3 Setting Attributes (during set-up and operation)

Applications and/or network protocol stacks can programmatically control the following aspects of the wireless NIC behavior:

- Attach to a specified access point (access point/base station/control point).
- Force the querying of some or all access points currently reachable.
- Enable/disable FEC.
- Adjust power-levels and control hibernate/resume, and standby/resume NIC activity

We note here that most of the functionality mentioned in this section is generally available in the wireless NICs propriety firmware. Our intension has been to build an API that provides hooks to these capabilities so systems can take advantage by incorporating the latest of research in the area (see Section 6). Consequently, we have built these extensions in the form of object identifiers (OID) in NDIS. Table 1 contains a partial list of the general objects proposed and built by us. *Query* and *Set* refer to device driver calls and *Indication* refers to use of object as a parameter to OID_WL_GEN_INDICATION request object. For a complete description we refer the reader to [6].

In the next section we discuss how these extensions can be used with WMI to provide flexible programmability for wireless-aware applications.

Name	Query	Set	Indication
OID_WL_GEN_INDICATION_REQUEST		M	
OID_WL_GEN_PACKET_TX_STATUS	M		
OID_WL_GEN_DEVICE_INFO	M	M	
OID_WL_GEN_NETWORK_TYPE	M	M	
OID_WL_GEN_NETWORK_ROLE	M	M	
OID_WL_GEN_NETWORK_MODE	M	M	
OID_WL_GEN_CHANNEL	M	M	
OID_WL_GEN_NETWORK_ID	M	M	
OID_WL_GEN_NEIGHBORING_ACCESS_POINTS	O		
OID_WL_GEN_CURRENT_ACCESS_POINT	O		
OID_WL_GEN_AUTHENTICATION_STATUS	O		O
OID_WL_GEN_HEADER_FORMAT	O	O	
OID_WL_GEN_PERMANENT_ADDRESS	O		
OID_WL_GEN_CURRENT_ADDRESS	O	O	
OID_WL_GEN_SUPPORTED_RX_MODES	O	O	
OID_WL_GEN_RX_SIGNAL_STRENGTH	O		O
OID_WL_GEN_RX_SS_THRESHOLD	O	O	
OID_WL_GEN_CHANNEL_QUALITY	O		O
OID_WL_GEN_RADIO_STATUS	O	O	
OID_WL_GEN_SUPPORTED_POWER_LEVELS	O	O	
OID_WL_GEN_POWER_SAVING_MODE	O	O	
OID_WL_GEN_SUPPORTED_LINK_SPEEDS	O	O	
OID_WL_GEN_FRAGMENTATION_THRESHOLD	O	O	
OID_WL_GEN_SUPPORTED_FEC_LEVELS	O	O	
OID_WL_GEN_SUPPORTED_ENCRYPTION_TYPES	O	O	O
OID_WL_GEN_PERFORMANCE_STATISTICS	O	O	

Table 1: The general wireless specific objects that we built into NDIS. "M" means that support for the object is mandatory, and "O" means its optional.

4 Wireless-Specific Application Programming Interface

Currently an application or network protocol stack wishing to exploit aspects of a wireless NIC that are not part of the generic Ethernet model of network devices must do so in an ad-hoc manner. Since each vendor exposes the wireless-specific features of their NIC using a different set of NDIS OIDs and associated data definitions, applications and network protocol stacks must contain different code to interface with each type of wireless NIC they employ. In order to avoid these vendor/hardware-specific dependencies we propose that a standard set of NDIS OIDs and related data definitions as defined in section 3 and [6] be used. Later we will describe how some of these extensions are used in wireless–specific applications that we have built.

While standardized set of wireless NDIS objects allow network protocol stacks to interface with arbitrary wireless NICs directly, in order to give applications the same capability as the protocol drivers, the relevant NDIS objects must be exported to user-level processes as well. The two ways one could do that is either via the WinSock or the WMI interface.

The primary disadvantage of going through WinSock is that it does not support the notion of event notification. Furthermore, it would require that all network protocol stacks be able to appropriately translate and pass through the relevant query and set operations to their low-level NDIS drivers. In contrast, the WMI interface supports asynchronous event notification and the WDM extensions to WMI provide direct access to NDIS objects from WMI. WMI is also now the recommended way for applications to access all system state information in the Windows world. For these reasons we argue that the standard way that applications should interact with the wireless-specific aspects of networks should be via WMI.

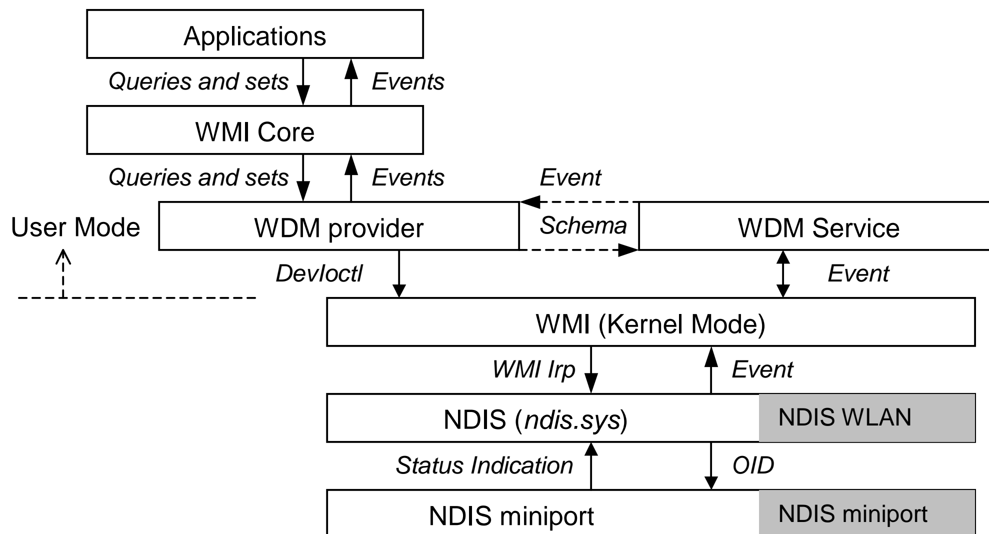

Figure 1:Provider Architecture for WMI along with the NDIS wireless extensions.

In addition to defining standardized wireless NDIS objects and making them available via NDIS and WMI it is also necessary to ensure that applications and network protocol stacks continue to operate correctly in an environment where device characteristics can be dynamically changed out from under them. Currently, if an application or higher-level network protocol layer changes a fundamental feature of a NIC through an IoControl operation on one network connection that change will not be noted by any code managing other network connections. Thus, for example, if a NIC is currently in "ad-hoc" mode – and hence listening for transmissions from any source – and is instructed to switch to "infrastructure" mode – where it listens only for transmissions from a particular base station – then any network connection(s) that depended on the NIC being in ad-hoc mode may behave incorrectly since they will be unaware of the sudden restriction that has occurred. To solve this dilemma several things need to be done. To begin with, high-level network protocol drivers need to register IndicateStatus functions for *all* state changes that might occur in a NIC and low-level NIC drivers must signal all state changes that occur. Wireless-aware protocol stacks and applications (via WMI events triggered by IndicateStatus functions) will thereby be suitably notified.

Unfortunately, wireless-unaware protocol stacks cannot be expected to register IndicateStatus functions for state changes they don't expect. Consequently low-level NIC drivers also need to maintain the illusion of an unchanged state for any network connections being managed by these protocol stacks. This will require that they remember what the appropriate state is for each such connection and that they change states whenever switching between actions requested by different connections. Another wrinkle that must be dealt with is that the NIC driver must be sure to be listening to all relevant transmission sources, even if some network connections have requested a more restrictive listening scheme.

At the same time, for applications and network protocol stacks wishing to optimize performance and power consumption by taking advantage of current, changed device state, there needs to be a means by which low-level NIC drivers can be told to *not* reset NIC state before acting on some request. This can be achieved by passing in an appropriate flag parameter when invoking NIC driver actions. If the low-level NIC driver doesn't see this flag parameter then it assumes that it is dealing with a "legacy" client for whom the illusion of an unchanging default state must be maintained.

Unfortunately, resetting a NIC to a default state in support of wireless-unaware network connections may defeat some of the optimizations that wireless-aware applications and protocol stacks might wish to employ. However, while this may prevent full exploitation of potential wireless performance and power management in the near term, we expect that in the long run it will be a moot point since there are not that many network protocol stacks in existence.

Figure 1 shows the WMI architecture along with the NDIS. The extensions we have built are illustrated as shaded boxes. The NDIS miniport is the hardware device driver that the hardware vendor provides.

In order for extensions to be meaningful we extended both NDIS and the miniport driver to offer the functionality described in Section 3.

We now discuss two applications that we have built using the extensions just described.

5 Examples of Wireless-Specific Applications

The point of the paper so far has been that wireless networks are significantly different from wired networks

and that they can offer functionality that is not possible with wired networks. Below we discuss two wireless systems we have built called RADAR and CHOICE, which use our extended programming interface to offer new and useful functionality.

5.1 RADAR [14]

RADAR is an in-building location-determination system in which radio-frequency (RF) wireless LAN enabled mobile nodes compute their location and use this information to inform other nodes of their location, and to determine which network resources (e.g., printers, scanners, etc.) are within close proximity. The basic system uses signal strength measurements of beacon signals that are broadcast from neighboring APs in conjunction with a pre-determined Radio Map of the building to determine location of the user. The signal measurements are extracted from the mobile's wireless network interface. The matching of real-time signal strength measurements to measurements stored in the Radio Map occurs either on the mobile computer (if privacy is a concern) or on the backend server (if mobile's power is an issue). Figure 2 illustrates the basic concept behind RADAR.

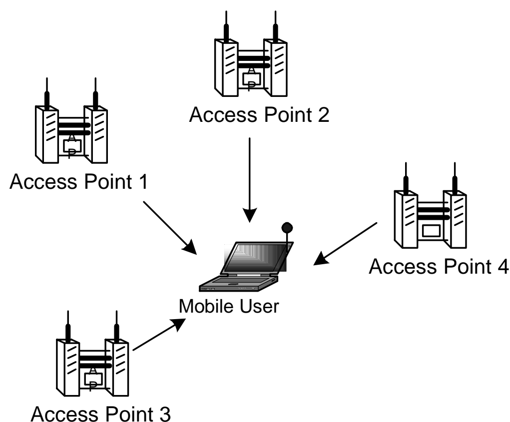

Figure 2: Beacon packets from neighboring APs are used to estimate the location of the mobile computer.

We have deployed RADAR in multiple buildings on our campus using two different wireless LAN technologies (Lucent's WaveLAN [5] and Aironet's 4800 series [7]) and two widely used operating systems (FreeBSD and Windows 2000).

The programming interface available to us in both operating systems did not provide the necessary hooks that we needed to build this location-determination system. In particular, RADAR requires knowledge of the APs identity, which it can "hear", and the ability to switch channels[1]. These requirements are very specific to wireless networks and are not available as part of the programming interface for wired networks.

In FreeBSD, we found a WaveLAN device driver that provided at least some of the functionality we required (possibly a anonymous contribution from the research community). Since we did not find the required functionality in the Windows operating system device driver, we extended NDIS to extract the signal strength information from the AP beacons. In particular, we added IoControl definitions specified in Section 3.2 and the corresponding functions in the hardware device driver exposing the wireless specific features of the underlying network.

We created a software library called WiLIB to provide application-level control of the wireless hardware (see Figure 4) while maintaining context as explained in Section 4. Our high-level objective for WiLIB has been two-fold: first, we want to enable the creation of novel user-level applications such as RADAR, and second, we want the ability to monitor and dynamically configure the hardware so that wirelessly connected systems can benefit from the latest research in adaptive system programming which relies on knowing the state of the underlying communications channel.

To create WiLIB, we relied on the NDIS extensions. In addition to gathering signal strength information and switching channels, we wanted to avoid the need to poll the wireless NIC to determine if the mobile had changed association with a particular AP – signaling that it had physically moved. To avoid overburdening the system with irrelevant processing, a mechanism for installing MAC based filters was also built.

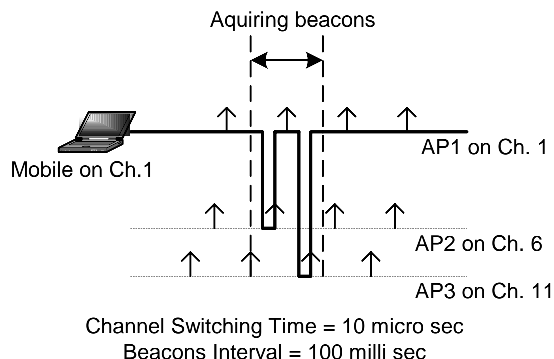

Figure 3: Mobile acquiring beacon packets from neighboring APs.

For RADAR, to work we incorporated the following functionality in WiLIB built on top of our NDIS wireless LAN extensions:

(1) WiLIB could configure the wireless NIC to operate on a specified channel.

(2) For every incoming packet from a specified MAC address, WiLIB could retrieve the packets received signal strength, noise floor at the transmitter, and noise floor at the receiver.

[1] To maximize system capacity, neighboring APs in a wireless LAN generally operate on different channels (a consequence of the classical frequency re-use concept in cell-based networks [8]). So the mobile cannot hear beacons from all APs within its range unless it tunes itself to those channels one by one.

Using the above two, we were able to generate a list of all APs that the mobile could hear. First, RADAR would put the wireless NIC in a promiscuous mode gathering the 48-bit MAC address of the AP and the beacon packets from the AP on each channel. Then it would build up a list of all neighboring APs, which would be used to find a best match to pre-recorded signal strengths tuples in the Radio Map.

Thus RADAR is a case study of an application that confirms that operating systems need to specifically provide support for wireless network programming.

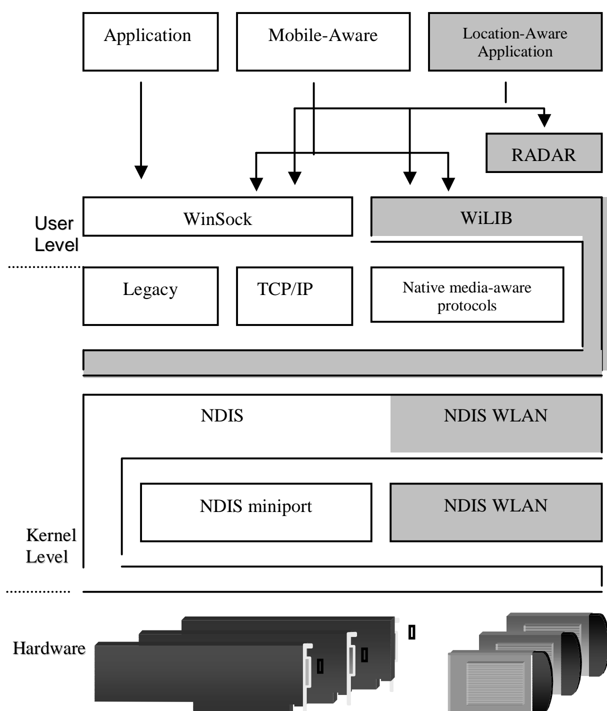

Figure 4: Placement of RADAR, WiLIB and our NDIS wireless extensions in relation to the networking protocols in Windows 2000.

We now describe another application that has been built and deployed that too requires wireless network programming support.

5.2 The CHOICE Network

In addition to RADAR, we have built and deployed a wireless network in a popular local mall. This network is called the CHOICE network and it authenticates users via a globally available database, provides local authorization, and then based on a pre-configured policy manages their use of the shared network resources (see http://www.mschoice.com).

While deploying our network we realized that the usage and service options of a pubic network generally differ from a private (enterprise and home) network and consequently, the two networks are often configured differently. The existence of such differing networks motivated our need to improve support and management of nomadic users who frequently roam between them. Examples of different configurations include difference in the wireless network's Service Set ID (or SSID) and differences in the use of the shared key authentication methods such as the Wired Equivalency Protocol (WEP) [23]. Due to these differences, a system that proposes to support true node mobility must allow the client devices to dynamically configure themselves and adapt to the local network configuration [24].

We built an elaborate beaconing system for determining the presence or absence of the CHOICE network. However, we still needed the ability to query the wireless NIC for its SSID and to set it to the appropriate SSID without any user intervention. Similarly, for the case of WEP, we found that while many private networks use WEP most public networks don't. This is because management of per-user keys can be difficult in a public setting. Therefore. we needed to query the wireless NIC to see if WEP was enabled and whether it was associated to an access point. If WEP was enabled and the NIC was not associated with an AP, the client would disable WEP and then try to associate with an AP.

Interfaces such as the one just described tend to be highly standard specific. In this case the notion of SSID and WEP is specific to the IEEE 802.11 standard [23]. Although we build OIDs to manage SSID's and WEP, we do not discuss in great detail standard the functionality specific to particular standards. Suffice to say wireless standards are different from wired standards and if one or two become prevalent (for example IEEE 802.11, Bluetooth or Home RF), then general OID's will have to take into account these standards as well. [24]

6 Additional uses for the Enhanced Wireless Interface

In the previous section we showed how wireless specific NDIS extensions are used in two novel applications that we have built. In this section we continue along the same theme and describe additional uses of these extensions, specifically to get better network performance and to enable many more novel services. Our assertion is that the scenarios we describe below are possible only if the suitable programming model for wireless NICs is exposed to networking protocol stacks and applications. Most of the extensions we propose are either already supported, in the form of propriety programming interfaces, by a majority of the mainstream wireless NICs or can be supported by them without too much effort.

6.1 Performance

Wired networks such the Public Switched Telephone Networks (PSTN) are characterized by low delay, error rates that are typically better than 10^{-6}, and channel conditions that remain constant with time. Applications, system services, and networking protocols have been designed and optimized for such "clean" channels and hence work reasonably well in such environments. Wireless radio networks incur higher bit error rates (typically in the order of 10^{-2}) over a channel whose characteristics vary unpredictably and severely. The

unpredictability is caused by both random and burst errors resulting from propagation phenomena such as multi-path, shadowing, path loss, noise and interference from other sources, all of which have a multiplicative effect on the transmitted signal, causing it to deteriorate. In these environments classical networking protocols perform poorly [9],[10].

There are two approaches to alleviating problems that occur due to time varying channel conditions. The first is to build error control protocols and mechanisms that hide channel imperfections from higher layer protocols. The second approach is to build adaptive systems and applications that dynamically change behavior to adjust to the channel conditions. The two approaches are complimentary and can generally be combined to produce a system that performs well by responding to the changing environment while saving the user from the effects of it. Researchers have shown that such hybrid schemes work well in bandwidth-poor, error-prone RF environments [13].

Masking channel imperfections is typically done by means of two distinct error control techniques: forward error correction (FEC) and automatic repeat request (ARQ) [11],[12]. FEC-based approaches alleviate the random error induction problem but aggravate bandwidth problems since several more redundant bits are added to the transmitted data packets. ARQ-based schemes improve error recovery against burst errors but aggravate latency (jitter) problems, which can render a time-critical bit stream useless.

Wireless NIC hardware vendors usually provide error control at the firmware and device driver level. However the techniques they employ are generally limited to correcting random bit errors only. For correcting burst errors most hardware designers assume that a higher layer networking protocol (e.g. the transport layer) will employ an ARQ-type technique. This requires that information about packet acknowledgements, loss, and retransmission be made available to the higher layers of each network protocol stack.

In any case, a "black-box" error correction approach cannot yield as good performance as a system that also employs adaptive techniques at the applications and network protocol level. For example, if a NIC supports programmable FEC then the network protocol programmer might choose to dynamically adjust how much FEC to use. When the bit error rate (BER) of the channel is low, he or she might decide to save bandwidth by not using FEC and relying only on ARQ, whereas when the BER is high he or she might apply FEC to every packet.

Adaptive techniques are also necessary to enable coordination of behavior among multiple network connections that might otherwise interfere with each other in a performance-degrading manner. For example, changing the wireless channel to use on a NIC may take a considerable amount of time (on the order of tens of milliseconds). If, among other things, a system is participating in an ad-hoc network then it may need to forward out on one channel some of the packets coming in on another channel. Knowing which channel the NIC is currently set to can allow the packet router to decide whether to forward a packet now or buffer it for forwarding – along with possibly other buffered packets – sometime later.

If the ad-hoc network router is a user-level process then information about current NIC state and notifications about changes in NIC state will need to be made available not just to higher network protocol layers in the OS kernel, but also to user-level applications.

No matter which approach a system designer chooses – a hybrid FEC/ARQ based approach, an adaptive applications and systems approach, or a combination of the two – he or she has to be able to determine the channel conditions, know the capabilities of the wireless NIC, and dynamically adjust them from above the lowest-level device driver. Based on what has been explored in the research literature so far, we would like to see the following capabilities being made available to applications and network protocol stacks from wireless NIC device drivers:

- A running BER average
- Programmable FEC with enable/disable ability
- Re-transmission count and delay reading in acquiring the medium for specified outgoing packets.
- Validation of support for link layer ACKs.
- Association of a callback with the receipt of an ACK or failure to receive an ACK.

6.2 Energy Conservation

A wireless network transceiver typically uses 15 to 30% of the power of a typical mobile computer (the display being the only part that uses more power) [16]. Consequently, battery-life continues to be a major concern for the designers of mobile devices and wireless networks. For example, an IEEE 802.11 [23] compliant wireless NIC operating in the 2.4 GHz ISM band uses over 1.8 Watts of power during transmit and about 1.6 Watts during receive; it can deplete the entire battery of a handheld PC or PDA in minutes!

Research has shown that all levels of the communications protocols stack can contribute to energy savings by exploiting knowledge of the channel condition, network availability and traffic characteristics of a wireless connection [17]. For example, a mobile node may implement a periodic wake-up pattern for listening for incoming transmissions in order to avoid keeping its receiver on all the time. Alternatively, it may instead employ a wake-up pattern adapted to match current traffic characteristics, as described in [18]. Another way to reduce receiver power consumption is to create a system where the access point can send wake-up command messages to a mobile unit whose receiver is in low-power mode on an as-needed basis [19]. Once woken up, applications and higher levels of the network protocols would decide when a current sequence of communications

is finished and the NIC receiver should return to low-power "hibernation" mode.

To conserve transmission power consumption, a mobile node can either adjust the power it uses for sending any given packet or it can control how many packets to send out in the first place. The choice of transmission power to use might be determined by things such as how far away a access point is and what kinds of packet retransmission rates are occurring [20],[21],[22]. Alternatively, if applications can determine how much power has been necessary to transmit recently-sent data packets then they can decide whether or not to modify their communications patterns accordingly. For example, a multi-media application might choose to trade off lower power consumption against reduced fidelity when transmission power requirements are high.

Trading off latency for batch delivery of multiple packets can also reduce power consumption by amortizing the overhead of acquiring a shared wireless transmission channel by means of things like "request-to-send" packets. Consequently, knowledge of application-specific latency tolerances could be used by each network connection to decide whether it should send a packet immediately or buffer it in the hopes of being able to batch it with other packets from its own or other network connections. Packet batching could be done within the low-level NIC driver, but this would require that application-specific latency requirements be imparted to the driver for each packet. The alternative is to let higher levels of the system buffer their packets and send them down for transmission when notified that a transmission opportunity is available.

To support these kinds of sophisticated power management strategies requires that a wireless NIC driver export the following methods:

- To determine if the NIC allows dynamic adaptation of transmission power.
- To retrieve for every specified outgoing packet the transmission power used.
- To adjust power-levels of the wireless NIC.
- To hibernate/resume and standby/resume NIC activity.

6.3 Mobility Management

Mobility management entails both network connection management and location management. In a wireless network, as mobile nodes move, segments of their network connections have to be torn down and re-established with a frequency that corresponds to the speed of the mobile node. Meanwhile data integrity in terms of packet sequence preservation, duplicate packet prevention, and packet loss avoidance has to be maintained.

Standards like IEEE 802.11 and HomeRF [25] do not explicitly specify how or when hand-offs should occur, leaving it to each hardware vendor's discretion to develop their own method. Given the right hooks, it should be possible to decouple handoff methods from the hardware, thereby enabling software vendors to develop value-added "smart" hand-off techniques. For example, sophisticated algorithms that employ predictions of a user's trajectory can be devised to reduce the probability of blocking. Similarly, techniques for providing hot-spot traffic relief and reducing congestion, and for improving resource utilization can be built [26],[27].

To enable mobile-assisted, software-based handoffs wireless NIC drivers should export information about what access points are nearby and what the wireless medium characteristics are for communicating with each access point. In particular, the following capabilities should be exported:

- A method to determine if the wireless NIC supports programmable handoffs between base stations.
- A method to obtain the current point of attachment (base station/control point) along with the following information: signal strength, noise floor at transmitter, MAC address, base station identifier, and frequency of beacon signals.
- A method to force the wireless NIC to connect to a specified base station.

7 Summary and Conclusions

Wireless networks are very different from traditional co-axial and fiber-optics based networks. They have significantly different error rates and types of errors, and they require a significantly richer interface to describe and manipulate the characteristics – such as current signal strength, currently attached base station, and current power level – that govern their behavior. Wireless networks also offer the possibility of enabling a new class of applications and services that wired networks cannot offer.

When applications and network protocol stacks ignore the differences that exist between wireless networks and traditional wired networks then performance of applications and systems suffers significantly and the ability to run novel applications and services is diminished. To enable wireless technology to reach its full potential the Windows networking architecture must be extended to enable a richer set of interactions between network devices, higher-layered network protocols, and applications. In this paper we have described a set of enhancements to the Windows NDIS API that, combined with application reliance on WMI, will enable the exploitation of most, if not all, the additional wireless functionality and performance that is inherent in the underlying communications hardware.

The changes we propose do not radically change the Windows networking architecture. For the most part, they merely seek to standardize the means by which information about wireless device and connection state is communicated among different layers of device drivers, as well as between the operating system and applications. As a result, ad-hoc, vendor-specific interactions would be

replaced with standardized NDIS and WMI calls. These would work the same across all wireless network devices.

Acknowledgements

We thank our colleagues who helped crystallize our ideas and designs. In particular we are grateful to Gavin Holland for implementing most of the extensions described in this paper. We also thank Pradeep Bahl, Tom Fout, and Stephen Hui from the Windows Networking Product Team and Marvin Theimer from Microsoft Research for the time they spent discussing and reviewing these extensions.

References

[1] P G. Viscarola and W. A. Mason, *Windows NT Device Driver Development*, Open System Resources, 1999

[2] MSDN Online - ***http://msdn.microsoft.com/***

[3] B. Quinn,and D. Shute, *Windows Sockets Network Programming*, Addison-Weley Advanced Windows series, 1996

[4] WMI: Windows Management Instrumentation Technology, http://www.microsoft.com/hwdev/ WMI/

[5] RoamAbout 915/2400 DS/PC Card and ISA Network Adapter: Installation and Configuration, *Digital Equipment Corporation*, April 1996

[6] P. Bahl and G. Holland, "Enhancing the Windows Network Device Interface for wireless networking, Part II", MSR Technical Report MSR-TR-2000-84, August 2000

[7] Developer's Reference Manual: PC4500/PC4800 PC card Wireless LAN Adapter, *Aironet Wireless Communications Inc*. 1999

[8] T. S. Rapport, Wireless Communications – Principles and Practice, *IEEE Press*, 1996

[9] H. Balakrishnan, "Challenges to Reliable Data Transport over Heterogeneous Wireless Networks," Ph. D. Thesis, University of California at Berkeley, Berkeley, California, USA, August 1998.

[10] P. Bhagwat, P. Bhattacharya, A. Krishna, and S. k. Tripathi, "Enhancing Throughput Over Wireless LANs Using Channel State Dependednt Packet scheduling," in IEEE INFOCOM '96, April 1996

[11] H. Liu, "Real-Time Video Transmission and Multimedia Services over Wireless Networks," Ph. D. Thesis, University of Pennsylvania, Philadelphia, PA, USA, October 1997

[12] D. A. Eckhardt and P. Steenkiste,"Improving Wireless LAN performance via Adaptive Local Error Control, ", Int. Conference on network protocols, pp. 327-338, 1998

[13] M. E. Zarki, and S. Gupta, Special issue on "Error Control in Wireless Packet Networks," of the *Mobile Networks and Applications Journal,* vol. 2, no. 2 (1997): 165-224

[14] P. Bahl, and V. N. Padmanabhan, "RADAR: An In-Building RF-based User Location and Tracking System," in the *Proceedings of IEEE INFOCOM '2000*, Tel-Aviv, Israel, March 2000

[15] S. Tekinay, "Wireless Geolocation Systems and Services," Special Issue of the *IEEE Communications Magazine*, April 1998

[16] E. P. Harris, et. al. "Technology directions for portable computers," *Proceedings of the IEEE*, vol. 83, no. 4, (April 1995): 636-658

[17] M. Zorki, Special Issue on "Energy Management in Personal communications and Mobile Computing," of the IEEE Personal Communications Magazine, vol. 5, no. 3, (June 1998)

[18] P. Bahl, "ARMAP - An Energy Conserving Protocol for Wireless Multimedia Communications," *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '98)*, Boston, Massachusetts, USA (September 1998)

[19] A. R. Koelle, S. W. Depp, and R. W. Freyman, "Short-range Radio-Telemetry for Electronic Identification, Using Modulated Backscatter," Proceedings of the IEEE, Vol. 63, No. 8, pp. 1260-1, August 1975

[20] R. Kravets and P. Krishnan, "Power Management Techniques for Mobile Communication, " ACM MobiCom '98, pp. 157-168 October, 1998

[21] J. M. Rulnick and n. Bambos, "Mobile Power Management for Maximum Battery Life in Wireless Communication Networks," IEEE INFOCOM '96, pp. 443-50, March 1996

[22] M. Zorzi, R.R. Rao, "Error Control and Energy Consumption in Communications for Nomadic Communications," IEEE Transactions on Computers, Vol. 46, no. 3, pp. 279-89, March 1997

[23] IEEE 802.11b/D3.0, "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification: High Speed Physical Layer (PHY) Extensions in the 2.4 GHz Band, " 1999

[24] A. Miu, and P. Bahl, "Dynamic Host Configuration for Managing Mobility Between Private and Public Networks, " *Microsoft Technical Report, MSR-TR-2000-85* (August 2000)

[25] J. Lansford and P. Bahl, "The Design and Implementation of HomeRF: A Radio-Frequency Wireless Networking Standard for the Connected Home," to appear in *The Proceedings of the IEEE* (October 2000) (Invited Paper)

[26] S. Pope, "Application Support for Mobile Computing," Ph. D. Thesis, Computer Laboratory, University of Cambridge, U.K, October 1996

[27] T. Liu, P. Bahl, I. Chlamtac, "Mobility Modeling, Location Tracking, and Trajectory Prediction in Wireless ATM Networks," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 6, (August 1998):922-936.

[28] M. Satyanarayanan et. al., "The Coda File System," Carnegie Mellon University, Computer Science Department, http://www.coda.cs.cmu.edu/, 1994 – present