The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

# Implementing IPv6 for Windows NT

Richard P. Draves, Brian D. Zill
*Microsoft Research*
Allison Mankin
*University of Southern California*

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

# Implementing IPv6 for Windows NT

Richard P. Draves,[1] Allison Mankin,[2] Brian D. Zill[1]

[1]*Microsoft Research*
*One Microsoft Way*
*Redmond, WA 98052*
*richdr@microsoft.com*

[2]*USC/ISI*
*4350 North Fairfax Drive*
*Arlington, VA 22203*
*mankin@isi.edu*

## Abstract

We have created a publicly-available implementation of IPv6 for Windows NT. Because we have made our source code available, we hope that our implementation can serve as a base for networking research and supply sample code for other implementations. In this paper we describe our solutions for several problems that any network protocol implementation for Windows NT will encounter. Based on our experience, we also comment on the utility of access to the source code for the Windows NT product.

## 1. Introduction

This paper reports our experiences developing a network protocol stack for Windows NT. We have created a prototype implementation, known as MSR IPv6, of IPv6 for Windows NT. We have released the implementation and its source code publicly, for testing, research, and educational purposes [10]. Our implementation should prove useful to people experimenting with IPv6 and to people wishing to use Windows NT as a platform for networking research or education.

IPv6 [3] is the next version of the Internet Protocol; it is an active effort within the Internet Engineering Task Force (IETF). IPv6 primarily solves scaling problems with the current version of the Internet Protocol (IPv4), but it also introduces many other major and minor architectural improvements. Most notably, IPv6 addresses have 128 bits (16 bytes) [7]. We will introduce other salient aspects of IPv6 throughout the paper, as they are relevant. Numerous vendors have pre-release IPv6 implementations and there are several free, publicly available implementations for BSD and Linux variants [9]. Our implementation is the first free, publicly available implementation for Windows NT.

We started this project at Microsoft Research in late 1996 primarily as a learning experience: we wanted to learn more about the Internet Protocol and this step in its evolution, and we wanted to learn more about Windows NT internals. We also hoped that our efforts might help bootstrap a Microsoft product implementation of IPv6. (We can not say anything further about Microsoft's product plans or schedules.) As we made progress, we realized that a public release, including source code, would be valuable for the community. USC/ISI East joined the project in December 1997. Our first public release was March 24, 1998.

Overall, Windows NT has been a good platform for protocol development. It accommodates new protocols, loadable at run-time, with great ease. The kernel debugger provides a good source-level debugging environment. Microsoft's Network Monitor tool, for capturing and viewing packets, was also very useful for debugging. We use Windows NT 4.0 for our development, but our MSR IPv6 implementation runs equally well on current versions of Windows NT 5.0.

We did come across several implementation challenges that are not specific to IPv6 and would be faced by any protocol implementation for Windows NT. Among the challenges for an efficient implementation are how to handle received packets given the multiplicity of ways to receive them, how to "pull-up" fragmented packet buffers into a contiguous buffer, and how to add link-layer headers when sending packets. In Section 4 we examine these and other problems and our solutions in detail.

We have had access to Windows NT source code during our development, which we have found useful but not essential. The learning curve for NT internals was very steep and sample code was particularly useful. (Our implementation can serve this purpose for others attempting Windows NT protocol development.) Not surprisingly, on several occasions source code was useful for debugging or for understanding poorly documented interfaces. In Section 5 we document more precisely when and how we made use of Windows NT source code.

The remainder of the paper is organized as follows. First, we present an overview of Windows NT's architecture for network protocols. Section 3 briefly describes our implementation and some of the major design choices we made. Next we report our experiences developing networking code for Windows NT, discussing solutions to problems inherent in Windows

NT's protocol architecture and also discussing the utility of access to Windows NT source code. Section 6 presents initial performance results for our implementation, and the next section compares our implementation to other IPv6 implementations. The paper ends with conclusions and a discussion of future work.

## 2. Windows NT Networking Architecture

Network protocols in Windows NT are dynamically loadable device drivers, much like any other device driver in Windows NT [1]. It is possible to add a new protocol to the system by writing two new components: a kernel-level driver (tcpip6.sys in Figure 1) that exports the TDI interface and uses the NDIS interface, and a user-level helper (wshipv6.dll in Figure 1) to support access to the driver via sockets.

Unlike the original BSD Unix sockets architecture, in which socket operations were direct system calls into the kernel, the Winsock architecture has several significant user-level components or Dynamic Link Libraries (DLLs). The Winsock DLL (ws2_32.dll) acts as a "traffic cop;" it redirects calls from the application to the appropriate Windows Socket Provider (WSP) or Name Space Provider (NSP). The WSP and NSP interfaces are both publicly documented. A WSP implementation can provide a new address family or an alternative implementation for an existing address family. WSP and NSP implementations make their presence known via entries in the system registry. The Microsoft WSP (msafd.dll) communicates with kernel drivers as described below, but other WSP implementations might function entirely at user-level or collaborate with a kernel driver via completely custom means. An NSP implementation supports name spaces for gethostbyname and related calls. For example, the Microsoft NSP (rnr20.dll) implements a DNS resolver.

To make it easier to add new protocols, msafd.dll supports multiple protocols through the use of helper DLLs. The helper DLL (like wshipv6.dll in our IPv6 implementation) exports the documented WSH interface, which msafd.dll uses when it wants to perform protocol-specific actions (like converting a socket address to a TDI address, or parsing the string representation of an address). msafd.dll handles almost all of the real work of being a Windows Socket Provider.

msafd.dll communicates with kernel protocol drivers via afd.sys. The interface between msafd.dll and afd.sys is not documented. Together msafd.dll and afd.sys handle buffering, select, and other minor issues as "glue" between Winsock and the TDI interface.

TDI (Transport-Device Independent) is Microsoft's kernel-level network protocol interface. It uses Microsoft's driver architecture [1], which encodes I/O operations in small structures called I/O Request Packets (IRPs). The driver architecture is inherently asynchronous. The kernel converts system calls into IRPs and drivers pass IRPs around until the operation that they represent completes. TDI is mostly just a family of operations encoded as complex ioctl IRPs. It uses its own address representation, distinct from Winsock's sockaddrs. It uses two kinds of pseudo-file objects, to represent connection-oriented and connectionless communication endpoints. It is documented, but not particularly well.

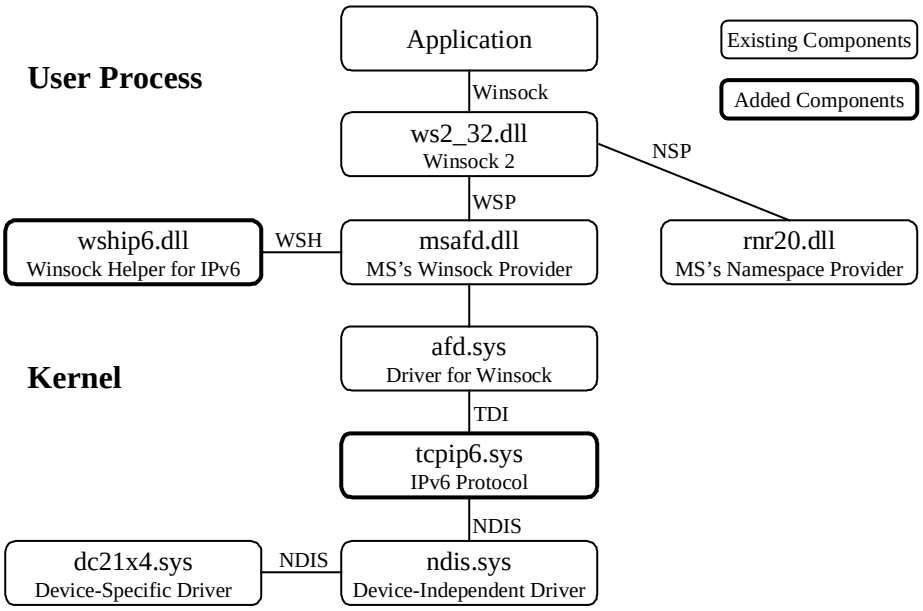


**Figure 1: Windows NT Networking Architecture**

A network protocol driver like tcpip6.sys implements TDI. Typically the protocol driver uses NDIS (Network Driver Interface Specification) to handle actual network interface cards. Like TDI, the NDIS interface is also asynchronous, but it uses callbacks instead of IRPs. For example the protocol calls NDIS to send a packet, and NDIS calls the protocol to indicate that a packet was sent and to indicate that a packet was received. Rather than have each hardware vendor reimplement NDIS, Microsoft supplies a common component (ndis.sys) that implements most NDIS functionality, and hardware vendors supply a relatively small "miniport" (like dc21x4.sys) that implements functionality specific to their device.

If one supplies an initialization file with configuration information, the standard Network Control Panel graphical user interface (GUI) can install, configure, and uninstall custom network protocols. The initialization file format is documented in the Device Driver Kit (DDK). Entries in the system registry control all configuration information, for network drivers, Winsock providers, and Winsock helpers. Registry entries also control the bindings between network interfaces and protocol stacks. For example, using Network Control Panel to modify the registry it is possible to disable a particular protocol for a given interface. One complication with configuration is that in Windows NT 5.0, both the initialization file format and the GUI for network installation and configuration have changed radically.

## 3. Our Implementation

Our implementation supports basic IPv6 functionality, but it is not a complete implementation. As of this writing, it supports Neighbor Discovery (which among other things replaces ARP), stateless address autoconfiguration (which allows hosts to configure automatically based on packets sent by routers), ICMPv6, Multicast Listener Discovery (essentially IGMPv2 for IPv6), several flavors of tunneling IPv6 via IPv4, and of course UDP and TCP over IPv6. It does not yet support security, authentication, mobility, or forwarding.

Future MSR IPv6 releases will include a separate driver for translating between IPv6 and IPv4, based on research at the University of Washington [4]. A translator allows an IPv6-only node to communicate with an IPv4-only node.

### 3.1. Implementation Strategy

We started with Windows NT 4.0 source for Microsoft's TCP/IP driver (tcpip.sys) and incrementally modified it. Eventually we replaced or rewrote all the IP-layer code, but our TCP and UDP layers are still strongly based on the original Microsoft code. Almost all of the machinery for supporting TDI is unchanged from the original code base. Starting with working IPv4 code and modifying it was very helpful in overcoming our learning curve and getting something working, but it resulted in intellectual property issues that we had to resolve when we wanted to plan a public source code release. Section 5 discusses our use of source code in more detail.

We also briefly considered starting with a public BSD-based IPv6 implementation and porting it to Windows NT. We feel that porting a BSD-based protocol, perhaps with TDI and NDIS glue layers, would be considerable work (the differences between BSD and Windows NT internals being much greater than the differences between IPv4 and IPv6) and probably result in an unsightly implementation. Because we would like our implementation to serve as a relatively clean example for others, we did not pursue this approach.

We decided to implement a "single stack," which only supports IPv6, as opposed to a "hybrid stack," which would support both IPv4 and IPv6 in an integrated fashion. We felt the single stack approach would be better for testing and experimentation, because the normal functions of the system, which rely on the IPv4 stack, are not affected by bugs or problems with the IPv6 stack. In practice this has worked very well and the presence of the experimental IPv6 stack has not caused problems for our systems. However for real product usage, the hybrid approach is probably superior for most scenarios. It eliminates the duplication of having separate IPv4 and IPv6 implementations of TCP and UDP. It also makes it easier to implement some transition mechanisms, like v4-mapped addresses (a way to have what seems to be an IPv6 socket which is really sending/receiving IPv4 packets) or an ioctl to change a IPv4 socket to IPv6 and back.

We decided very early on not to support Windows 95. The Windows 95 network architecture is similar to but not identical to Windows NT's, and we felt that for our purposes it was not worth the effort of understanding these differences, coding around them, and testing for two environments. The IPv4 code base from which we started was built with a proprietary glue layer to support both Windows NT and Windows 95, but we had to remove references to this glue layer prior to our public release. For a product implementation, Windows 95 support would be more interesting. The Windows 98 and Windows CE driver models more closely resemble Windows NT's, but we have not yet thoroughly explored the possibility of ports to those operating systems.

### 3.2. Code Organization

Our IPv6 implementation has three layers: the link layer, the network or IPv6 layer, and transport and other

upper-layer protocols. (See Figure 2.) The IPv4 code base from which we started had the same divisions into three layers, but we simplified the interfaces between the layers.

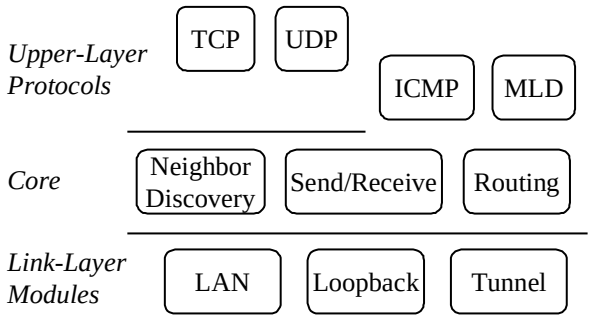


**Figure 2: IPv6 Code Modules**

The link-layer modules each manage a specific low-level interface type. The LAN module uses NDIS to handle Ethernet interfaces, and with minor changes it could also handle token-ring and other LAN media. The loopback module creates a pseudo-interface that reflects packets back to the sending machine. The tunnel module implements configured tunnels and automatic tunneling [5], and "6-over-4" [2]. In all three variations, the tunnel module uses IPv4 as the link layer to send and receive IPv6 packets. The tunnel module communicates with the IPv4 stack using TDI, in much the same way that the LAN module uses NDIS to communicate with Ethernet interfaces.

The core IPv6 modules communicate with link-layer modules through a well-defined interface. The link-layer module supplies IPv6 with a structure of information containing the length of link addresses, the network interface's own link-layer address, the link maximum transmission unit (MTU), and six entry points. The two substantial entry points send a packet and control multicast address assignment. The other four entry points are small helper functions, for creating an IPv6 address for the network interface given an IPv6 address prefix, for creating a link-layer multicast address given an IPv6 multicast address, and for reading and writing the link-layer address option fields in Neighbor Discovery packets. In the other direction, the link-layer modules call up to IPv6 to deliver received packets and to indicate the completion of packet-send operations. Our link-layer interface could easily be exposed to allow link-layer modules to reside in other kernel drivers or components, but we have not yet taken that step.

The IPv4 code base from which we started had a similarly well-defined link-layer interface, but the details were much more complicated. For example, IPv4 link-layer modules exported ten non-trivial entry points. Furthermore, IPv4 link-layer modules were responsible for allocating buffer space for their link-layer header (see Section 4.3) and for address resolution (ARP). In our implementation, common IPv6 code handles these responsibilities.

The core IPv6 modules implement the basic send/receive functionality, including fragmentation and reassembly and extension header processing, Neighbor Discovery (which replaces ARP for address resolution), and routing. Our current routing module only performs on-link determination and default router selection, using the data structures described below. We do not yet have any true routing table data structure or packet forwarding support.

The interface between higher-layer protocols like TCP or UDP and the core IPv6 code is fairly narrow but not yet as clearly defined as the link-layer interface. The IPv6 code calls up via a protocol switch table to deliver packets and control messages, and the higher-layer protocol code calls down to allocate packets, select source addresses, perform routing, and send packets. ICMP and MLD are technically upper-layer protocols but their implementation is integrated with the core IPv6 modules and data structures. For example, Neighbor Discovery uses ICMP messages and MLD uses the ADE data structures describe below.

## 3.3. Data Structures

Our design hews fairly closely to the conceptual data structures found in the Neighbor Discovery specification [11]. For example, it has a Neighbor Cache, a Destination Cache (although we call it a Route Cache), a Router List, and a Prefix List. The design deviates from the conceptual data structures in two major ways. First, we support multi-homed hosts (hosts with multiple interfaces), and this complicates the data structures slightly. Second, Route Cache entries cache the preferred source address for destinations as well as caching the next-hop neighbor, path MTU, and other information. Figure 3 presents the major data structures and their interconnections.

The Interface (IF) is our central data structure. There is one Interface for each network interface card, plus additional Interfaces for logical or virtual links like loopback, configured/automatic tunneling, and 6-over-4. In addition to link-layer information and configuration information, each Interface has a list of NTEs, which represent the addresses assigned to the interface that can be used as source addresses; a list of ADEs, which represent the addresses for which the Interface can receive packets; and a cache of NCEs, which represent neighboring nodes on that link.

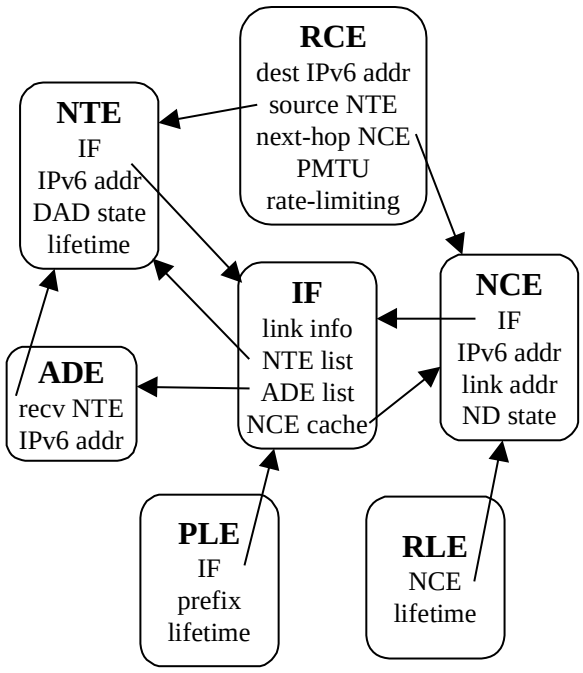


**Figure 3: IPv6 Data Structures**

The Net Table Entry (NTE) represents a unicast address assigned to an interface for use as a source address. It primarily contains state information for Duplicate Address Detection (DAD) and lifetime information derived from stateless address autoconfiguration. The name NTE derives from an IPv4 data structure similar in concept but different in detail (IPv4 does not have DAD or stateless autoconfiguration).

The Address Entry (ADE) represents a destination address for which the interface can receive packets. A unicast address can have both an ADE and an NTE, but a multicast address will have only an ADE. Each ADE maps to the NTE that logically receives the packets (would be used to reply to the packet if necessary). For example, IPv6 has the concept of addresses with link-local scope. An ADE for a multicast address with link-local scope maps to the NTE for the interface's link-local unicast address.

The Neighbor Cache Entry (NCE) represents a neighboring node on the interface's link. The NCE maps the neighbor's IPv6 address to its link-layer address. The Neighbor Discovery algorithms manage the state transitions for NCEs. We use a simple least-recently-used (LRU) algorithm to manage the cache.

The Prefix List Entry (PLE) and Router List Entry (RLE) represent routing information learned from advertisements sent by routers. Together they determine the next hop (NCE) to which a packet should be sent. If the destination address matches the prefix in a PLE, then the destination is assumed to be "on-link," or directly reachable. Otherwise the list of default routers (RLEs) must be consulted to choose a router for that destination.

The Route Cache Entry (RCE) caches the results of the next-hop selection algorithm. That is, an RCE maps a destination IPv6 address to a next-hop neighbor (NCE). In addition, the RCE caches the preferred source address (NTE) to use when sending to the destination, the path MTU, and ICMP error rate-limiting information. The name RCE derives from a similar IPv4 data structure.

We use a deliberately coarse-grained locking discipline for our data structures. Our intent is to refine the locking after we gain more experience with the data structures. Currently each Interface has a lock that protects the Interface itself and its NTEs, ADEs, and NCEs. There is a global route lock that protects PLEs, RLEs, and RCEs. In the normal case, sending a packet using a cached RCE does not require the acquisition of the route lock. The relevant Interface lock is briefly taken to send a packet (to check the Neighbor Discovery state) and to receive a packet (to search the ADE list).

The NCE, NTE, and RCE structures have reference counts, so pointers to them can be safely kept. Many of their interesting fields are read-only or can be safely accessed without holding a lock. The ADE, PLE, and RLE structures can only be accessed while holding the relevant Interface or route lock. The Interface structure does not have a reference count, but this is not a problem because our implementation does not yet support the plug-n-play features of Windows NT 5.0 that would allow interfaces to be removed at run-time. Once created, an Interface is never destroyed.

We also use deliberately simplistic data structures to represent our lists and caches. To keep the code simple, we use singly and doubly-linked lists instead of sorted arrays, trees, or hash tables. As we gain experience with the data structures in more demanding environments we plan to revisit these choices.

Using the conceptual data structures has been very successful for us. It has made it easy to track changes in the specification, because of the close correspondence between the spec and the code. We also hope that it will make our implementation more useful as an example for people interested in learning about IPv6. On the other hand, our implementation does not currently support routing tables, or forwarding of packets between interfaces. Efficient routing table support might require a more complicated data structure that would merge the PLE, RLE, and possibly the RCE data structures.

## 3.4. Configuration

Our implementation automatically configures itself as much as possible [14]. There is no configuration dialog in Network Control Panel. At boot time, it assigns link-local addresses to all interfaces, performs duplicate address detection, and solicits configuration information from routers. When it receives configuration information from a router, it automatically configures the receiving interface appropriately. This may involve creating new NTEs, PLEs, or RLEs. If the IPv4 stack is also present (normally the case), then it enables "automatic tunneling" [5] and creates a virtual "6-over-4" interface [2] for each IPv4 address. With automatic tunneling, packets sent to special v4-compatible IPv6 addresses are encapsulated and sent to an IPv4 destination address that is extracted from the low bits of the v4-compatible IPv6 address. With 6-over-4, the IPv6 code uses a multicast-capable IPv4 network as a true virtual link, with Neighbor Discovery and all other IPv6 features operational.

The only aspect of the implementation that must be configured manually, if the administrator wishes to enable it, is the "configured tunnels" transition mechanism [5]. For example, to connect an MSR IPv6 machine to the 6bone [12], a global IPv6 test network primarily composed of tunnels over the Internet, requires the following two registry entries. First, the administrator must supply an IPv6 address to be assigned to the tunnel interface. The machine accepts encapsulated IPv6 packets with this destination address. This results in the creation of an NTE and an ADE on the tunnel interface. Second, the administrator must supply the IPv4 address of the tunnel endpoint. IPv6 packets sent via the configured tunnel are encapsulated and sent to this IPv4 address. This results in the creation of an NCE (with a special state value that inhibits Neighbor Discovery) on the tunnel interface to represent the tunnel endpoint, and an RLE to represent the tunnel endpoint's role as a default router.

# 4. Problems and Solutions

During our implementation, we came across several challenges that would be faced by any protocol implementation for Windows NT. This section examines four such problems and our solutions in detail. In addition, we briefly document how our starting point IPv4 code addressed these problems. Note that we describe here the Windows NT 4.0 TCP/IP code base and Microsoft's TCP/IP stack has evolved very significantly in subsequent versions.

## 4.1. NDIS Receive Handlers

NDIS offers two different ways to receive packets. Unfortunately, the best method depends on the choice of network interface card and miniport. We were able to hide the differences between these methods inside the LAN link-layer module without sacrificing performance in any interesting cases.

A protocol implementation must support the most common method for receiving packets, called ProtocolReceive. With this method, NDIS calls the ProtocolReceive entry point with a pointer to a flat look-ahead buffer containing packet data. The buffer contents must be treated as read-only and are only valid for the duration of the call. Furthermore, the buffer may not contain the entire packet. For longer packets, the protocol may have to request NDIS to transfer the packet data to a new buffer (or chain of buffers) specified by the protocol. When this transfer is complete, NDIS calls another protocol entry point. This transfer-data case increases the interaction overhead with NDIS, but if used cleverly it can eliminate a copy by placing packet data directly in its final destination. If the protocol doesn't like what it sees in the look-ahead buffer and chooses not to transfer, this method may eliminate I/O operations.

NDIS 4.0 introduced an optional new method, called ProtocolReceivePacket, for receiving packets. (If a miniport wants to use ProtocolReceivePacket and the protocol does not support it, NDIS falls back to ProtocolReceive.) With this method, NDIS calls the ProtocolReceivePacket entry point with a pointer to a packet structure. The packet structure contains a chain of buffers, which the protocol must treat as read-only. The protocol can hold onto a received packet by returning a non-zero reference count to NDIS and later relinquish the references to return the packet. However it's not clear for how long a protocol may safely hold a packet. The miniport owns the packet's buffers, and preventing the miniport from reusing them may cause denial of service problems. Furthermore, in this situation the packet structure does not contain a "context" area that the protocol can use for its own purposes.

We examined the receive behavior of several different network interface cards and miniports. The Digital DE435 (dc21x4.sys) was the only one that used ProtocolReceivePacket. It always provided a packet with a single buffer. The SMC 9432TX EtherPower II (smc9432n.sys), the 3com 3c905 Fast Etherlink XL (el90x.sys), and the Intel EtherExpress PRO/10 (epro.sys) used ProtocolReceive, but the look-ahead buffer always contained the complete packet. The older Intel EtherExpress 16 (ee16.sys) was the only one that used ProtocolReceive with small look-ahead buffers, necessitating the use of transfer-data.

Given this data, we decided not to take advantage of ProtocolReceive's transfer-data case. We support transfer-data, so our implementation works with cards like

the EtherExpress 16, but inside the link-layer module we always transfer-data immediately to a temporary buffer instead of postponing the transfer until the data's final destination is known. This hides the transfer-data complexity from the core IPv6 modules. However, it introduces a copy, relative to the IPv4 stack, in some circumstances.

We support ProtocolReceivePacket, but we do not take full advantage of its capabilities. Given the time scales involved, it seems ill-advised to hold onto the miniport's buffers across the reassembly of IPv6 fragments or to buffer data for TCP. Hence we always return to NDIS a zero reference count for the packet. When we implement packet forwarding, we may want to hold a packet from one interface just long enough to resend it on another interface; this will complicate our design slightly.

To present a common picture to the core IPv6 modules, our link-layer module hides the apparent differences between ProtocolReceive and ProtocolReceivePacket. Our receive path uses our own IPv6 packet structure, defined with the fields we want, instead of the NDIS packet structure. For ProtocolReceive, we initialize a stack-allocated IPv6 packet structure to point to the look-ahead buffer. In the ProtocolReceivePacket case, we initialize an IPv6 packet structure to point to the chain of buffers from the NDIS packet. Receiving modules can deal in a unified way with the IPv6 packet structure, reducing the number of parameters passed up the stack at each layer from seven to two.

The base IPv4 code did not support ProtocolReceivePacket. It implemented the transfer-data case of ProtocolReceive, but it would not defer the transfer-data in most interesting cases (like receiving a TCP packet). It took advantage of transfer-data to eliminate a copy when discarding packets not actually destined for the receiver, when reassembling fragments, and when forwarding packets.

## 4.2. Pull-up

Our implementation "pulls-up" non-contiguous packet data into contiguous data with no overhead in the common case when pull-up is not required. When examining headers, it is very convenient to have contiguous data, and in most cases the header is in fact contiguous. However, the IPv6 packet structure mentioned above does permit a packet representation consisting of a chain of buffers. In this situation, a header may fall across two or more buffers. A BSD-style pull-up design does not work, because the buffer chain is read-only for the protocol stack and must be left undisturbed.

With the network interface cards we have tried to date, the NDIS ProtocolReceivePacket indication never provides more than one buffer for a packet, rendering pull-up moot. However, the NDIS interface explicitly allows for multiple buffers and some cards may take advantage of this capability. Furthermore, our other link-layer modules do in practice deliver multi-buffer packets.

We use the IPv6 packet structure mentioned in Section 4.1 to solve the pull-up problem. To support this, the packet structure contains several relevant fields. The Data field points into the current data area being examined. The ContigSize field tracks the amount of remaining contiguous data, and the TotalSize field the total amount of data remaining. An Auxiliary field remembers any side allocation of buffer space for the most recent pull-up, so it can be properly freed. Normally the Data field points into the packet's first buffer. When a pull-up is required, auxiliary buffer space is allocated and initialized from two or more buffers in the chain and the Data field is updated to point to the auxiliary buffer area. This solution does not modify the original buffer chain, which is read-only for the protocol stack.

Our implementation leverages the common case checks to hide the pull-up if it is never required. Before casting a buffer data pointer to a header structure, a receive handler must in any case perform a length check to verify that there is enough remaining data. Figure 4 shows an example code fragment.

```
if (Packet->ContigSize < sizeof(Header))
    if (! PacketPullup(Packet, sizeof(Header)))
        ; // pullup failed – packet too small
h = (Header *) Packet->Data;
```

**Figure 4: Pull-up Example**

The base IPv4 code did not implement a generic pull-up solution. The IPv4 code did not support extension headers between the IP header and the upper-layer header, making it possible to bound the total amount of header data. Furthermore, the IPv4 code did not support ProtocolPacketReceive.

## 4.3. Adding Link-Layer Headers

Our implementation avoids chaining new buffers to add link-layer headers, reducing packet-send latency and simplifying the link-layer modules. When sending a packet, it is most efficient to allocate space for all the headers up front, as one contiguous area. However, when constructing a packet one does not always know in advance the exact total size of the headers. For example, the outgoing interface may be chosen later, and in that case the size of the link-layer header that will be needed is not known.

The obvious solution, which we use, is to track the maximum link-layer header size needed by any interface. When constructing the packet's headers, it is easy

to leave space for the worst-case link-layer header at the front of the packet.

The problem with this solution is that the NDIS packet structure does not allow for unused buffer space at the front of a packet. This will occur if the actual NDIS interface has a smaller link-layer header than the pessimistically allocated maximum size. The same problem occurs when sending a packet via TDI in the tunnel link-layer module.

To solve this problem, we make a pair of helper routines available to link-layer modules. The first helper routine rewrites a packet's first buffer descriptor (which is actually a memory descriptor list, or MDL), to hide any unused buffer space. The second helper routine undoes the work of the first, so that the link-layer module can return the packet unchanged. The two helper routines communicate some state information (an offset value) via the protocol context area of the NDIS packet structure. The NDIS packet format allows for a context area, or protocol-defined annex following the main packet structure. The only tricky aspect to rewriting the MDL occurs if the unused buffer space crosses a physical page boundary, in which case an array of physical page addresses in the MDL must be rewritten in place.

## 4.4. Preventing Deadlock

During our implementation effort, we discovered that our IPv6 stack would hang some machines during boot. Debugging this problem (see Section 5.2) led us to an interesting deadlock between the core IPv6 modules and our link-layer modules. This section explains the deadlock, which is a potential danger for any protocol implementation, and our solution for avoiding it.

The potential for deadlock lies in NDIS's control operations. NDIS supports control operations such as changing the current list of link-layer multicast addresses to which the interface should listen. These control operations are asynchronous: one calls NDIS to request the operation, and NDIS calls a protocol entry point to indicate completion. This is inconvenient, and in fact our LAN link-layer module has a helper function (inherited from the IPv4 code base) that implements a synchronous control operation. The helper function initiates the control operation and then waits for the completion call back via a synchronization object.

The problem arose when our stateless address autoconfiguration code would configure a new address on an interface. This could change the list of multicast addresses for the interface, because for each IPv6 unicast address there is a corresponding address, the solicited-node multicast address, used in Neighbor Discovery. A change in the list of IPv6 multicast addresses in turn could change the link-layer multicast addresses. This would result in a "synchronous" call into NDIS via the helper function. In most cases this worked fine, but with some interface cards it would deadlock because NDIS would call the protocol stack's receive entry point while holding an internal lock. The multicast address control operation needed to acquire this same lock.

We tried two different solutions. The first solution was to expose in our internal link-layer interface the inherently asynchronous nature of the set-multicast-address-list control operation. This worked, but it greatly complicated our link-layer interface. The final solution was to document that the set-multicast-address-list operation in the link-layer interface can only be called from a safe thread context; it can not be called from the receive path. When the receive path wants to invoke this operation (for example because of stateless address autoconfiguration), the actual call is deferred to a kernel worker thread.

The final solution also solved a problem with the tunnel link-layer module. The tunnel link-layer module uses the TDI interface to the IPv4 stack instead of NDIS. The IPv4 stack exposes a TDI operation for controlling multicast addresses. We never saw this problem in practice, but we realized that the IPv4 implementation of this TDI operation implicitly assumed that it was being called from a preemptible thread context. (For example, it executes pageable code.)

The lesson we draw from this is to carefully document the call-context assumptions in interfaces. It should be clear whether a function must be called from a preemptible thread context, what locks may be held when a function is called, etc. The NDIS and TDI interfaces are both inadequately documented from this perspective.

# 5. Source Code Access

In this section, we explore how we made use of Windows NT source code. We have access to all Windows NT source, and in fact for our MSR IPv6 implementation we started from the source code for TCP/IP in Windows NT 4.0. The TCP/IP stack was very valuable as sample code, but in the end we have replaced almost all of the core IP-layer code. We have made limited use of the source for other Windows NT components.

## 5.1. Source for Windows NT 4.0 TCP/IP

We based our implementation on an old version (NT 4.0 with no Service Packs) of Microsoft's IPv4 protocol stack. This source code played several roles for us:

- Sample code. Sample source code was essential for us to get started quickly. We had no prior experience with network protocol programming for Windows NT.

- UDP/TCP code. Not having to implement UDP, and especially, TCP has saved us much effort.
- TDI glue code. Our stack's support for the TDI interface derives almost unchanged from the IPv4 code. This includes the code for managing TDI's pseudo-file objects that represent communication endpoints.

At this point, we have replaced or written from scratch most lines of code in the link-layer modules, the core IPv6 modules, and ICMP and MLD. There is still a noticeable genetic resemblance to the original IPv4 code, in terms of code organization and function and variable names. In the UDP and TCP modules we have made relatively minor changes. We changed the buffer handling in TCP to support our IPv6 packet structure in the receive path. We updated the UDP and TCP checksum calculations. We updated variables, fields, and parameters that represented addresses. Addresses as parameters generally changed from by-value to by-reference.

For sample code, we did not look early enough at Microsoft's Driver Development Kit (DDK). The DDK has a sample network driver, which looks useful but we have not evaluated it carefully. The DDK also contains a sample Winsock helper DLL. This sample is really just the Windows NT 4.0 IPv4 helper, with minor modifications for the DDK build environment. Our IPv6 Winsock helper DLL was originally based on Windows NT source, but we easily recreated it based on the DDK sample. More importantly, the Winsock helper sample implicitly reveals some aspects of the TDI interface to the IPv4 stack that are nowhere else documented. For example, our tunnel code uses the IPv4 stack via TDI and the ioctls (specific to the IPv4 stack) for controlling multicast are only documented in this DDK sample.

## 5.2. Other Windows NT Components

We have made limited use of the source code for other Windows NT components. It has occasionally been useful for debugging or for informational purposes, when interfaces were inadequately documented. We have not modified any existing Windows NT components for our implementation, with the exception of one bug fix in msafd.dll.

We did not have good sample code for Winsock's WSALookupServiceBegin/Next/End APIs, and source code for rnr20.dll (the DNS Name Space Provider) was essential for our use of these ostensibly documented APIs. These APIs are a generalized version of gethostbyname. In fact it turns out that with the right combination of arguments, one can use these APIs to request AAAA (IPv6 address) records from DNS, and have the raw DNS replies returned. We have wrapped this in a helper function that looks up IPv6 addresses, despite the fact that rnr20.dll does not support IPv6. Unfortunately in this usage, rnr20.dll does not cache the DNS replies.

Source code for other Windows NT components (like ntoskrnl.exe, afd.sys, ndis.sys) was occasionally very useful for debugging but not essential to the project. During debugging it would be helpful to step through these other components as well as the IPv6 driver to understand a problem. The deadlock problem described in Section 4.4 was a prime example of this.

We came across a problem in getsockname that we traced to a bug in msafd.dll that only showed up with large addresses. Of course, source code for the Winsock components was essential for finding and fixing this.

## 5.3. Network Monitor

Microsoft's Network Monitor tool captures network packets and parses them for display. It has been extremely useful during our development. It supports run-time loadable parsers, developed with Microsoft's SDK. The parser architecture is fairly simple and clean and it was very easy to write a new IPv6 parser. Although in general new parsers can be developed without access to Network Monitor source, we did have to modify the existing IPv4, UDP, and TCP parsers to deal properly with the interactions between the IP layer and the UDP/TCP layers. One example is verifying the TCP checksum in the TCP parser, when the previous header could be IPv4 or IPv6.

# 6. Performance

We examined the TCP performance of our IPv6 implementation, and found roughly 2% performance degradation relative to IPv4 for both 10 and 100 Mb/s Ethernet. Because IPv6 packets have a larger header, we expected roughly 1.4% performance degradation. We have not yet tuned our stack for performance, so we are satisfied with these initial results. Fiuczynski [4] has performance numbers for an earlier version of our implementation.

Our testing environment consisted of two machines directly connected via a reversing Ethernet cable. The sending machine was a 300 MHz Pentium II Gateway E5000; the receiving machine was a 266 MHz Pentium II Gateway E3100. Both were equipped with SMC 9432TX EtherPower II network interface cards, which are capable of running in either 10 or 100 Mb/s mode. We performed tests in both modes. While the cards also support full-duplex transfers, we performed all our testing at the more commonly used half-duplex setting.

Both machines were running Windows NT Version 4.0 with Service Pack 3 installed. However, to achieve a fair comparison we replaced the TCP/IPv4 driver (tcpip.sys) with a driver that we built using the Windows NT 4.0 sources from which we derived our IPv6 implementation. The software driving the test was the public domain "ttcp" program, which we ported to run on our IPv6 stack. We used 128KB socket buffers. (We observed very similar performance with 64KB buffers, but buffers smaller than 64KB performed poorer.)

We ran TCP throughput tests for both IPv4 and IPv6 over both 10 Mb/s and 100 Mb/s Ethernet. Each test was run six times, with each run taking about 100 seconds. We present the mean throughput and standard deviation for each test.

We expected to see a performance degradation of approximately 1.4% with the IPv6 stack. IPv6 headers are 40 bytes, IPv4 headers are 20 bytes, and TCP headers are 20 bytes. The maximum Ethernet frame size is 1500 bytes. So the degradation is 20 bytes out of 1460 bytes.

| | 10 Mb/s | 100 Mb/s |
|---|---|---|
| IPv4 | 1058±4 | 10995±20 |
| IPv6 | 1032±3 | 10790±30 |

**Table 1: TCP Throughput in KB/s**

Table 1 depicts our actual results. For 10 Mb/s Ethernet, we see 2.5% degradation. For 100 Mb/s Ethernet, we see 1.9% degradation.

While performing these measurements, we noticed several things that we have not been able to follow up on. The IPv4 stack that we built yields slightly better throughput than the IPv4 stack released with Windows NT 4.0 Service Pack 3. With the 266 MHz machine sending and the 300 MHz machine receiving, we observed better 10 Mb/s performance and poorer 100 Mb/s performance. We also tried Digital DE500 interface cards, but we observed substantially poorer 100 Mb/s performance.

## 7. Other IPv6 Implementations

This section describes our IPv6 stack in comparison with other current IPv6 implementations.

The IPng Home Page [9] implementations list includes many implementations for host computers, notably actively supported "early adopter" implementations from Digital [6] and Sun [13]. These implementations are not available as source code.

Both Digital and Sun developed hybrid stack implementations, in which an integrated implementation of IPv4 and IPv6 share common transport-layer code. An IPv6 socket (AF_INET6) is usable for both IPv6 and IPv4 traffic, so that the result of a domain name lookup at runtime can determine what IP should be used on output, and the arrival of an IPv6 or IPv4 datagram determines the stack used on input.

We can compare the Digital and Sun implementations with MSR IPv6 and another freely distributed source code, that of INRIA Roquencourt [8]. We and INRIA have separate transport modules for IPv6. In our case, the goal was to avoid interfering with IPv4 traffic in any way during experiments using IPv6. MSR IPv6 does not modify the TCP/IPv4 driver. The drawback is excess code in the system and a requirement for two sockets in applications for them to run over both IP versions. This seems like the right tradeoff for a frequently changing research environment where both the stack and the applications may be modified at will.

On other dimensions, the four stacks here are very comparable: all have complete implementations of the base specifications and Neighbor Discovery. None have as yet fully implemented the required IPSec and mobility functions. As of this writing, compared to the other implementations MSR IPv6 has more complete multicast support.

### 7.1. Size Comparison with INRIA IPv6

This section compares the MSR IPv6 source code with the freely available INRIA implementation's source code [8].

INRIA used the strategy of creating clones of the IPv4 files for IPv6. INRIA IPv6 files are very similar to their IPv4 counterparts. In contrast, though MSR IPv6 started from IPv4 code, the core IPv6 modules have been largely designed and written from scratch.

Table 2 shows code size comparisons of INRIA IPv6 and IPv4 and of MSR IPv6 and its IPv4 starting point code. The table uses raw source lines because it compares INRIA to INRIA and MSR to MSR, and a partial set of comparisons with comments stripped looked very similar.

| Module | INRIAv4 | INRIAv6 | MSRv4 | MSRv6 |
|---|---|---|---|---|
| IP input | 1427 | 2205 | 1145 | 1109[1] |
| IP output | 1432 | 2503 | 1867 | 451[1] |
| TCP | 4239 | 4678 | 12089 | 11275 |
| ICMP[2] | 1653 | 1654 | 2286 | 2230 |

**Table 2: Lines of Code**

*[1]MSR IPv6's subr.c (869 lines) is not included in the IP input and output line counts. [2]Including IGMP or MLD, but not Neighbor Discovery.*

As expected, the cloning method results in larger code sizes for adding IPv6 to an IPv4 base. The method of rewriting and optimizing that we did often results in a

code for an IPv6 function, such as packet output, that is smaller than the parallel IPv4 function. To a degree that is difficult to quantify, the streamlined design of the IPv6 base protocol may contribute to the shrinkage. For example, IPv6 has simpler header structures.

The INRIA ICMP module has similar line counts for IPv4 and IPv6 because the revisions in the design of ICMP from IPv4 to IPv6 have led INRIA to rewrite this code instead of cloning it.

The minimal effect of IPv6 on TCP is clear from the comparison of TCPv4 and TCPv6 code sizes. The TCP pseudo-header checksum calculation changes for IPv6, but otherwise the protocol is not modified. The shrinkage seen for the MSR TCP results from simplifications to buffer management stemming from our IPv6 packet structure. The MSR TCP implementation does not yet take advantage of an opportunity to optimize Neighbor Discovery by using reachability information gleaned from TCP acknowledgments to suppress Neighbor Discovery messages. This would add slightly to the TCP code size for IPv6.

## 8. Conclusions

Once past the learning curve, we have found Windows NT to be a good platform for protocol development. The internal interfaces are often complex, but so far we have been able to accomplish everything we need to do with them.

We plan to continue our development, with periodic public source code releases. Our first public source release was March 24, 1998, available at http://www.research.microsoft.com/msripv6. Our first priority in subsequent releases is finishing a full host implementation, including security, authentication, and mobility support, and adding interesting applications to the release.

Beyond our own work of completing this IPv6 implementation, we believe also that the code is a good resource for hands-on research in a variety of areas. Some examples include active network protocols used beside or instead of IP, new unicast and multicast transport protocols or algorithms, signaling protocols, queue management algorithms, and network aware applications.

## Acknowledgments

Maryann Pérez Maher and Paul Dyke contributed significantly to our implementation. Marc Fiuczynski wrote our IPv6-IPv4 translator, based on his research at the University of Washington. The Windows Networking Group generously allowed us to release portions of their code in our public distribution.

## References

[1] Art Baker. The Windows NT Device Driver Book: A Guide for Programmers. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1997.

[2] B. Carpenter and C. Jung. Transmission of IPv6 Packets over IPv4 Networks without Tunnels. Internet Draft, draft-carpenter-ipng-6over4-03.txt, September 1997.

[3] S. Deering, R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 1883, December 1995.

[4] Marc E. Fiuczynski, Vincent K. Lam, and Brian N. Bershad. The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator. Proceedings of the 1998 USENIX Technical Conference, June 1998.

[5] R. Gilligan, E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 1933, April 1996.

[6] Daniel T. Harrington, James P. Bound, John J. McCann, Matt Thomas. Internet Protocol Version 6 and the Digital UNIX Implementation Experience. Digital Technical Journal, Volume 8, Number 3, http://www.digital.com/DTJN01/DTJN01HM.HTM, 1996.

[7] R. Hinden, S. Deering. IP Version 6 Addressing Architecture. RFC 1884, December 1995.

[8] INRIA Rocquencourt IPv6. ftp://ftp.inria.fr/network/ipv6/.

[9] IPng Working Group Web Site. http://playground.sun.com/pub/ipng/html/ipng-main.html.

[10] Microsoft Research IPv6 Implementation. http://www.research.microsoft.com/msripv6.

[11] T. Narten, E. Nordmark, W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC 1970, August 1996.

[12] 6bone Web Site. http://www.6bone.net/.

[13] IPv6 for Solaris. http://playground.sun.com/pub/solaris2-ipv6/html/solaris2-ipv6.html.

[14] S. Thompson, T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 1971, August 1996.

Restrictions for Implementing Tunneling for IPv6. â€¢ The IPv6 rapid deployment (6RD) feature is supported in an ethernet-only topology. â€¢ IPv6 VRF is not supported with the 6RD feature. â€¢ The Cisco ASR 1000 Series Aggregation Services Routers support as many as 2000 6RD tunnel. interfaces. Americas Headquarters: Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706 USA. Information About Implementing Tunneling for IPv6.Â  Information About Implementing Tunneling for IPv6. Table 1. Suggested Usage of Tunnel Types to Carry IPv6 Packets over an IPv4 Network. Tunneling Type Manual. Suggested Usage. Usage Notes. Simple point-to-point tunnels that Can carry IPv6 packets only. can be used within a site or between sites. Many enterprises are beginning to implement IPv6, often starting with enabling IPv6 on their email and web servers. This, at least, makes it possible to communicate with the outside world via both protocols. Some are also enabling IPv6 in their internal networks, including corporate WANs and data centres.Â  But what about the client networks consisting of mostly Windows PCs? In this post, Iâ€™ll outline how enterprises can deploy IPv6 and continue to run IPv4 on client networks. Todayâ€™s enterprise networks.Â  Enabling IPv6 in these networks would make it even worse, because adding a protocol to an insecure environment makes it more insecure. Therefore, this is not an option. Further, DHCPv6 works differently from DHCPv4; this makes a one-to-one replacement impossible. To be precise, IPv6 traffic gets rerouted to the VPN server and never leaves it, only IPv4 traffic does in order to ensure that your real IP address is not leaked.Â  In order to disable IPv6 on your router, please check router user manual or consult with an IT specialist. You can turn off IPv6 traffic directly on Windows, macOS and Linux. Post Comment. 18 comments.