Internet Protocol Version 6 and the Digital UNIX Implementation Experience

Daniel T. Harrington
James P. Bound
John J. McCann
Matt Thomas

In the early 1990s, the Internet community recognized that the current TCP/IP architecture was not capable of sustaining the explosive growth of the Internet. In July 1994, the Internet Protocol next generation (IPng) directorate responded to the problem with the Internet Protocol version 6 (IPv6) as the replacement network layer protocol. Working groups of the Internet Engineering Task Force (IETF) then began to build specifications that would address the needs for an expanded Internet address space, an increase in router table size, and new technology features. As a contributor to these efforts, Digital has implemented IPv6 on the Digital UNIX platform. The primary goal of Digital's efforts has been to evaluate the technical feasibility of the proposed architecture and provide critical feedback to the standards development process in the IETF. The secondary goal has been to evaluate system design alternatives to gain the experience needed to allow Digital to incorporate this new architecture into existing products.

As one of its ongoing advanced development efforts in networking technology, Digital has built an Internet Protocol version 6 (IPv6) prototype for the Digital UNIX operating system. In this paper, we describe the design of the Digital UNIX IPv6 prototype and its history relevant to the Internet Protocol next generation (IPng) effort in the Internet Engineering Task Force (IETF). We also compare its relationship with the existing Transmission Control Protocol/Internet Protocol (TCP/IP) suite. We emphasize techniques and technologies that were developed to accommodate particular aspects of the IPv6 architecture and issues that required further discussion in the IETF. In particular, we discuss the modifications to the transport layer modules to use two distinct network layer protocols, along with the implications to the UNIX socket layer and applications. In addition, we describe the new IPv6 and Internet Control Message Protocol (ICMP) network layer modules, including their interactions with both the data link layer and the IPv4 protocol. We review the new Neighbor Discovery Protocol and its algorithms and give details of its implementation.

To accommodate the dynamic nature of future networks, IPv6 includes mechanisms to do both stateless and stateful address configuration, as well as router discovery; we explain the design of a user-mode process that implements these functions. The paper includes a discussion of enhancements to well-known IPv4 services, such as dynamic updates to the domain naming service (DNS), as well as general techniques to support the transition of existing applications. The paper concludes with an overview of what we have learned in this project and summarizes our current status and future work, including efforts in nonbroadcast multiple access (NBMA) data link technologies such as asynchonous transfer mode (ATM) and resource reservation protocols.

Internet Protocol Next Generation

In the early 1990s, the members of the Internet community realized that the address space and certain aspects of the current TCP/IP architecture were not capable of sustaining the explosive growth of the

Internet. Within the IETF, several efforts were undertaken to both study and improve the use of the 32-bit Internet Protocol (IPv4) addresses, as well as to identify and replace protocols and services that would limit growth. The 32-bit addressing architecture in the network layer was quickly determined to be the crux of the problem, with both hardware and human limits approaching fundamental boundaries.[1] IPv4 addresses are unevenly allocated in blocks that are often too large or too small; they are also difficult to change within any existing network.

When the IETF called for replacement proposals, Digital participated in this industry-wide effort by submitting white papers outlining issues and by developing and evaluating prototypes of the various proposals. Digital also participated in the IETF working groups and in the IPng directorate, which had the responsibility for making the ultimate decision. In July 1994, the IPng directorate selected the Internet Protocol version 6 (IPv6) as the replacement network layer protocol, and IETF working groups began to build specifications. "The Recommendation for the IP Next Generation Protocol" summarizes the candidates and explains the selection of this protocol.[2]

Digital UNIX Prototype

The current Digital UNIX IPv6 prototype project is Digital's most recent addition to an ongoing effort to develop and evaluate the competing IPng proposals. This began with the Simple Internet Protocol (SIP), which used eight octet addresses. SIP was later melded with another early proposal and became known as Simple Internet Protocol Plus (SIPP), the direct antecedent of IPv6.[3] The primary goal of Digital's efforts has been to evaluate the technical feasibility of the proposed architecture and provide feedback to the IETF working groups. This is critical to the standards development process in the IETF, which requires multiple independent and interoperable implementations of a specification before it may become an Internet standard. An additional goal has been to evaluate system design alternatives to gain the experience needed to allow Digital to incorporate this new architecture into existing products. Digital has made the prototype available to researchers within the company as a source code distribution and more recently has begun to supply binary kits for early adopters and evaluators in the Internet community. As the IPv6 protocol and architecture matures, we have begun to focus on how to best integrate the code into the Digital UNIX product.

IPv6 Overview

To understand the system-wide impact of IPv6, we review some of its new features and contrast them with the IPv4 model. IPv6 is both a completely new network layer protocol and a major revision of the Internet architecture. At both levels, it builds upon and incorporates experiences gained with IPv4.

Figure 1 shows the evolution of the packet format into the new IPv6 header. It retains some fields (version, source, and destination address), clarifies the role of others (for example, the Time To Live [TTL] field is renamed the Hop Limit), and introduces new ones (such as Flow ID) with as yet untapped potential. The next header field allows for modular construction of complex packets: different header types can be chained together to provide specialized functionality, including security and source routing. Finally, all headers are structured to allow 64-bit alignment, which should allow optimal processing both at source and destination systems, as well as in transit.[4]

The most striking departure from IPv4 is the address size: it has increased from 32 bits to 128 bits. The IPv6 addressing architecture is rich, with prefixes for multicast addresses and predefined scopes for both unicast and multicast addresses. One special type of unicast address is the link-local address, which permits communications with only those systems directly connected on the same link. This allows a standard bootstrapping mechanism, so that systems can learn about neighbors and services before a routable address is assigned to an interface. Various address assignment options have been defined, including hierarchical models based upon regional registries and service provider identifiers.[5,6] In each case, care has been taken to ensure proper route aggregation, which will help yield more efficient backbone router performance.

Multiple means of acquiring addresses have been defined for IPv6 addressing, with the goals of allowing flexibility through different administrative policies

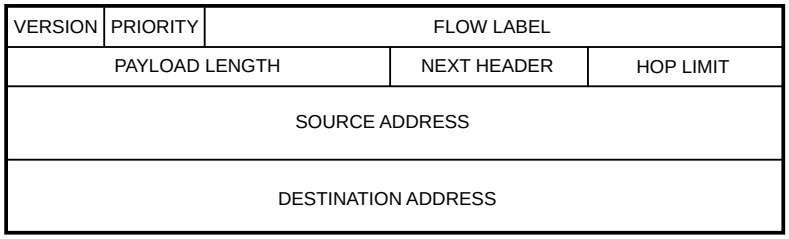

Figure 1
IPv6 Header

and, perhaps more important, of demanding that network address reassignment be supported throughout the architecture. The two new addressing services are Stateless Address Autoconfiguration and the stateful, transaction-based Dynamic Host Configuration Protocol version 6 (DHCPv6).[7,8] In the stateless model, address prefixes are learned by listening for router advertisement packets. Addresses are formed by combining the prefix with a link-specific token such as the 48-bit Ethernet hardware address. In the stateful procedure, hosts may request addresses, configuration information, and services from dedicated configuration servers, with routers potentially serving as relay stations during the initial phase. In both cases, the resulting addresses have associated lifetimes, and systems must be prepared to both learn new addresses and release expired addresses. Combined with the ability to register updated address information with DNS servers, these mechanisms provide a path toward network renumbering, a goal that has proved difficult to achieve in the IPv4 world.

Finally, the Internet Control Message Protocol version 6 (ICMPv6) was developed.[9] This specification aimed to merge the functions of two distinct IPv4 protocols for reporting errors and status, ICMP for unicast packet transmission and the Internet Group Message Protocol (IGMP) for multicast traffic.

The messages defined in this protocol are categorized as either error or informational, with a family of messages in the second group used to provide the Neighbor Discovery Protocol.[10] Neighbor discovery serves multiple purposes with the overall theme of providing a system with topological and environmental hints. For example, link-layer address resolution, router discovery, destination address redirection, and address autoconfiguration mechanisms are all specified using neighbor discovery packet types.

Although the network layer did experience the largest amount of change, Figure 2 shows that the effects of this work touch nearly all aspects of the Digital UNIX system. We point out examples of decisions made due to our fundamental design philosophy, which is based upon integration with the UNIX system framework, modular and extensible software, support for multiple operational policies, and a desire to take advantage of the Alpha platform without compromising portability.

In the following sections, we study these topics in depth, beginning with the network layer, then covering the transport layer modifications and the new neighbor discovery algorithms. After that, we discuss address autoconfiguration mechanisms and their effects upon the system. We conclude with services that will be affected by the transition from IPv4 to IPv6 such as the socket application programming interface (API) and DNS.

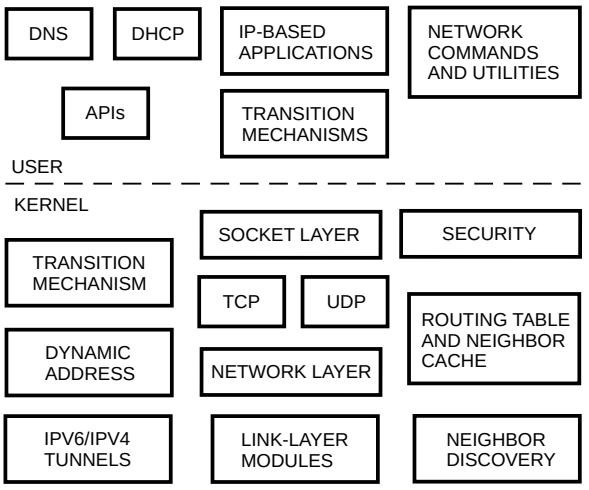

Figure 2
Base Platform Changes

Network Layer

In this section, we review the processing requirements of the IPv6 modules, including ICMPv6, extension header options, and fragmentation. An early design decision was made to base the networking subsystem on the Berkeley Standard Distribution (BSD) 4.4 model and code base, which allows great flexibility in dealing with multiple network layers.[11] This also has the advantage of providing support for variable-bit-length netmasks (also known as CIDR-style netmasks, from Classless Inter-Domain Routing), which are appropriate to both IPv4 and IPv6.[12] We have also tried to take maximum advantage of the 64-bit Alpha architecture when implementing IPv6, while making certain that this implementation would run on 32-bit CPUs as well. For example, the checksum routines operate on 32-bit quantities (allowing the carry to overflow into the upper 32 bits of a 64-bit register). The checksum routine is also designed to allow it to be issued to multiple Alpha execution units, which remains a topic for further investigation.

Adaptations to Existing IP and ICMP Routines

The IPv6 and ICMPv6 routines are completely independent of the corresponding IPv4 and ICMPv4 routines, and the processing styles have distinct differences. In IPv6, the incoming packet is treated as being read-only, while the BSD IPv4 code manipulates fields within the IPv4 header. We also avoid unnecessary use of the m_pullup routine (which consolidates chained memory buffers into a single large buffer) because this could cause the packet to be needlessly lost. Finally, instead of passing numerous arguments when calling from function to function, a common data structure is

used to store necessary data and pointers; for most function calls, it is only necessary to pass a pointer to this structure. This reduces the stack overhead and also yields modular and easily extensible subroutines.

IPv6 has a dedicated interrupt processing thread, and received IPv6 packets are placed onto their own interface input queue (ifqueue). When an IPv6 packet is taken off the ifqueue, basic validity tests are done; only after passing them is the packet tested to see if it is directed to a unicast or a multicast address.

If the packet is to a multicast address, the destination is compared to the enabled IPv6 multicasts for the interface over which the packet was received. If the destination matches, the packet is passed up to normal packet processing; otherwise, a copy of the packet is passed to the multicast forwarder.

Similarly, unicast packets are checked to see that the destination matches one of the system's addresses. In the special case of the packet being targeted to a link-local address, only the link-local address for the receiving interface is compared. If there is an exact match, the packet is processed normally; otherwise, it is passed to the unicast packet forwarding routine.

Header Processing

After a packet has been matched to a local address, the IPv6 headers must be processed, independently of whether the packet is multicast or unicast. This processing is done in a common routine that handles all types of IPv6 headers. A number of actions may result from the verification and analysis phase, including an ICMPv6 packet being sent back to the source, the packet being silently dropped, or being forwarded to another node due to a source route. If none of these possibilities occurs, the next IPv6 header in the packet is processed.

If the header is a known IPv6 header type, the appropriate routine is called. If not, this packet is probably destined for another protocol module such as TCP, the User Datagram Protocol (UDP), or ICMPv6. The header type is looked up in the list of active protocols and passed to the matching protocol input routine. If no entry is found, an ICMPv6 error may be sent back.

Header Options

Since the hop-by-hop and destination node headers have the same format, a common routine processes both types. As the routine processes each option, it validates the option. If this fails, it checks whether an ICMPv6 parameter problem error should be sent, whether the packet should be discarded, or the option ignored.

ICMPv6 Processing and Checksums

Upon receipt of an ICMPv6 packet from a node in the network reporting an error or other information, it is first validated for correct packet format and checksum. The packet is then further processed based upon its ICMPv6 type value. If it has an ICMPv6 error type (i.e., type value less than 128), the appropriate notifications are sent to the affected protocol. Neighbor discovery packets, which are all informational, have a number of additional consistency checks, and the packet is dropped if it fails them. After the ICMPv6 packet has been processed, it is also sent to any ICMPv6 raw sockets that have requested reception of that type. The exception to this rule is an ICMPv6 echo request packet, which is not copied to the raw sockets.

When an ICMPv6 echo request is received and validated, the ICMPv6 echo response packet is prepared. In the typical case, it is identical to the echo request except for the ICMPv6 type and checksum value. The exception would be an echo request sent to a multicast address, in which case a source address must also be selected. Rather than computing the checksum on the new packet, the received checksum is simply adjusted down by 1, since the sole difference between the two packets is the value of the ICMPv6 type fields, and ICMPv6 echo request and echo response types differ by 1.

IPv6 requires all nodes to support multicasting, specifically level 2 as defined in "Host Extensions for IP Multicasting."[13] Although this was written for IPv4, the same general algorithms are used for IPv6. One notable exception to this is that the multicast addresses used for neighbor solicitions and the predefined link-local multicasts such as all-nodes and all-routers do not require periodic status reports.

Path Maximum Transmission Unit Discovery

One of the significant differences between IPv4 and IPv6 concerns fragmentation. In IPv6, fragmentation may be done only by the node from which a packet originates. Forwarders, which may be routers or hosts acting upon source routing headers, are not permitted to fragment packets. The burden is on the originating node to send packets that are small enough to fit through all the links along the paths to their destinations, where each link type may have a different maximum transmission unit (MTU). To ease this burden, IPv6 defines a minimum link MTU of 576 bytes. A node may use this as the upper limit on packet size and be assured that its packets are sufficiently small to reach their destinations.

The minimum MTU of all the links in a path between two nodes is referred to as the path MTU.[14] In many cases, the path MTU will exceed 576 bytes, and it is desirable to send the largest possible packets. IPv6 provides a mechanism by which a node may discover a path's MTU.[15] When a forwarder cannot forward a packet because the packet is too large for the next hop's link MTU, it sends an ICMPv6 Packet Too Big (PTB) message back to the source of the packet. The PTB

message contains the MTU of the constricting link. The source node adjusts its packet size to fit through this link.

Path MTU information is kept on a per-destination basis and is stored in the routing table entry for a given destination. Packets sent on that route will be sized according to the path MTU value. When a PTB message is received, the appropriate route is updated to contain the new path MTU value as reported in the PTB message, and a timer is started. When the timer expires, the path MTU value is increased to the (known) MTU of the first hop link. This allows the node to detect increases in the path MTU.

Switches are provided to disable path MTU discovery system-wide, on a per-destination basis and on a per-socket basis. When path MTU discovery is disabled, packets are limited to 576 bytes.

Fragmentation

A packet that is larger than the MTU of the path on which it is to be sent must be fragmented. Unlike IPv4, the IPv6 header contains no fields to carry fragmentation information. Instead, this information is carried in a specialized extension header, called the fragment header. As shown in Figure 3, the fields in the fragment header include an offset, in eight octet units, and an identifier common to all fragments of the original packet. The M (managed) flag is used to indicate intermediate fragments; the terminal fragment has the bit cleared. Note that the amount of data in a fragment packet is derived from the total packet length.

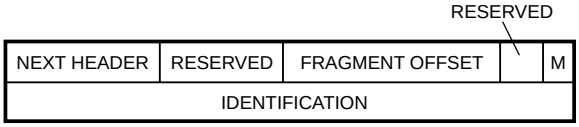

Figure 3
Fragment Header

The first step in the fragmentation process is to identify the fragmentable and unfragmentable parts of the original packet (see Figure 4). The unfragmentable part of the packet consists of the IPv6 header and any extension headers that must be processed by each node traversed by the packet (e.g., hop-by-hop header, routing header). The fragment header is appended to the unfragmentable part. The rest of the packet is divided into fragments, and each fragment is appended to a copy of the unfragmentable part plus fragment header.

When the fragment header is appended to the unfragmentable part, two fields in the unfragmentable part must be updated. First, the payload length field in the IPv6 header must be updated to reflect the length of the fragment packet. Second, the next header field in the last header of the unfragmentable part must be changed to indicate that a fragment header follows.

A copy of the unfragmentable part is created for each fragment packet. As an optimization, Digital UNIX allows portions of a packet to be shared among copies of the packet, to avoid an actual data copy. As with IPv4, care must be taken to ensure that fields being updated are not contained in shared buffers. This is typically accomplished by copying the portions that must be updated into a private memory buffer (mbuf). Unlike IPv4, the unfragmentable part may not fit in a single mbuf, and the IPv6 fragmentation code must be capable of handling this case.

To reduce the possibility of fragment loss at the source node, all the fragment packets are built before any is passed to the data link for transmission.

A question that arises here is how big should the fragment packets be? Should they be sized according to the path MTU, or should they be limited to 576 bytes? The former yields the desirable larger

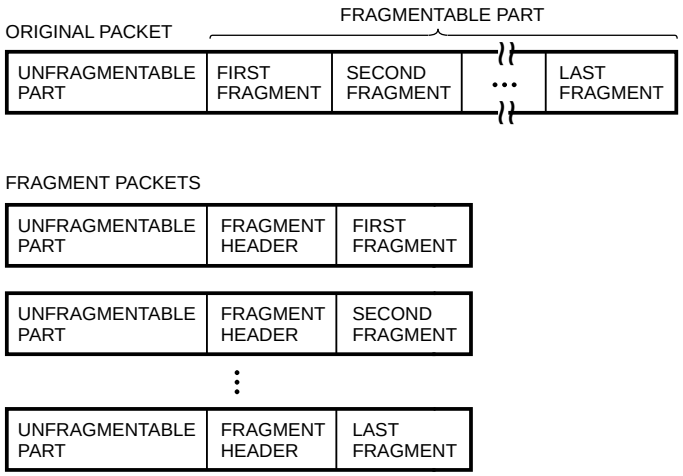

Figure 4
Fragmentation

packets, while the latter avoids undesirable fragment loss (due to the fragment packet being too big). The Digital UNIX IPv6 prototype supports either choice on a system-wide, per-destination, or per-socket basis. This is an example of separation of mechanism from policy, a basic guideline being used across this project.

Reassembly

The reassembly process reconstructs the original packet from fragment packets. Fragments belonging to the same packet are identified by a combination of source IP address, next header type (first header of the fragmentable part) and fragment identifier. Individual fragments are queued within the network layer until the original packet can be completely reassembled, at which point it is passed to the appropriate protocol module.

When all fragments have arrived, the original packet can be reassembled. A single copy of the unfragmentable part is kept, and the data from each fragment packet is appended. The payload length field of the IPv6 header is updated to reflect the length of the reassembled packet, and the next header field of the last header of the unfragmentable part is restored to reflect the first header in the fragmentable part.

As with the fragmentation code, care must be taken so that fields being updated are not in buffers shared with other copies of the packet.

When the first fragment of a packet arrives, a timer is started. If the timer expires before that packet is complete, the fragments are discarded. If the offset zero fragment has been received, an ICMPv6 error message is sent.

Forwarding and Routing

If a received packet does not match one of the system's addresses and the system is not acting as a router, the packet is silently dropped. Otherwise, an attempt is made to forward the packet. The first step in forwarding is to do a lookup in the routing table; the type of lookup depends on whether the packet contains a nonzero flow label. If it does, the lookup is based on both the source address and the flow label; otherwise the destination address is used. If the lookup succeeds and the length of the packet fits within the MTU of the resultant route and interface, the packet is transmitted to the next hop as indicated by the route. Otherwise an appropriate ICMPv6 error is sent back to the originating node.

Tunnels

Tunneling is a mechanism that allows packets of one network type to be encapsulated and forwarded within a network layer packet of the same or a different type. IPv6 packets can be tunneled over either IPv4 or IPv6 networks, as may IPv4 packets.[16,17] The tunneling routine takes as input a packet, prepends the appropriate IP header for the network over which the packet will be tunneled, and transmits the resultant packet over that network. Tunnels are unidirectional; there need not be a corresponding tunnel in the reverse direction.

Rather than having multiple tunnel interfaces (one for each possible combination of protocol Y over protocol X), the Digital UNIX implementation uses a single tunnel interface. This method was the suggestion of Keith Sklower of the University of California at Berkeley.[18] When the interface is initialized, only automatic tunneling of IPv6 over IPv4 is enabled.[19] To configure a static tunnel, where fixed end points are used, a static route is added to the routing tables with the proper destination and gateway (tunnel end point) addresses.

When a packet is presented to the tunnel interface, it looks up the route entry of the destination address. The route contents tells the tunneling routine how the packet is to be encapsulated and forwarded. The route's gateway address indicates what underlying network to use, and the route's destination address indicates what type of packet is being tunneled.

When a tunneled packet is received, the initial header is stripped and the resulting packet is placed on the appropriate IPv6 or IPv4 ifqueue.

Transports

One of the strengths of the IPng effort was the commitment to preserve the well-understood transports, TCP and UDP, upon which a wealth of applications have been built.

The IPv6 specification calls for three particular requirements of upper-layer protocols:

1. The pseudoheader checksum must accommodate larger addresses.
2. The maximum packet lifetime is no longer computed.
3. The larger IPv6 header(s) must be taken into account when computing the maximum payload size (e.g., TCP's maximum segment size [MSS]).[4]

In addition to these mandated modifications, we had to make a fundamental design choice. With two different network layer protocols in the system, each using a different size address, our design choice could have been to use two independent transport modules, one for each network layer. Figures 5 and 6 show the independent versus the integrated transport design options.

Although the independent model offers an element of design simplicity, it wastes memory by duplicating each transport layer function. In the Digital UNIX implementation, these modules are implemented in the kernel, and duplication would be expensive. Also, the design and use of a single programming interface to access both sets of services would be complicated.

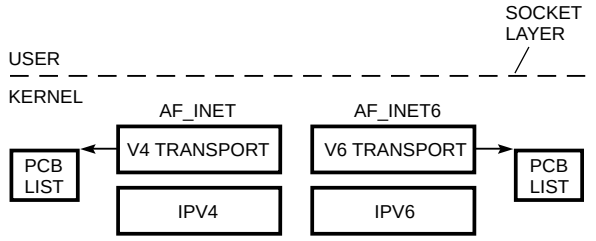

Figure 5
Independent Transport Implementation

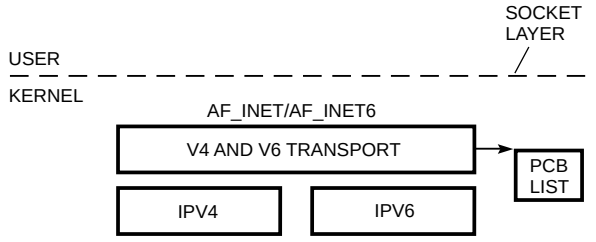

Figure 6
Integrated Transport Implementation

The ability to maintain, let alone extend, the code base would also suffer. Fortunately, due to the fact that IPv4 addresses are a well-defined subset of the entire IPv6 address space, it is relatively straightforward to implement the transports so that a single set of modules can be used over both network layers.[20] To accomplish this, we increased the storage space allocated for addresses and separated those functions that are dependent upon a particular network layer. We discuss each of these issues in this section.

Storing Large Addresses

Two specific data structures must be modified to accommodate addresses larger than the 32-bit IPv4 type. The first of these is the sockaddr struct, which is used when dealing with the BSD socket layer and passed along to user applications. The second is the Internet Protocol Control Block (PCB) data structure, the in_pcb. In this section, we review the modifications to each structure.

A program that uses a transport does so by means of the BSD sockets interface and passes addressing information in a sockaddr structure. For IPv6, this is a sockaddr_in6. Internally, the structure is defined so that 64-bit alignment is preserved; however, it has the following public definition:

```
struct sockaddr_in6 {
        u_char  sin6_len;
        u_char  sin6_family;
        u_short sin6_port;
        u_int   sin6_flowlabel;
        struct  in6_addr sin6_addr;
};
```

Although the concept of a sockaddr is generic in the BSD architecture, the flow label and in6_addr members of this structure are unique to IPv6 and would be used only in the AF_INET6 address family. The details of this are specified in Reference 21.

The in_pcb data structure is created for each socket using TCP, UDP, or other clients of the network layer. In addition to storing the source and destination addresses, various other pieces of information required for proper communication are stored here, including the port numbers, options and flags, a pointer to the socket receiving the data, a header template, and a pointer to the routing entry for the given destination. For IPv6, this basic model has been retained, and additional information is stored. This information includes local and remote flow labels and indicators of which address family the application is using and which network layer the transport communication is using. Finally, a partial checksum of the transport pseudo-header is stored here as well; its use is described in the following section.

In addition to the explicit storage of the network layer and address family, the fundamental technique that facilitates the use of a common transport is the storage of IPv4 addresses in an IPv6 format. This is known as an IPv4-mapped address and is described in "IP Version 6 Addressing Architecture."[20] This address format is explicitly reserved to store addresses of systems that are capable of using only the IPv4 protocol, and thus is an appropriate form of storage in the PCB for communications that will be sent using the IPv4 protocol, as opposed to IPv4-compatible addresses, which are sent using IPv6 packets. These mapped addresses are of the following form:

```
0000:0000:0000:0000:0000:FFFF:204.123.2.75
```

These addresses are manipulated within the IPv4 TCP and UDP protocols by means of macros that allow the IPv4 addresses to be inserted, extracted, or compared while in an IPv6 address structure (in6_addr). As an example, the code fragment in Figure 7 shows an address being extracted for use in evaluating a configurable IPv4 socket option.

Special Transport/Network Layer Interactions

Within the integrated transport layers, the transport protocol is treated independently of the particular network layer being used, and network-layer-specific functions are used to interface to either IPv4 or IPv6.

There are two particular instances in which the transport layer has interactions with the IPv6 network layer over and above the exchange of data packets for input or output. These are the notification and update of path MTU, which is required in IPv6, and the potential to refresh the neighbor discovery cache based on forward progress; i.e., if the transport knows that data is reaching its destination, it can validate the

```
/*
 * Test address for IPv4 characteristic
 */
if (inp->inp_netlayer == AF_INET) {
          struct in_addr tmp;

          tmp.s_addr = IN6_EXTRACT_V4ADDR(inp->inp_faddr);
          if (!in_localaddr(tmp))
              :
              :
}
```

Figure 7
Code Fragment of a IPv4-mapped Address

current network layer path. We investigate each of these issues in turn.

Path MTU discovery, as previously described, is triggered by ICMP messages processed in the network layer, with learned information stored in the routing table. In the course of processing a PTB message, the transport layer is notified through its control input (ctlinput) path. This is required because the reception of such an ICMP message indicates that the packet in transit has been discarded, thus the protocol may need to take appropriate action. In the case of TCP, it is necessary to recompute the maximum segment size and retransmit the affected data. Although this is not required for UDP, which is a pure datagram service, this knowledge can be made available to the corresponding socket owner.

The other interaction between an upper layer and the IP layer occurs when the upper layer, specifically the TCP transport, wishes to indicate that communications with a destination host has made forward progress, for the purpose of resetting the timer in the neighbor discovery cache. This positive feedback mechanism is described in the neighbor unreachability detection portion of the "Neighbor Discovery for IP Version 6" specification and prevents unnecessary probing of the current path.[10] When acknowledgments to previously sent data have been received, the TCP updates the routing table entry by means of an RTM_CONFIRM message. This call is handled by the neighbor discovery module, which resets the internal neighbor discovery state for appropriate route entries, as described later in this section.

Source Address Selection

Many applications do not specify a particular source address to use when initiating communications with a remote host but instead use the symbol INADDR_ANY, which allows the transport to select a source address (and corresponding interface) to use. For most IPv4 systems, this is a trivial exercise if only a single address on a single interface exists. However, multiple addresses per interface will be a common occurrence on IPv6 hosts, and so the process of choosing a source address to use becomes important. The source address selection is typically done when the PCB is bound to the application's socket, but this function is also available to users of raw sockets and to other network-layer users such as ICMPv6.

The source address selection function takes as arguments a destination address and an optional interface pointer. The latter is used when known, but in the case of initiating a transport connection it is null. The destination address is first checked against the list of current prefixes that have been advertised on the host's links, which would indicate which interface to use. (It also indicates that the destination is a potential neighbor, but this knowledge is not used at this point.) Next, the address is tested for multicast versus unicast, and then the scope (link-local, site-local, organization-local, and global) is evaluated. Finally, a local address of equivalent (or greater) scope than the destination with the longest prefix match is returned. The scope must be taken into consideration to ensure that the destination system will be able to successfully respond to the communication. The longest prefix match is an attempt to ensure a reasonable routing path between the two systems, which could involve a complex mix of service providers.

Checksum Optimization

Although the IPv6 header itself does not contain a checksum, the TCP, UDP, and ICMPv6 protocols do require a 16-bit one's complement checksum calculation to validate the integrity of transmitted and received data. Performing this checksum can be an expensive operation. While this prototype was being developed, some alternative mechanisms were investigated, such as varying the size of the sum variables and operand units and tight versus expanded loops. The selected algorithm offered the best performance for the Alpha processors, while retaining the ability to be used on 32-bit processors.

At the point where the PCB is established for transport communications, a partial checksum is calculated

for the IPv6 pseudoheader, which consists of the source and destination addresses, the packet payload length, and the next header value. This partial checksum, with the exception of the payload length (which varies per packet), is then stored in the PCB, to be passed along with the pointer to user data within the memory buffer to the checksum function. The initial checksum calculations are done using 32-bit values in 64-bit registers, and later are collapsed to the final 16-bit sum. This is coded as one large C statement, adding the various pseudoheader components in piecemeal fashion. This allows the compiler to schedule the instructions for optimal performance. The final packet checksum can then be computed by adding the partial checksums for the pseudoheader with the checksum values for the data itself, plus the payload length.

Neighbor Discovery Overview

The Neighbor Discovery specification describes several important aspects of an IPv6 node's behavior in relation to other IPv6 nodes connected to a common link. IPv6 nodes on the same link use neighbor discovery to discover each other's presence, to determine each other's link-layer addresses, to find routers, and to maintain reachability information about the paths to active neighbors and remote destinations.[10] These functions are performed with several ICMPv6 messages and options, as shown in Figure 8. The same messages are also used for address autoconfiguration and duplicate address detection, as described in "IPv6 Stateless Address Autoconfiguration."[7]

Interface Initialization

When an interface is initialized for use with IPv6, a link-local address may be created without any external configuration, allowing the system to begin transmitting and receiving packets to nodes sharing a common link. This is performed by appending an interface token to the predefined link-local address prefix, FE80::. The length and content of the interface token is link specific. For example, the address token for an Ethernet interface is the interface's built-in 48-bit IEEE 802 address, resulting in a link-local address such as FE80::0800:2BBE:6260.[22]

Duplicate Address Detection

Before a unicast address can be assigned to an interface, a process known as duplicate address detection (DAD) must be performed.[7] This process provides a degree of assurance that two nodes do not use the same address on the same link. The basic mechanism involves sending an ICMPv6 neighbor solicitation (NS), where the target address is the address being tested. If another node is using the address, it will respond with a neighbor advertisement (NA). Multicast is used for both NS and NA packets, so DAD can be performed for all unicast addresses, including the first address assigned to the interface.

While an address is undergoing DAD, it is said to be a tentative address. It is not used to receive packets, nor is it used in outbound packets. The LA6_TENTATIVE flag in the in6_localaddr structure identifies addresses in this state. When a duplicate address is detected, the error is logged and the LA6_DADFAILED flag is set in the in6_localaddr structure. If a duplicate address is not detected, the LA6_TENTATIVE flag is cleared, making the address available for use on the interface.

Address Resolution

In IPv6, the function of mapping unicast IPv6 addresses into link-layer addresses is performed using ICMPv6 messages. This is a departure from IPv4, which relied on separate protocols (e.g., Address Resolution Protocol [ARP]) to perform this function.[23] IPv6 unicast address resolution is defined in a generic manner and can be run over any link layer that provides a link-layer multicast service (this includes point-to-point and broadcast links, special cases of multicast). This protocol may also be used for nonmulticast-capable media (e.g., nonbroadcast multiple access [NBMA] media such as ATM), provided that the link layer provides the necessary services. The function of mapping multicast IPv6 addresses into link-layer addresses is specific to each link type.

The unicast address resolution function uses two ICMPv6 message types: the NS and the NA. When a node needs to resolve the unicast IPv6 address of a neighbor to a link-layer address, it builds an NS containing the IPv6 address to be resolved (target) and sends it to the solicited-node multicast address corresponding to the target address. As an optimization, the node includes its own link-layer address as an option in the NS message.

When an address is assigned to an interface, a node is required to join the solicited-node multicast group corresponding to that address, so a node will receive NSs sent to its solicited-node multicast address. Upon receipt of an NS, the target node builds an NA containing its link-layer address. The NA also contains the IPv6 target address, so that the soliciting node can associate the response with the corresponding request. The NA is then sent back to the soliciting node.

Upon receipt of an NA, the soliciting node can map the target IPv6 address to the corresponding link-layer address and send any packets that were queued awaiting address resolution. Once a node has resolved an IPv6 address, the link-layer address is cached until it must be replaced or deleted. A different link-layer address may be received in a subsequent NA packet, with the O-bit (override flag) set to indicate a new value. If the neighbor unreachability detection algorithm (explained in the next section) detects that the cached

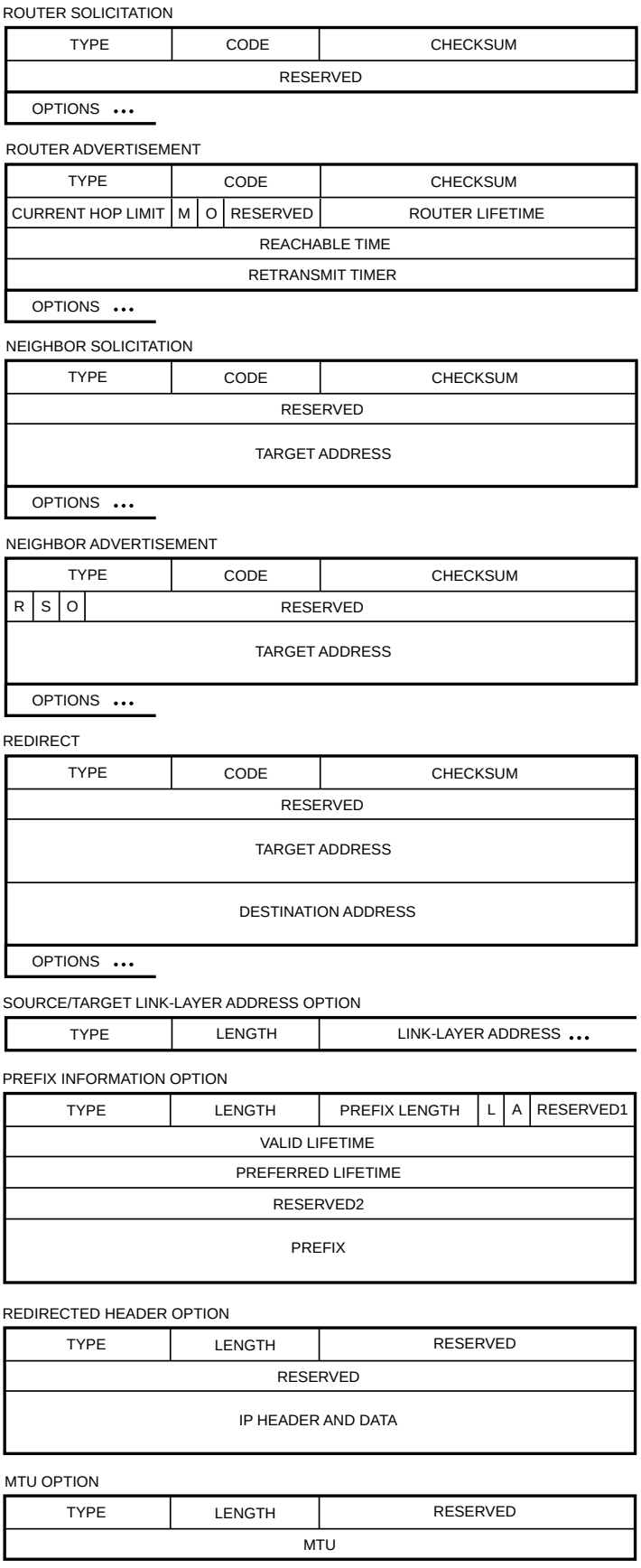

Figure 8
Neighbor Discovery Packets

value is not reachable, the mapping will be deleted.

The address resolution process has several implications for the implementation. Outbound packets must be queued pending link-layer address resolution, and link-layer addresses must be stored somewhere. The "Neighbor Discovery for IP Version 6" specification describes a conceptual neighbor cache to hold this information.[10] The Digital UNIX IPv6 prototype uses several data structures to implement the neighbor cache. An nd6_llinfo structure keeps track of each entry in the neighbor cache. This structure contains the queue header for packets awaiting link-layer-address resolution. The link-layer address is stored in the routing table, in a host route entry for the destination IPv6 address. The RTF_LLINFO flag in the route entry indicates the presence of link-layer information. Each nd6_llinfo structure contains a pointer to the corresponding routing table entry, and the routing table entry points back to the nd6_llinfo structure.

The use of routing table entries to hold the link-layer-address information is an optimization. A routing table entry is associated with the majority of packets transmitted for reasons other than address resolution. Storing the link-layer address in the routing table entry avoids the overhead of a separate link-layer-address table. This approach is modeled after the BSD 4.4 system's ARP implementation.

Neighbor Unreachability Detection

Neighbor unreachability detection (NUD) has its roots in the dead gateway detection in IPv4 but has been generalized in IPv6 to include all neighboring nodes (not just gateways).[24] Unlike IPv4, the mechanisms supporting NUD are an integral part of IPv6. IPv6 nodes monitor the reachability of neighboring nodes to which packets are being sent. An IPv6 node relies on reachability confirmations to determine the reachability state of a neighbor. In the absence of any reachability indications, an IPv6 node will periodically use an NS to actively probe the reachability of a neighbor. An NA sent in response to an NS provides reachability confirmation. The S (solicited) flag in the NA is provided specifically for this purpose. If neither method succeeds within a given period of time, a neighbor is considered unreachable. Figure 9 shows the neighbor unreachability states.

A reachability confirmation may take several different forms. Any packet received from a neighbor can be viewed as a reachability confirmation, provided that the packet would only have been sent by the neighbor in response to a packet sent from the local node. A TCP acknowledgment is one example: receipt of a TCP ACK indicates that a packet sent to the neighbor did in fact reach it. Another example is an ICMPv6 redirect message. Receipt of a redirect message indicates that the neighboring router received a packet from the local node.

In the Digital UNIX IPv6 prototype, the nd6_llinfo structure holds NUD state and retransmit count information. A field in the routing table entry is used for NUD timers. The RTF_LLVALID flag in the route entry is used to indicate that the neighbor is reachable. A new routing message type (RTM_CONFIRM) was defined to pass reachability confirmations to the neighbor cache. This mechanism is used by TCP upon receipt of new acknowledgments.

Autoconfiguration

One of the goals of IPv6 is to work properly in a dynamic network environment without the need for manual intervention on each system attached to the

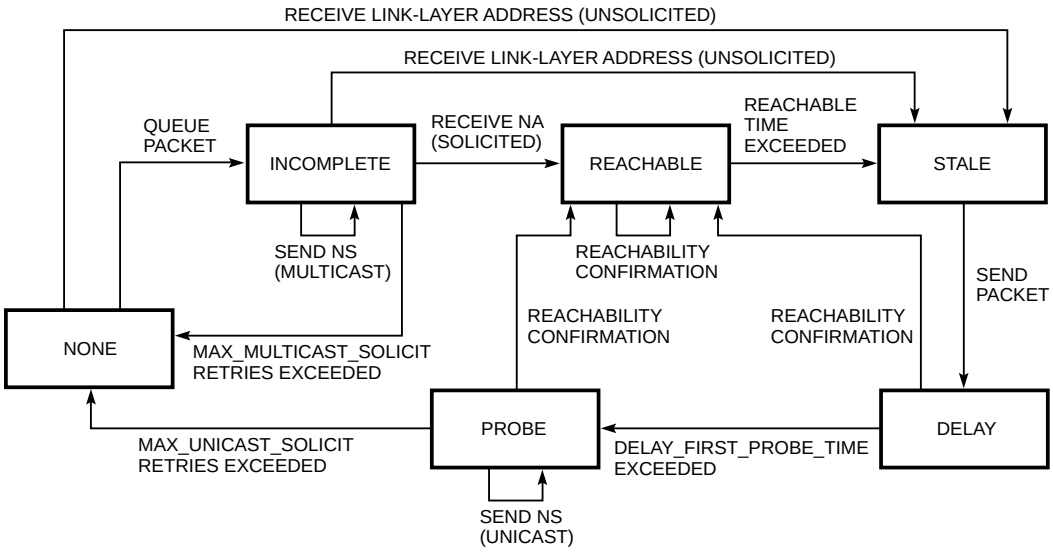

Figure 9
Neighbor Unreachability States

network. The solution is to allow important pieces of information to be learned and the system to autoconfigure itself using this data. IPv6 autoconfiguration encompasses the following items:

- Router discovery
- On-link prefix discovery
- Interface attribute configuration
- Stateless address configuration
- Stateful address configuration

The mechanism for delivering this information to the hosts is the router advertisement (RA) packet of the Neighbor Discovery Protocol. In the following sections, we describe the methods we developed to process these packets and update the system.

Host Autoconfiguration Daemon

To process these RAs, we designed a host daemon called nd6hostd, which resides in the application space of the Digital UNIX operating system. We determined that a user-mode daemon was the most efficient way to implement IPv6 autoconfiguration for the following reasons:

- A user-mode daemon would avoid kernel bloat.
- Maintenance and extensibility would be easier.
- The function is not performance critical.

The autoconfiguration processing is implemented as a single executable image, as a cohesive set of tightly coupled modules. The daemon currently is designed as a single-threaded application that uses a dispatch mechanism to call each specialized function module in turn. We will examine the idea of having this daemon run as a multithreaded application in the future.

The nd6hostd daemon communicates with the network subsystem in the kernel through multiple techniques. Figure 10 shows the autoconfiguration processing modules. The raw socket interface is used to receive RAs, and I/O control messages (ioctls) are used to manipulate kernel data structures. Also, the routing table is updated as necessary, by means of a raw socket interface to the PF_ROUTE protocol family.

We designed the IPv6 raw socket's interface with the ability to pass only specific ICMPv6 messages to a user and to filter extraneous packets or protocols. The nd6hostd daemon sets a socket option to receive only neighbor discovery RAs. It then executes a dispatch routine that polls the raw socket, awaiting packets. When data is available on the socket, the daemon determines the characteristics of the message, creates a data structure to contain it, and calls the necessary functions to perform autoconfiguration. The dispatch module, in addition to polling socket descriptors, supports necessary timer management functions such as creation, deletion, and expiration. Figure 11 shows the application daemon design center.

Kernel Interface Data Structures

In many ways, the data link interface is the focus of IPv6 autoconfiguration support. The kernel data structures for IPv4 interfaces are not sufficient to implement the necessary IPv6 functions. We designed and implemented new interface data structures that encapsulated the existing IPv4 structures. This allowed us to avoid a recompilation of the existing data link drivers on the Digital UNIX operating system. In the future, we will attempt a design in which the interface structures for IPv4 and IPv6 are completely integrated.

As shown in Figure 12, we designed an in6_ifnet structure to support each data link type (e.g., Ethernet, PPP, loopback) and used the existing ifnet structures to point to those link interfaces. The in6_ifnet has its own in6_ifaddr structure for each IPv6 address configured in the data structure in6_localaddr. We also defined the in6_router structure to support each router available for the implementation. The in6_router structure specifies the interface of the router, neighbor cache route, and the IPv6 address of the router.

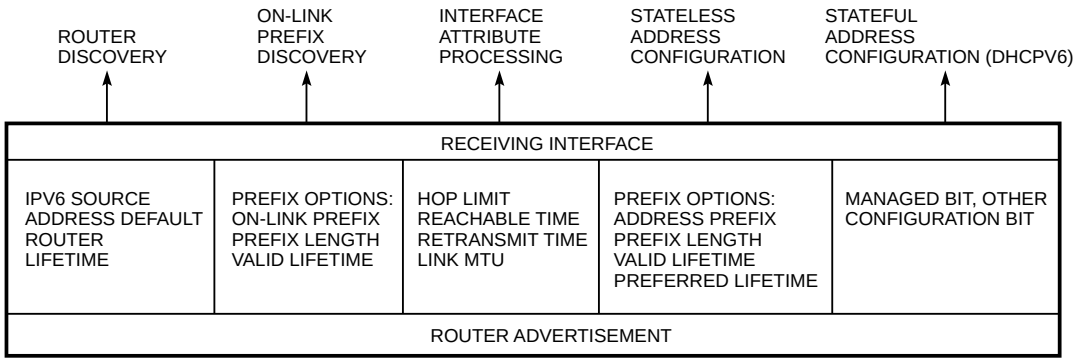

Figure 10
Autoconfiguration Processing Modules

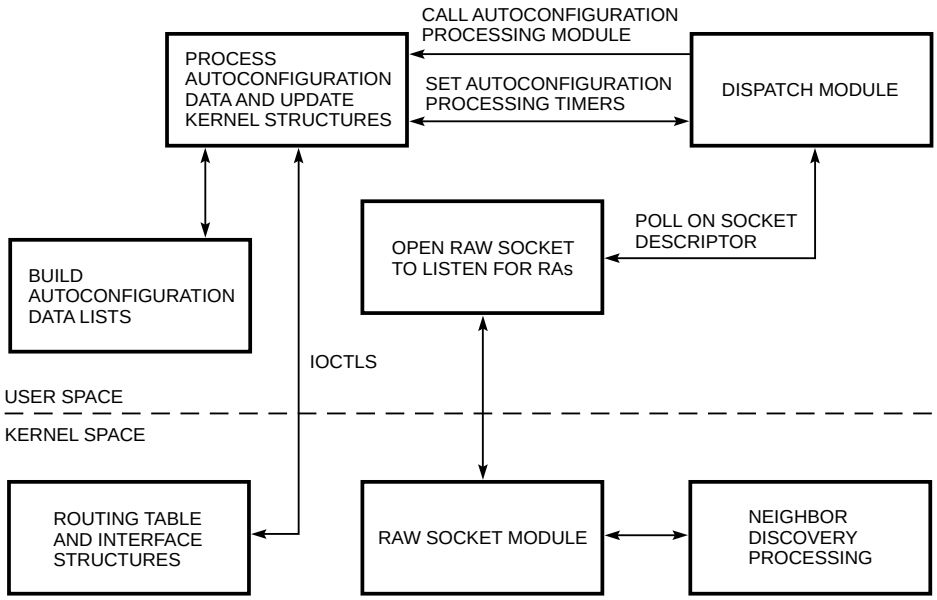

Figure 11
Application Daemon Design Center

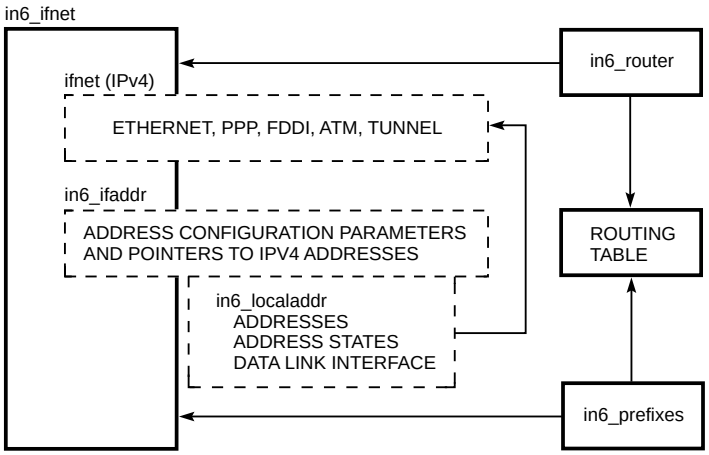

Figure 12
Autoconfiguration Interface Structures and Relationships

Interface Attribute Autoconfiguration

To autoconfigure the interfaces for IPv6, we created new ioctl functions to create, delete, update, and access the interfaces. In addition to their use by the nd6hostd daemon, these ioctls may be used by any future modules that need to access or manipulate the interfaces. This might include specialized configuration utilities, Simple Network Management Protocol (SNMP) management functions, security tools, or other services.

The interface module to update and maintain interface structures for nd6hostd serves two purposes: to update data link attributes provided by the RA, and to maintain the data structures as a set of linked lists for router discovery, on-link prefixes, and address configuration. Figure 13 shows the interface attribute updates.

Router Discovery

An RA packet has mandatory and optional parts. Before a default router is added to the routing table, the following interface attributes must be determined:

1. Receiving interface
2. Current hop limit
3. Reachable and retransmit times for use in NUD

The link-local address from the source link-layer option of the RA is then added to the routing table,

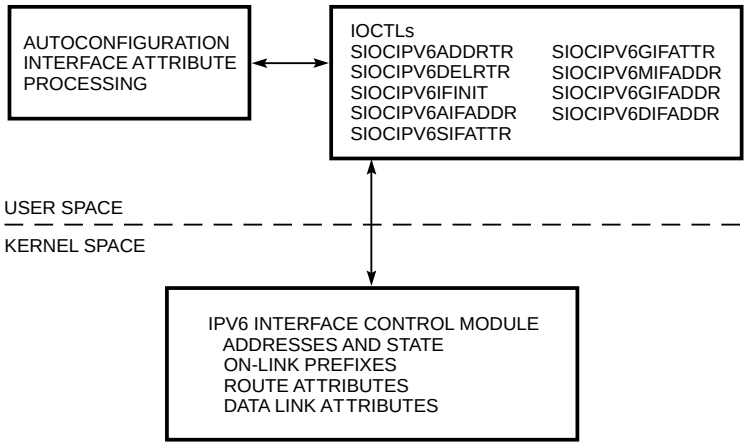

Figure 13
Interface Attribute Updates

and the kernel data structures for router information are updated. The router lifetime field in the RA defines how long this router may be used as a default router.

The nd6hostd daemon first updates the interface attributes. A timer is set using the appropriate routine from the dispatch module. When the timer expires, the delete default router routine is called, and the router is deleted from the routing table. The daemon must also be able to delete the router if it receives an RA with a zero lifetime value, which can occur when a node is acting as a router but is reset to be a host.

On-link Prefixes

An on-link prefix in IPv6 defines a subnet and is typically configured on a router for a specific link by the network administrator. The router then advertises this prefix to all nodes connected to that link as a prefix option, appended to an RA. A prefix option defines a single prefix only, but an RA may contain more than one such option. As shown in Figure 8, the prefix option provides the following information:

- Prefix length
- Link- or L-bit, which is set if the prefix is directly readable on link (i.e., a neighbor)
- Autonomous- or A-bit, which is set if the prefix can be used for stateless address configuration
- The length of time the prefix is valid

The daemon adds the prefix to the routing table. Then a timer routine is called from the dispatch module and is set for the time the prefix is valid. When the dispatch routine calls the delete on-link prefix module, the prefix is deleted from the routing table. A prefix can also be deleted when a new RA presents the prefix with a lifetime of zero. In that case, the on-link prefix module will stop the timer routine and delete the prefix from the routing table.

Address Configuration

Address configuration is one of the new paradigms that must be supported in IPv6. Two configuration methods, stateless and stateful, are provided to autoconfigure addresses for a host. The M-bit flag in an RA message determines which method to use and informs a host. In addition, the other-bit (O-bit) flag is provided to configure other network parameters required for the host's operation on the network when the stateful configuration is used.

Address autoconfiguration in IPv6 supports the ability to dynamically renumber a link or a complete network through the use of lifetimes specified in the RA message. The valid lifetime is the time the address has before expiration. When the timer expires, all connections using that address are dropped by the implementation, and no new connections are permitted. The preferred lifetime is provided to inform an implementation that an address is about to expire; it typically is set to a lower value than the valid lifetime. When this timer expires, the address is said to enter the deprecated state, at which point an implementation is permitted (as a configuration option) to prevent new communications using this address as a source or destination. This model is designed to provide network administrators with control over the use of network addresses without manual intervention of each host on the network. The stateless model is intended for users who do not need tight control over address configuration; stateful mechanisms will be used where the administrators want to delegate addresses based on a client/server method. Figure 14 shows the address autoconfiguration diagram.

When the daemon receives an RA, and the A-bit is set, the daemon can use the prefixes provided to perform stateless address configuration. The daemon uses the on-link prefix(es) provided in the RA to configure addresses for an interface. Addresses are created,

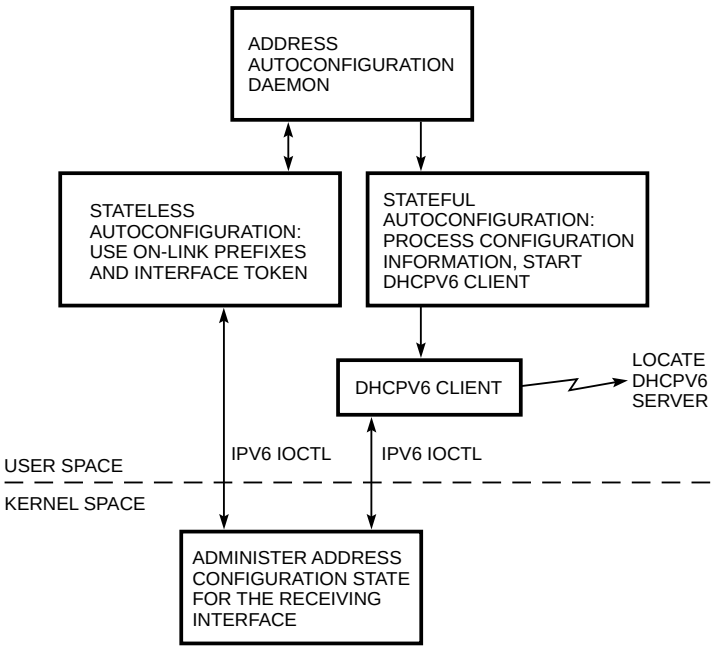

Figure 14
Address Autoconfiguration

deleted, or updated on the interface based on the prefixes and lifetimes received in the RA packet.

Interactions with Stateful Address Configuration

When the daemon receives an RA, and the managed bit flag is set, the host can use stateful address configuration, using DHCPv6. DHCPv6 is implemented as a separate daemon process in our prototype. DHCPv6 defines a complete new model from the existing DHCPv4 implementations in the industry to dynamically configure addresses. The use of link-local addresses, multicast, address configuration, and inherent support for dynamic renumbering of hosts in IPv6 caused a new architecture and design in the DHCPv6 specification. A comparison of the architectural changes between DHCPv4 and DHCPv6 can be found in the DHCPv6 specification.[8]

Application Services

Most TCP and UDP applications can be used with IPv6 with relatively minor modifications. The primary issue is the larger address size, both for internal storage needs in the application and for address transfer across system interfaces. In this section, we review these issues and others.

API

Any API currently in use for IPv4 could be modified for IPv6, but only the BSD sockets API is being investigated within the IETF for two reasons.[21,25] First, large numbers of applications use the sockets interface for IPv4, which represents a very large investment and a potential pool of IPv6 applications. Second, this API is perhaps in the most widespread use in the industry and is available on a wide variety of platforms: the benefits of standardization are compelling.

DNS AAAA Support

DNS provides support for mapping names to IP addresses and mapping IP addresses back to their corresponding names.[26] The type A resource record is used to hold an IPv4 address. Since its size is fixed at 4 bytes, a new resource record type, AAAA, was defined to hold IPv6 addresses.[27] The Digital UNIX IPv6 prototype includes a widely used implementation of the DNS known as Berkeley Internet Name Domain (BIND), which has been modified to support AAAA records.

Address Manipulation Routines

A typical IP implementation provides several library routines for manipulating IP addresses. These include routines for converting addresses between binary and textual representations and routines for translating names to addresses and addresses to names. New routines had to be provided to perform these functions for IPv6 addresses. The Digital UNIX IPv6 prototype provides the routines described in "Basic Socket Interface Extensions for IPv6."[21]

inetd Daemon

The inetd daemon creates sockets on behalf of applications, invoking the applications only when needed and

passing the open sockets to them. With the advent of the AF_INET6 socket type, inetd was modified to accept a new application configuration option in its configuration file. The keyword inet6 is used to indicate an application that wants to use AF_INET6 sockets. The keyword inet (or the absence of a keyword) indicates use of AF_INET sockets.

Applications

A typical application needs only minor modification to use the AF_INET6 address family. Applications that use addresses as part of their design or protocol, such as the File Transfer Protocol (FTP), require more extensive modification. The Digital UNIX IPv6 prototype includes several basic applications that have been modified to support IPv6, including Telnet and FTP. These programs were modified to use IPv6 sockets, address structures, and library routines. Note that the IPv6 sockets also support communications over IPv4, so that applications need not maintain separate sockets for IPv4 and IPv6, and a single executable image can interoperate with both types of remote system.

Future Work

Future implementation efforts will include security, routing, stateful address configuration, dynamic updates to DNS, IPv6 over PPP and ATM, resource reservation, and service location. In addition, we will review elements of our existing design and implementation architecture to increase performance and to ease the transition from IPv4 to IPv6. We will continue to participate in the IPv6 industry multivendor interoperability events, which is a practical and concentrated effort to debug the specifications and the code base.

IPv6 security supports both the authentication and the encryption of IPv6 packets end-to-end.[28] The module for these functions will reside in the kernel and most likely will be called at the point where the IPv6 network layer packet is processed. A key management framework is being developed to support both authentication and encryption. To access the key management interface, a sockets API extension will be provided to supply the keying criteria for the security modules.

To test the interoperability and robustness of the IPv6 implementations, a test network known as the 6BONE has been created on the Internet. This nascent test bed is currently being built with statically defined tunnels connecting IPv6 networks. Our next step in IPv6 development will be to implement routing protocols, starting with Routing Information Protocol version 6 (RIPv6) for unicast routing. Subsequent goals will be to support Open Shortest Path First version 6 (OSPFv6) and to provide multicast routing.

Stateful address configuration will be implemented as specified in DHCPv6 and will contain a client, a server, and a relay-agent. This work will be tightly coupled with dynamic updates to DNS to provide autoconfiguration in conjunction with autoregistration in the directory service. Even for networks that use stateless address autoconfiguration, DHCPv6 will be available to configure other parameters for the host and to add, delete, and update name information associated with addresses in DNS.

Additional data link interfaces will be supported for PPP and ATM. These nonbroadcast architectures will require some design analysis to implement in order to support neighbor discovery, autoconfiguration, and the routing models for IPv6. Digital has been active within the IETF working groups that are defining the ATM solutions.

IPv6 now supports flow information in the IPv6 header and in the IPv6 BSD socket API structure. This inherent quality-of-service (QOS) mechanism in IPv6 meshes well with efforts to support reserve resources on a network as specified in the Resource Reservation Protocol (RSVP).[29] Using RSVP over broadcast and nonbroadcast data links will encompass a design center that supports a wide range of resource reservation parameters to maintain a consistent performance model for video- and audio-related applications across a network path.

Service location is an emerging technology that will permit a host to query the network about the location of different services (e.g., NFS, security key management, directory services).[30] Currently in development for IPv4, service location holds promise for IPv6 and may benefit from the greater level of support for basic technologies, such as security and multicast capabilities.

Summary

Digital has designed a prototype of IPv6 on the Digital UNIX operating system. Techniques and technologies have been developed to accommodate aspects of the IPv6 architecture; in particular, the transport layer modules were modified to use two distinct network-layer protocols. The new Neighbor Discovery Protocol and algorithms have also been implemented in the prototype. IPv6 includes mechanisms to do both stateless and stateful address configuration as well as router discovery. The Digital UNIX IPv6 prototype contains a user-mode process that implements these functions. In addition, enhancements have been made to IPv4 services, and techniques have been developed to support the transition of existing applications.

References

1. P. Gross and P. Almquist, "IESG Deliberations on Routing and Addressing," RFC1380 (November 1992).

2. S. Bradner and A. Mankin, "The Recommendation for the IP Next Generation Protocol," RFC1752 (January 1995).

3. R. Hinden, "Simple Internet Protocol Plus White Paper," RFC1710 (October 1994).

4. S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC1883 (January 1996).

5. Y. Rekhter and T. Li, "An Architecture for IPv6 Unicast Address Allocation," RFC1887 (January 1996).

6. R. Hinden and J. Postel, "IPv6 Testing Address Allocation," RFC1897 (January 1996).

7. S. Thomson and T. Narten, "IPv6 Stateless Address Autoconfiguration," RFC1971 (August 1996).

8. J. Bound and C. Perkins, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)," Work in progress (August 1996).

9. A. Conta and S. Deering, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6)," RFC1885 (January 1996).

10. T. Narten, E. Nordmark, and W. Simpson, "Neighbor Discovery for IP Version 6 (IPv6)," RFC1970 (August 1996).

11. M. McKusick et al., *The Design and Implementation of the 4.4 BSD Operating System,* (Reading, Mass.: Addison-Wesley, ISBN: 0-201-54979-4, 1996).

12. V. Fuller et al., "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," RFC1519 (September 1993).

13. S. Deering, "Host Extensions for IP Multicasting," RFC1112 (August 1989).

14. J. Mogul and S. Deering, "Path MTU Discovery," RFC1191 (November 1990).

15. J. McCann et al., "Path MTU Discovery for IP Version 6," RFC1981 (August 1996).

16. W. Simpson, "IP in IP Tunneling," RFC1853 (October 1995).

17. A. Conta and S. Deering, "Generic Packet Tunneling in IPv6 Specification," Work in progress (October 1996).

18. K. Sklower, private communication to Matt Thomas, September 1995.

19. R. Gilligan and E. Nordmark, "Transition Mechanisms for IPv6 Hosts and Routers," RFC1933 (April 1996).

20. S. Deering and R. Hinden, "IP Version 6 Addressing Architecture," RFC1884 (January 1996).

21. R. Gilligan et al., "Basic Socket Interface Extensions for IPv6," (Work in progress, April 1996).

22. M. Crawford, "A Method for the Transmission of IPv6 Packets over Ethernet Networks," RFC1972 (August 1996).

23. D. Plummer, "An Ethernet Address Resolution Protocol," RFC826 (November 1982).

24. D. Clark, "Fault Isolation and Recovery," RFC816 (July 1982).

25. W. Stevens and M. Thomas, "Advanced Sockets API for IPv6," Work in progress (October 1996).

26. P. Mockapetris, "Domain Names—Concepts and Facilities," RFC1034 (November 1987).

27. S. Thomson and C. Huitema, "DNS Extensions to Support IP Version 6," RFC1886 (December 1995).

28. R. Atkinson, "Security Architecture for the Internet Protocol," (Work in progress, June 1996).

29. "Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification," (Work in progress, August 1996).

30. J. Veizades et al., "Service Location Protocol," (Work in progress, June 1996).

General References

S. Bradner and A. Mankin, eds., *IPng—Internet Protocol Next Generation* (Reading, Mass.: Addison-Wesley, ISBN: 0-201-63395-7, 1996).

S. Thomas, *IPng and the TCP/IP Protocols* (New York: John Wiley & Sons, Inc., ISBN: 0-471-13088-5, 1996).

R. Braden, "Requirements for Internet Hosts—Communication Layers," RFC1122 (October 1989).

G. Wright and R. Stevens, *TCP/IP Illustrated, Volume 2—The Implementation* (Reading, Mass.: Addison-Wesley, ISBN: 0-201-63354-X, 1995).

Biographies

Daniel T. Harrington
As a principal software engineer in Digital's IPv6 Program Office, Dan Harrington participated in the Digital UNIX IPv6 prototype effort. Prior to this work, he helped develop the DECnet/OSI products on the ULTRIX and the Digital UNIX platforms. After joining Digital in 1982, Dan worked in performance analysis, field support, and software development. He received a B.S. in mathematics from Rensselaer Polytechnic Institute. Dan is currently with Lucent Technologies.

James P. Bound
Jim Bound is a consulting software engineer and the technical director for IPv6 within the IPv6 Program Office. Jim is responsible for the overall advanced development architecture and reference Alpha Digital UNIX code base, which verifies that the IPv6 specifications are implementable. He is also Digital's IETF IPv6 technical leader and one of the IPv6 advanced development engineers on Alpha Digital UNIX. In 1993, Jim began his participation in the IETF to work on the IPng and the advanced development IPng prototype. As a member of the IETF's IPng Directorate, Jim helped determine the requirements and core architecture for IPng's Internet protocol and related functionality to support IPng. The result was the selection of a proposal, now known as the Internet Protocol version 6 (IPv6). Jim has an A.S. in business management and an A.S. in computer science. He is a coauthor of several IPv6 specifications and a contributing author to the book *IPng: Internet Protocol Next Generation*. He is a member of the IEEE and the Internet Society.

John J. McCann
Jack McCann is a principal software engineer in the UNIX Engineering Group and a member of the IPv6 project team. He contributed to the design and implementation of the Digital UNIX IPv6 prototype, including router discovery, autoconfiguration, fragmentation, reassembly, path MTU discovery, forwarding, and the IPv6 API. He participates in several IETF working groups and is a coauthor of Internet RFC 1981, "Path MTU Discovery for IP version 6." Jack joined Digital in 1988 to become a member of the Distributed Systems Technical Evaluation Group. He also worked in the DECnet/OSI for OpenVMS Engineering Group before taking his current position. He received a B.S. in computer science (magna cum laude) from the University of Lowell in 1988 and an M.S. in computer science from Boston University in 1995.

Matt Thomas
Matt Thomas joined Digital in 1983 with Software Services in California. Although he is a principal software engineer in the OpenVMS Systems Software Group, Matt has spent the last eight years as a developer of networking products for the Digital UNIX and ULTRIX systems. In addition to his ongoing involvement with Digital UNIX IPv6 efforts, he is responsible for adding IP security to the Digital UNIX operating system. Matt is an active participant in various IETF working groups and is a coauthor of several Internet Drafts.