1-1-1992

An Introduction To The Implementation Of The User Datagram Protocol

Michael A. Craft

INSTITUTE FOR SIMULATION AND TRAINING

AN INTRODUCTION TO THE IMPLEMENTATION OF THE USER DATAGRAM PROTOCOL

BY: MICHAEL A. CRAFT

AUGUST 21, 1992

IST

B 174

An Introduction to the Implementation of the User Datagram Protocol

presented by

Michael A. Craft

representing

The Institute for Simulation and Training

August 21, 1992

Table of Contents

Scope

- Fundamental network concepts

- Implementation techniques for best effort connectionless protocols

- Implementation of UDP/IP on a host.

Much of the IP literature covers IP routers (gateways). The routers are considerably more complex than the topics covered here.

TCP is not covered here. It is significantly more complex (offering a connection based reliable network).

==========================

Some code in this paper is written in a form of pseudo-code. The majority is given as C/C++ code extracted from a running implementation. However, the extracted code was modified to remove system dependencies. Because of the edits done to code after extraction there are certainly C errors; perhaps it is best to take the C code as a form of pseudo code. In the C code "uchar" is a typedef for unsigned char, with uint and ulong being similar.

Network Basics

- International standards refer to octets rather than bytes.

- Networks allow processes on separate computers to communicate. The term network refers to the conglomerate communication facility (wires, satellites, communications software, etc.).

- Modern networks are implemented in layers. There are various layering schemes, but most, if not all, agree that the bottom layer is the physical layer, consisting of the physical medium which carries bits, how bits 1 and 0 are distinguished, and associated questions. This is the realm of electrical engineers more than computer scientists.

  Perhaps the best known and most important layered model is the ISO (International Standards Organization) OSI (Open System Interconnection) Reference model.

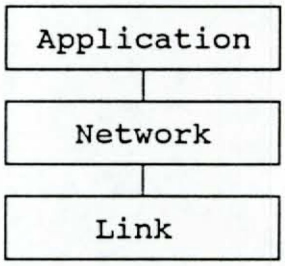
- Protocol refers to the language (rules and conventions) used for communication between corresponding layers. If there are 20 layers in a protocol stack, there will be 20 protocols.

  By necessity, early network development was at the lower layers of the protocol stack (physical and link), and network development has been bottom up (except in that the top level, the application, has always driven development).

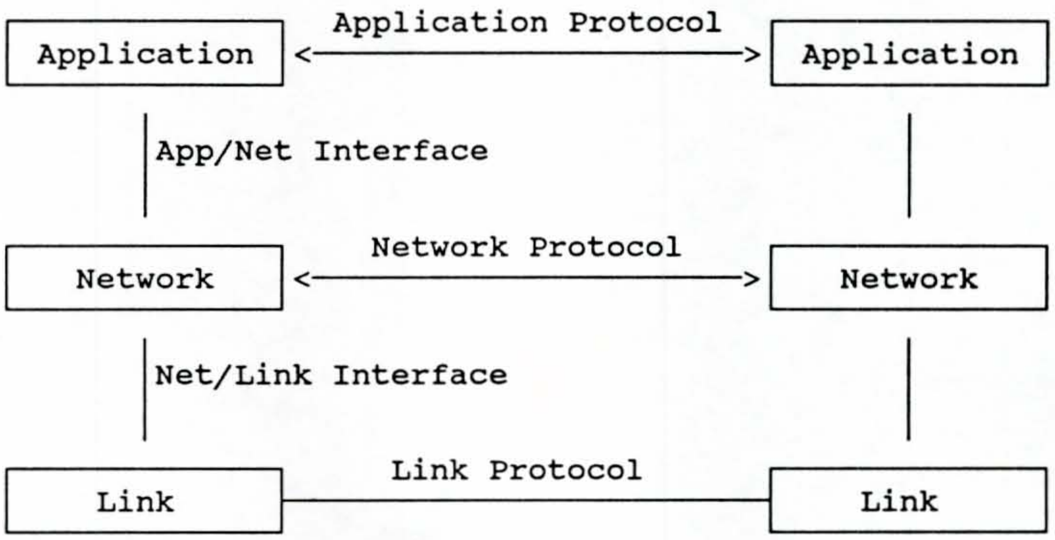
- The connections between layers of the protocol stack are called layer interfaces.

- Requests from applications to the protocol support are done through service access points (SAPs); when discussing protocols "interface" refers to inter-layer communication.

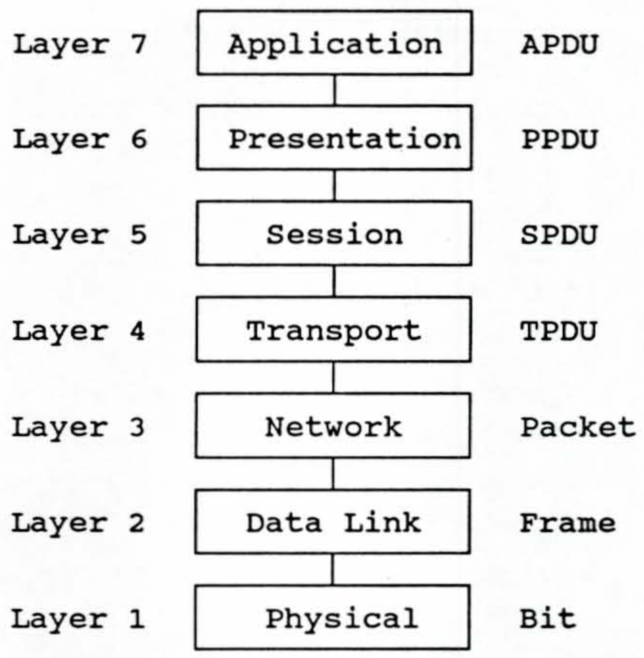
To illustrate these points, and for simplicity in our UDP/IP discussions, we will often use a 3 layer stack:



In use on two communicating machines the stack architecture looks like this:



Representations of the OSI Reference model can be found in any modern network text and, unfortunately, as boiler plate verbiage in hundreds of documents covering networks in any shape, manner or form. For the record, the OSI Reference Model Follows:



For the UDP/IP discussion at hand, the application, session, and presentation are collectively labeled "application." Transport and network are combined as "network." Data link and physical are collectively called "link."

Network Classification

Most networks can be categorized as connection based or connectionless. Roughly:

Connection based networks:

- Parallel telephone service.
- Use <u>virtual circuits</u>.
- Deliver packets (units of information) in the order sent.
- Are expected to be reliable.

Connectionless networks:

- Parallel mail delivery.
- Packets are sent as independent messages.
- Packets may be lost or discarded without notice.
- Delivery may be out of order.

Classification of Machines on a Network

There are various categories of computers connected to, and interconnecting, networks. A single machine may play several roles:

Host:	Any end user computer connected to the net.
Repeater:	copies bits from one line (physical connection) to another. Usually used to extend a LAN when cable length restrictions prevent direct connections of lines. Repeaters are associated with the physical layer.
Bridge:	Interconnects networks at the data link layer. A UDP/IP implementation on 802.3 could use an 802.3/Ethernet bridge to connect to an implementation on top of Ethernet.

The remaining terms, like so many in computer science, are not used consistently by the networking community.

Router:	Connects networks at the network layer. Often used in a much more general sense. In the IP world its use is supplanting the use of "gateway." When one hears "router" in an IP context, one normally thinks of a computer accepting packets from two or more IP networks for appropriate redistribution among those IP networks.
Gateway:	Supports connections between arbitrary nets. A gateway may connect to X.25 and IP networks, and allow traffic between hosts on these diverse technologies.

Network Address Classifications

Network messages may be addressed (indicate their destination) in various ways:

- Point to point (unicast): supported by most protocols. The protocol user specifies a unique target for the message. The type of the target depends on the protocol layer (higher level protocol targets may be specific applications, lower layer protocols may have computer hosts (computers connected to networks) for targets).

  In the IP world, IP addresses specify network/host connections (many people think the address is for a host). There is a PC at UCF with the address 132.170.191.146. Upper bits of this 4 octet value indicate the network (in this case 132.170 represents an IST network) and lower bits indicate the precise connection (191.146 is associated with a particular PC on the IST LAN 132.170). Where the network specification ends and the connector (host) specification begins depends on the class of the address (which can be determined by the upper bits of the first octet; the IST address is class B).

- Broadcast: messages are directed to "everyone." On a LAN the meaning of "everyone" is clear.

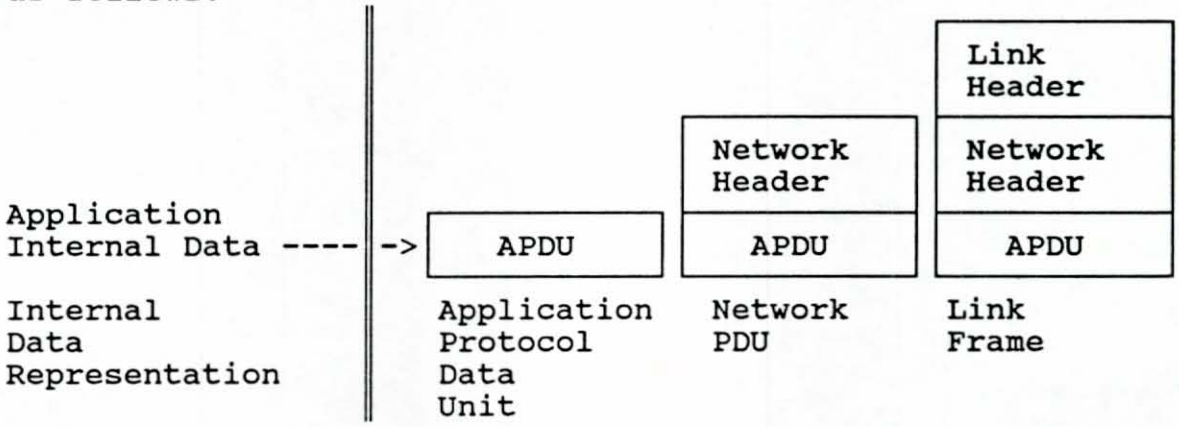
- Multicast: messages are directed to a subset of the interconnected machines.

  Class D IP addresses are multicast addresses. The IP addresses 255.255.255.255 gives what IP people call "limited broadcast" -- delivery to everyone on the local net (in general CS terms this would be a multicast send with the multicast group being the hosts on the local LAN). An address beginning with a net specification but ending with all 1's (e.g., 132.170.255.255) can be used as a directed broadcast.

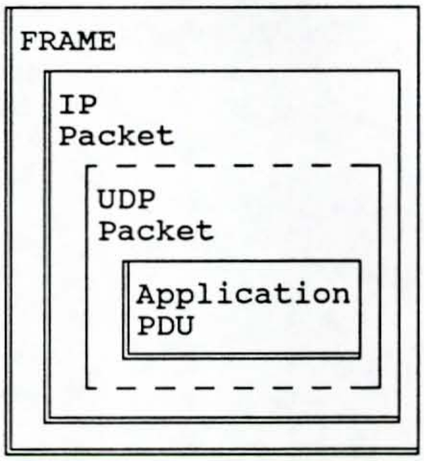
As we shall see, the IP address is enough to reach a host, but does not distinguish between processes on the host. The UDP "port" (effectively an address) identifies an application.

Protocol Layer Envelopes

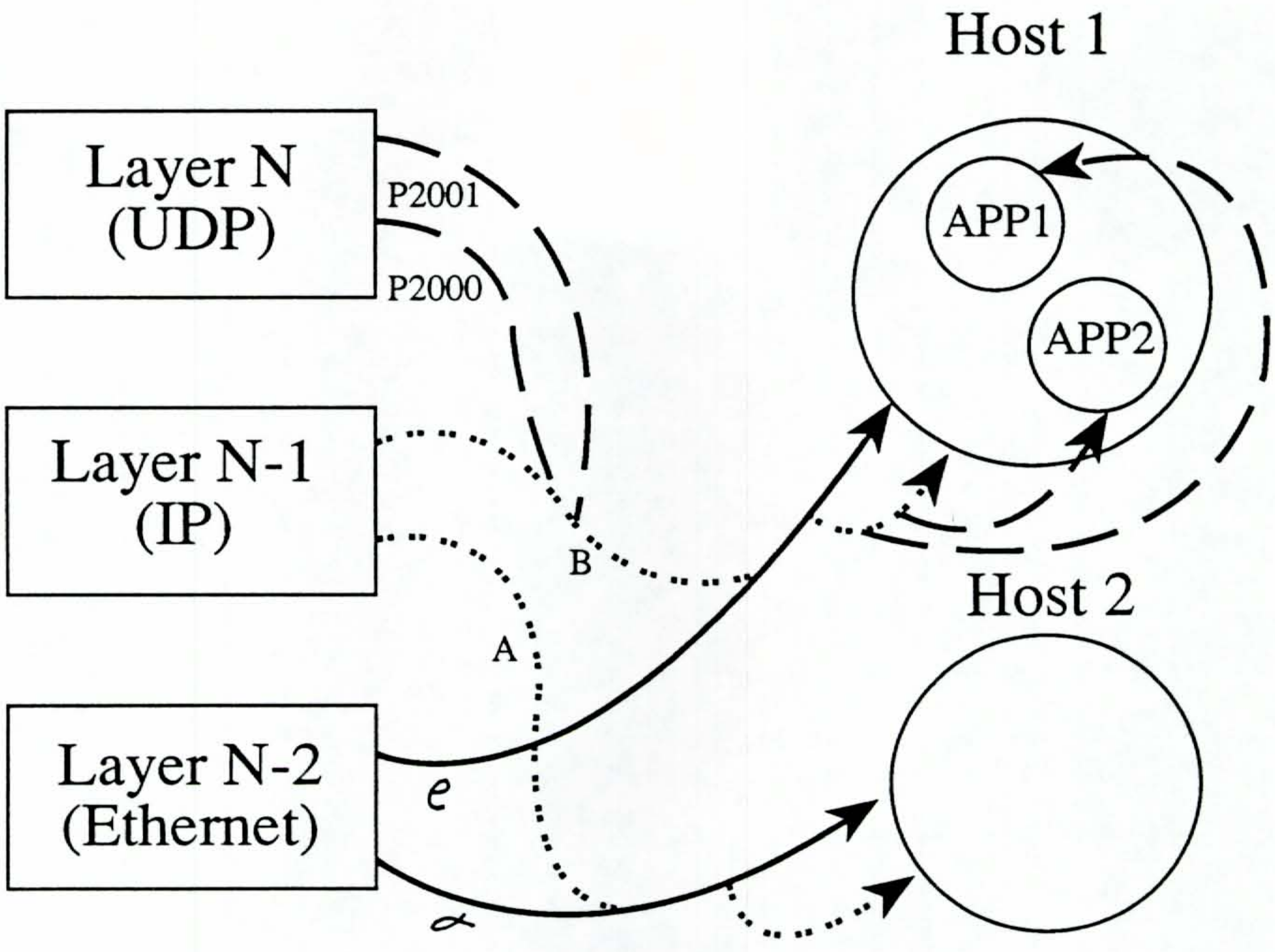
Using a 3 layered model, an application's data is encapsulated as follows:

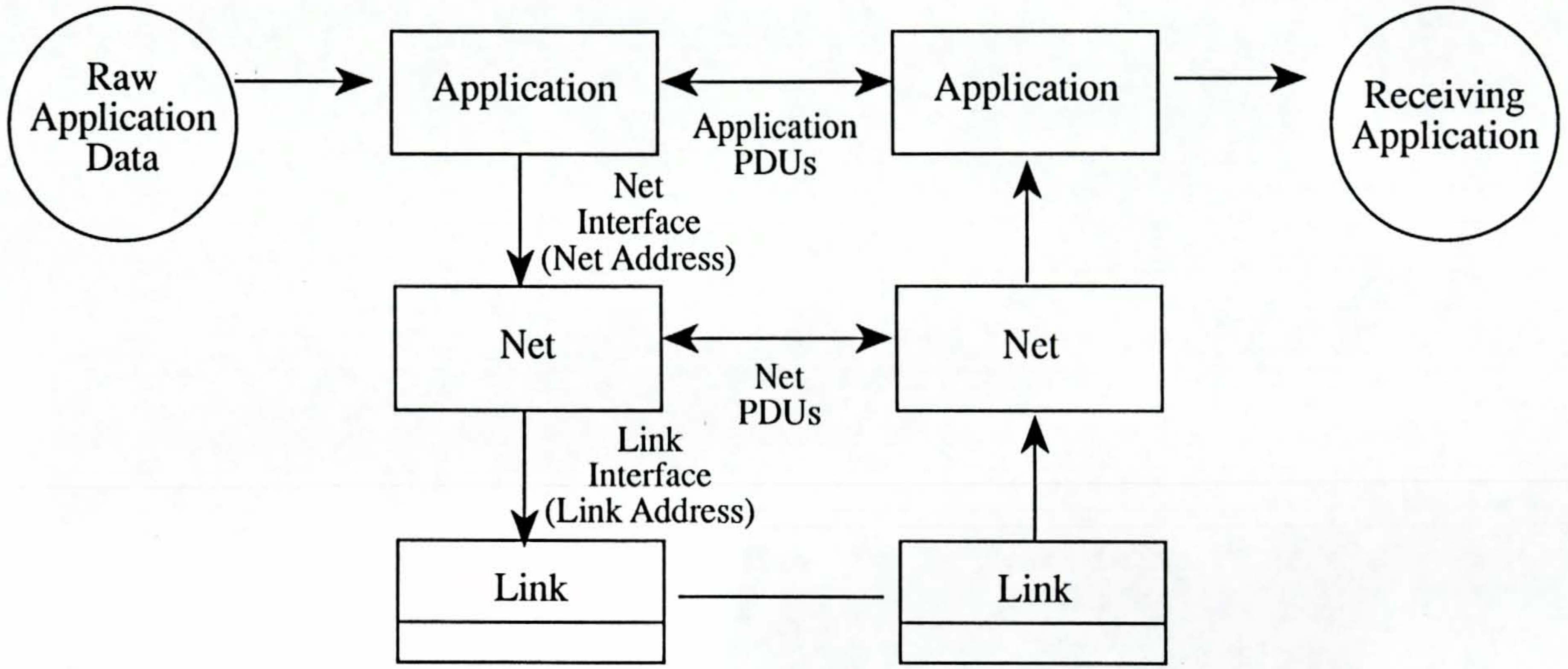


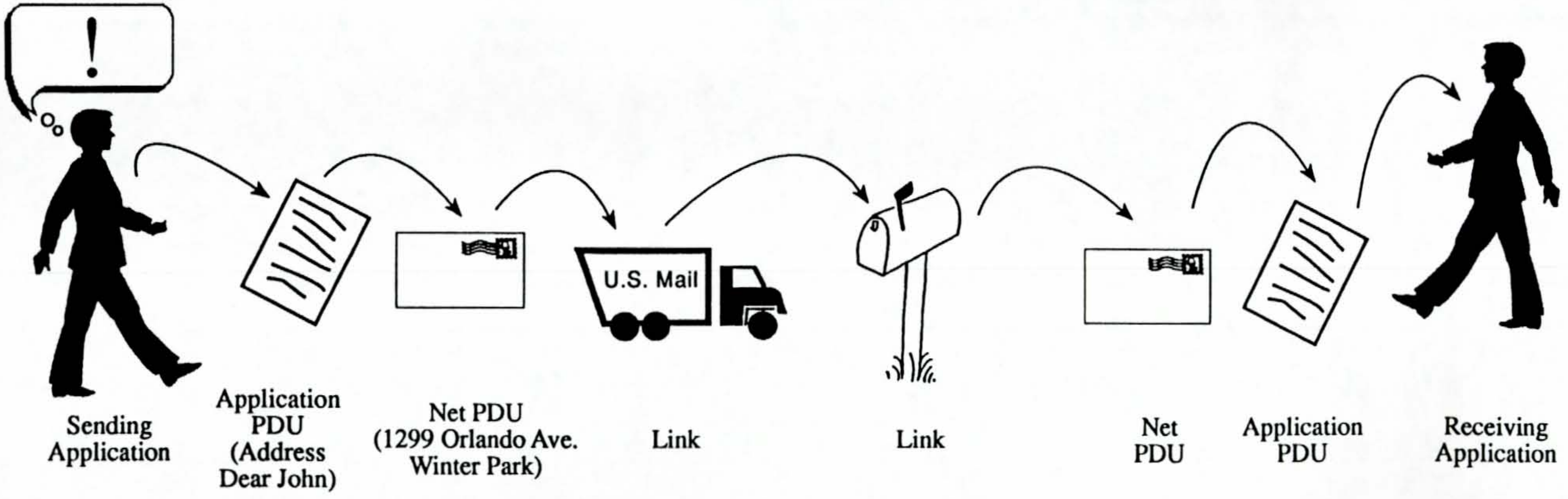
Some protocols have both headers and trailers (frames may have-end-of frame sequences, for example). In any case, another way to visualize the data encapsulation is:



Layer K needs to specify a layer K-1 address.
Applications track UDP Ports and IP addresses.
IP tracks link addresses.



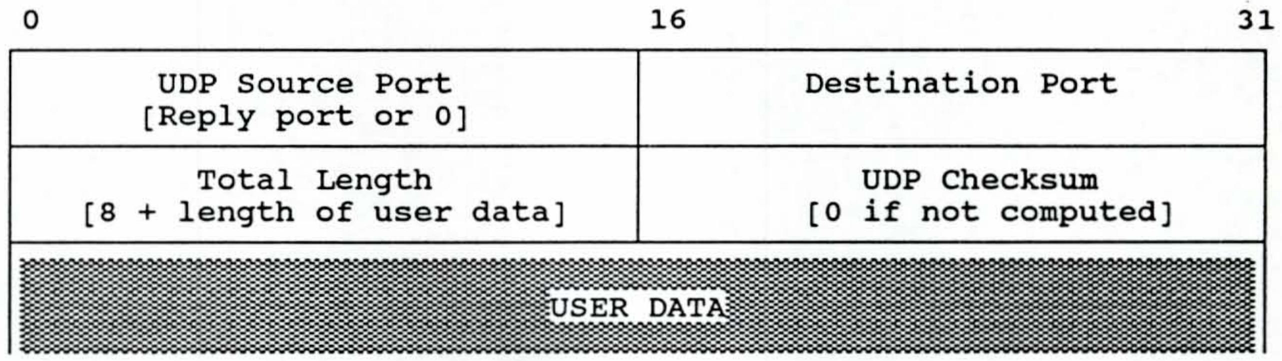

Raw Application Data
Application
Application PDUs
Application
Receiving Application
Net Interface (Net Address)
Net
Net PDUs
Net
Link Interface (Link Address)
Link
Link


!
U.S. Mail
Sending Application
Application PDU (Address Dear John)
Net PDU (1299 Orlando Ave. Winter Park)
Link
Link
Net PDU
Application PDU
Receiving Application

0416-2985

The UDP Header

As the last diagram shows, the application information is enclosed in a UDP envelope. The UDP envelope consists of a header only:



On a machine with 16 bit integers, the UDP header may rendered in C as:

```
struct udpHdr
{
    int sourcePort;        // Our port number
    int destPort;          // Target port number
    int length;            // Length, includes header and text
    int checksum;          // 0 implies none
};
```

Octet (byte) Ordering

Different machines use different octet (byte) ordering. On a 386 the address of an integer is the address of the octet containing the least significant bits. Other machines store integers such that the address of the integer is the address of the most significant bits. If machines do not agree on octet ordering, a 0XAACC will be interpreted as 0XCCAA after a direct data transfer. To solve this, each protocol (not the protocol stack) must specify an order for transmission. When the most significant octet is sent first it is termed "big endian" transmission, "little endian" transmissions send the least significant octet first. The TCP/IP suite of protocols (which includes ARP, RARP, UDP, ICMP, etc.) specifies "big endian" communications.

An applications using the IP suite is not required to change its PDUs to a big endian format. Application protocols, because they are protocols, can use whatever octet order is thought best. In particular, if the applications are restricted to a family of machines using little endian ordering it is probably best not to swap application PDU integers.

The UDP header consists of 4 2-octet integers. On machines (such as IBM PCs) which store integers in little endian form, the header received is not suitable for computation. For this reason, byte swapping must be done on such machines.

The standard C function "swab" is intended to swap bytes. It is more general than we need here so a function "swap" is assumed which will swap two adjacent bytes. For example:

```
void swap(void *ptr)
{
    // Swap bytes in place

    char *buf = (char *)ptr;
    char store;

    store    = *buf;
    *buf     = *(buf+1);
    *(buf+1)= store;
}
```

When writing code to be transportable between big and little endian machines, the simplest approach is to make this swap routine a no-op (simply return) on big endian architectures.

Given this, it is easy to write a function capable of adjusting UDP headers:

```
void fixUDP(udpHdr *udpp)
{
    swap(&udpp->sourcePort);
    swap(&udpp->destPort);
    swap(&udpp->length);
    swap(&udpp->checksum);
}
```

This routine will change back and forth (toggle) between big and little endian, as will similar routines introduced later.

An important question when developing on a little endian machine is whether headers, as they are built or deciphered, should be kept in a big or little endian format.

One approach is to keep the headers big endian but to also maintain "shadow" values in swapped form. This requires care in that the shadow (little endian forms) and the headers must be kept in synchronization. The approach taken here is to keep everything in the machine's native form. When a procedure is called to manipulate a data structure, the procedure is to assume any required swapping must be done within the procedure. With such an approach swapping is always a localized phenomenon, and human confusion is minimized. Making software understandable by human beings is of overriding importance. Eventually, double swaps might be removed, or conditionally compiled out.

A General Design for Nth Layer Protocol Transmission

When layer N+1 is ready to transmit, it needs to pass its data to the Nth layer through the layer N+1/N interface. The layer N support must envelop the incoming data and then pass it to layer N-1 for further processing. Only the highest layer (the application, which offers <u>service access points</u> to the non-protocol application) and the lowest layer (physical, or in our example, link) do not fully conform to this pattern.

```
procedure layer_N_send(data,lenData,targetAddr,protoNumber)
begin
    npdu = <allocate enough memory for layer-N header + data>

    <fill in the header of the npdu, includes protoNumber>
    <copy the user data after header in npdu>
    newAddr <- <map targetAddr to layer N-1 address>

    layer_N-1_send(npdu,lenData+<length of layer N header>,
                   newAddr,layer_N_ProtoNumber)

    <count the packet as sent if gathering statistics>
endproc
```

As will be shown shortly, data copying is not actually necessary.

Statistics gathering is encouraged. Some local counters and an interface to determine the statistics for each layer can yield some important information.

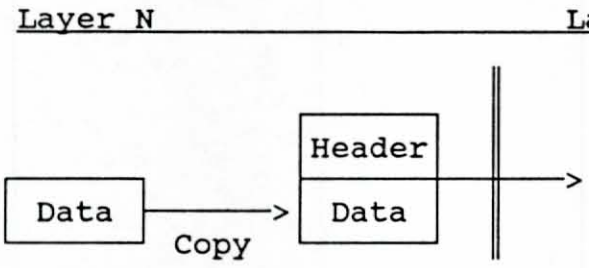
A General Design for Nth Layer Protocol Receipt

To develop the receiver for layer N you must be able to get the data intended for layer N. For UDP/IP on an Ethernet LAN, this means getting to the data after the LAN header.

The line interrupt handler simply accepts the data and queues it to the lowest layer of the network support. The Nth layer network support has the following form:

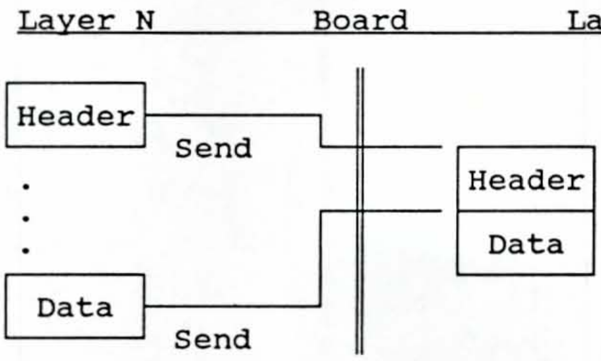
```
procNRecv(ourData, ourLen, ourProto)
begin
    if proto <> expectedProto OR <header is not valid> then
        <discard the packet>
        <count the packet as discarded>
    else
        { We have a valid packet }

        if <a local layer-N packet> then
            <process, possibly sending packets out>
            <discard the packet>
            <count packet as handled locally>
        else
            theirProtoNumber <- <extract layer N+1 proto #>

            procNPlus1Recv(<data without our header>,
                           ourLen - <length of our header>
                           theirProtoNumber)
        endif
    endif
endproc
```

In the send/receive scenarios shown, headers were added and deleted by copying data. For example, to send Data layer N copies data after its header:

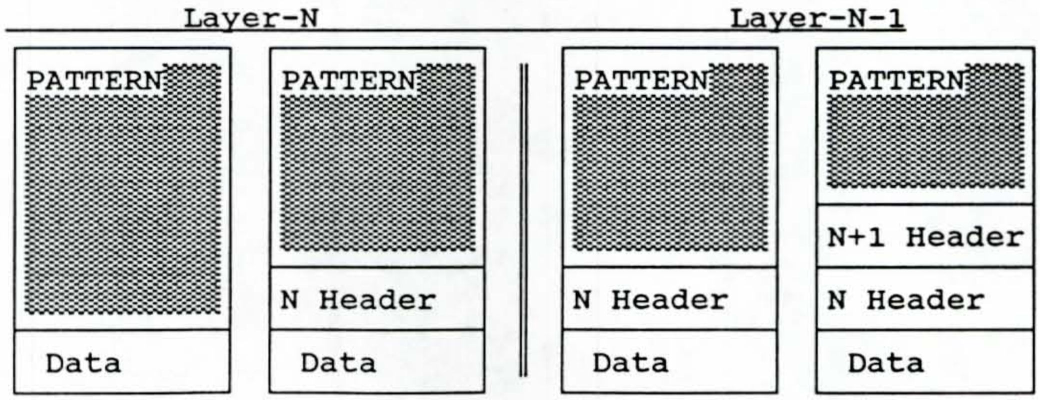


There are at least two other methods to manipulate headers:

- If layer N-1 is handled on a separate board from layer N, the layer-N header and layer-N data may be sent independently and read contiguously on the board.



- Over allocate the application PDUs and build the application PDU far enough down in the allocated block to allow enough room to build all the lower layer headers. Each layer assumes there is enough room in front of the data passed to build its header.

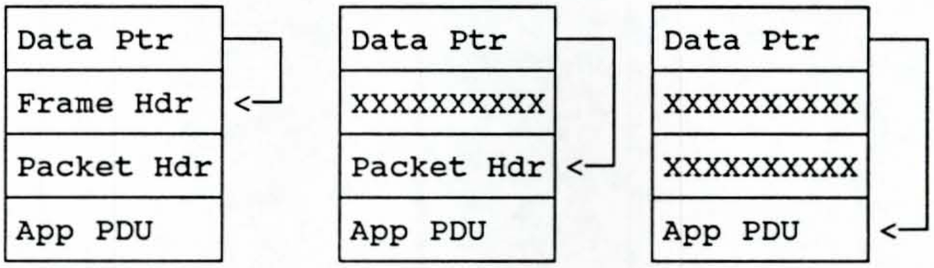


Although this method may have to be applied at a layer below the application layer (if there is a pre-existing application layer) it should be used at the first opportunity in order to avoid repeated copying.

It is suggested a pattern be built into the top of the generic header area and that the pattern be checked after all layers have added their headers. If the suggested sender design is used, the pattern should be inserted just before calling layer_N-1_send and validated on return.

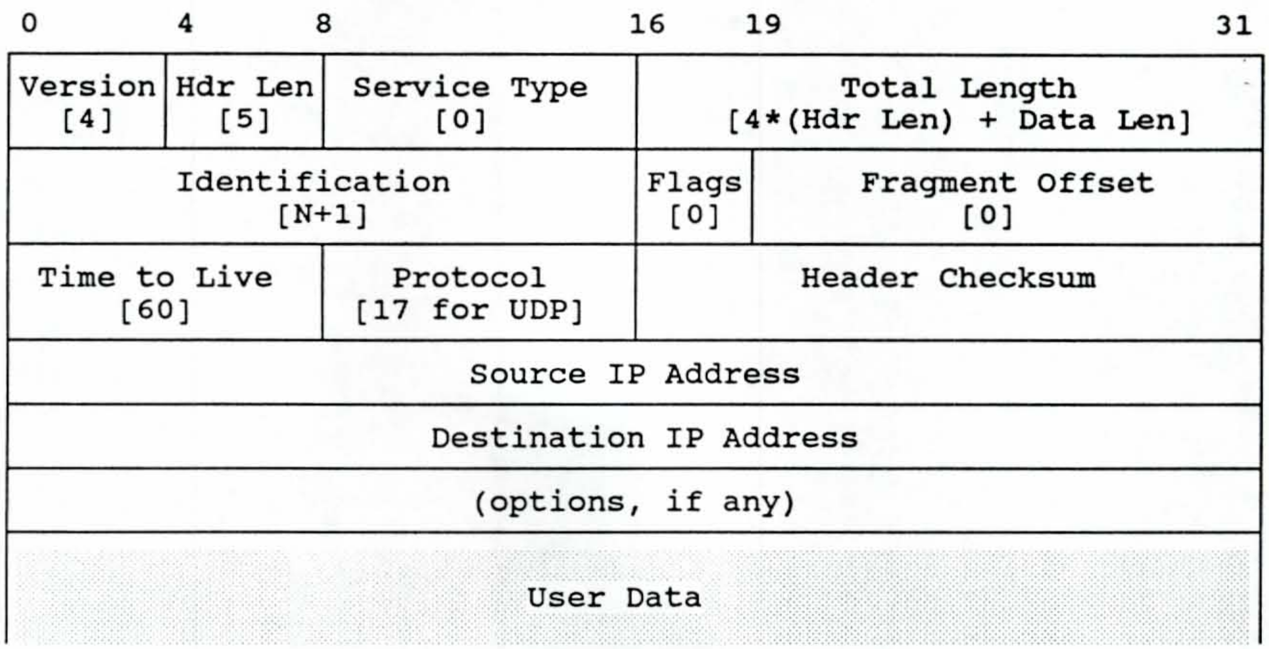
If multiple lower layer protocols are supported, the allocation should be large enough for the worst case to minimize layer binding.

To avoid copying on incoming data, use a data pointer associated with the incoming packet (possibly at the top of the packet). Each layer can "strip" its header by moving a pointer without copying data. At the top level the SAPs to the application will probably lead to the packet being discarded anyway and so the mechanism is hidden from the application.



IP Header Format

The IP header is much more complex than the UDP header. IP carries many protocols and must be suitable for transmission across many networks. The format is:



Options usually need not be of concern as most have to do with network control and are not processed by hosts.

One possible C representation is made with using 2 structures:

```
    struct ipAddr
    {
        uchar dot0;     // uchar is unsigned char
        uchar dot1;
        uchar dot2;
        uchar dot3;
    };
```

```
struct ipHdr
{
    // For the compiler used, bit fields must have low order
    // bits declared first.  This will vary among compilers.

    int ihl     : 4;      // Header Length in 32 bit words
    int version : 4;      // Version number

    char typeService;     // Precedence?, low delay?, high
                          // throughput?  0 for us.

    int totalLen;         // Length in octets, includes header
                          // and text
    int ident;           // Packet ID, 1 value for all fragments
    int fragOffset;       // 0 for unfragmented pkts, upper 3
                          // bits are fragment status indicators

    char timeToLive;     // Seconds or hops before packet death
    char protocol;        // What protocol are we carrying?
    int hdrChecksum;      // Header checksum

    ipAddr source;        // Source address
    ipAddr dest;          // Destination address;
};
```

- Service type: precedence (3 bits), low delay bit, high throughput bit, high reliability bit, 2 bits unused. Clear (0) is typical: yields normal precedence and no special handling.

- Data length can be determined from total length by subtracting four times the header length.

- Flags: tied to fragmentation, 0 for non-fragmented packets. Has a bit to indicate "do not fragment" should your target be unable to handle fragmented packets. One bit specifies "more fragments," needed since each fragment otherwise is complete (no grand total length) and packets may arrive out of order. Without this mechanism you would never know when the full packet was complete.

- Fragment offset: a multiple of 8 octets. For non-fragmented packets both the fragment offset and flags will be clear, resulting in a 16 bit zero.

- Protocol: in a sense, an extension of user data. Since no format alignments exist across the IP user community, this is necessary to allow users to recognize their own data.

- Although the IP header allows for 65,535 octets, lower layers usually are more restrictive (Ethernet limit is 1500). If this is a problem you may have to implement IP fragmentation, which is not covered sufficiently here.

Building an IP Header

There are four integers in the IP header, so on a little endian machine a swap routine is needed:

```
    void fixIP(ipHdr *ipp)
    {
        swap(&ipp->totalLen);
        swap(&ipp->ident);
        swap(&ipp->fragOffset);
        swap(&ipp->hdrChecksum);
    }


    #define currentIPVersion     4   // IP version number
    #define serviceType          0   // Type of service we use
    #define stayAlive           60   // Seconds (or hops) for
                                     // packet life max.

    #define protocolUDP         17   // Protocol selection UDP
    #define protocolTCP          6   // Protocol selection TCP
    #define protocolICMP         1   // Protocol selection ICMP



    void buildIP(ipHdr *ip,       // The packet being built
                 int dataLen,     // How much data is there?
                 ipAddr source,   // Who are we?
                 ipAddr dest)     // Who is the message's target?
    {
        ip->ihl              = sizeof(*ip)/4; // Size in 32 bits
        ip->fragOffset       = 0;             // We don't fragment

        ip->totalLen    = sizeof(ipHdr) + dataLen;
        ip->version     = currentIPVersion;
        ip->protocol    = protocolUDP;      // Should be parameter
        ip->typeService = serviceType;
        ip->source      = source;
        ip->dest        = dest;

        ip->timeToLive  = stayAlive;
        ip->ident       = nextID();

        ip->hdrChecksum = IPCheckSum(ip);
}
```

The IP Checksum

The IP checksum is computed by treating the header as a collection of 16 bit integers and adding them using one's complement arithmetic, and then taking the one's complement of the result. The algorithm can be implemented on a two's complement machine. The following routine will be used in computing both the IP header checksum and the UDP checksum:

```
int checkTotal(
    void *data,         // Data over which checksum is computed
    int len,            // Number of octets of data
    long initTotal)     // Begin total with this, 0 but for UDP
{
    char *block = (char *)data;      // Make it a byte pointer
    unsigned long sum;               // 32 bit total built here

    sum = initTotal;                 // Start with the
                                     // designated value

    while (len >= 2)                 // As long as there's
    {                                // at least two bytes...
        sum += *((uint *)block);     // Add in 2 octets
        len     -= 2;                // 2 bytes less to process
        block   += 2;                // Move to the next pair
    }

    if (len)
    {
        // The number of bytes was odd, add in the last byte

        int val = *((uchar *)block);
        sum += val;
    }

    sum  = (sum >> 16) + (sum & 0XFFFF);      // Add any 16 bit
                                              // overflow...
    sum += (sum >> 16);                       // Twice!
    return ~sum;                              // Bit flip
}
```

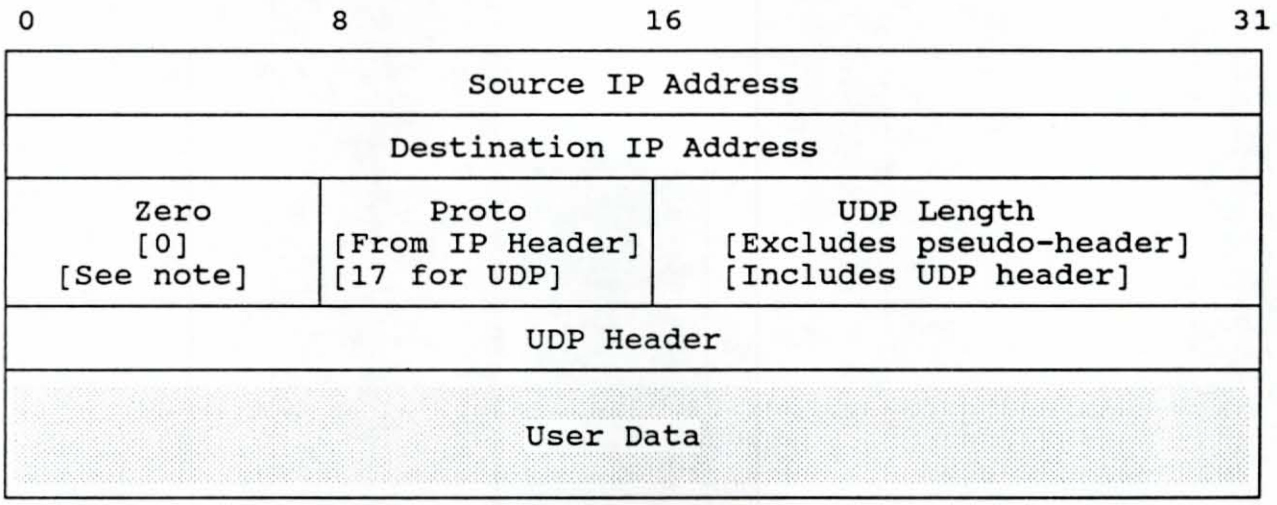
Using this, the IP Header checksum can be computed:

```
int IPChecksum(ipHdr *ipp)
{
    int holdCheck;      // Don't want to forget the sum
    int computed;       // Resulting checksum

    // Raw packet is big endian, we may maintain little endian
    fixIP(ipp);                           // Big endian, please

    holdCheck = ipp->hdrChecksum;         // Save the current value
    ipp->hdrChecksum = 0;                // Clear it for computation

    computed = checktotal(ipp,4*ipp->ihl,0);

    ipp->hdrChecksum = holdCheck;         // Restore the value

    fixIP(ipp);                           // Back to local
                                          // representation
    swap(&computed);                      // Computation yields
                                          // big endian

    return(computed);
}
```

The UDP Checksum

UDP is so closely tied to IP it is doubtful it can legitimately be called an independent protocol. This is shown dramatically be the way the UDP checksum is computed. Not only does the UDP checksum cover all the user data (which the IP checksum omits), it covers a mock IP header, called the "phantom header."

The quantity to which the UDP checksum applies is:



Note: the consecutive octets "Zero" and "Proto" should be viewed as a single 16 bit integer. This is important on little endian machines. The protocol number is kept as a single octet in the IP header, so it is shown as a single octet here, but it must be swapped with the zero on little endian machines.

When the checksum is computed, the checksum field in the UDP header must be zero (paralleling the IP computation). The format need not be built as a structure; it is simple enough to allow direct computation.

Here a partial checksum is computed for the pseudo-header, which is then combined with the checksum for the rest of the data.

```
long pseudoTotal(ipHdr *ip)
{
    // Compute a partial UDP checksum, just the total over
    // the pseudo-header

    long total;          // 32 bit running total
    uint *block;         // Total as 16 bit quantities
    uint dataLen = ip->totalLen - sizeof(ipHdr);
    uint protocol;

    // Protocol is an octet in the IP header, it's treated
    // as a 16 bit int in the pseudo header, so copy and swap
    protocol = ip->protocol;
    swap(&protocol);
    fixIP(ip);          // Big endian for checksum work
    swap(&dataLen);     // Including the length

    total = 0;

    block = (uint *)&ip->source;     // Add in source address
    total += *block++;
    total += *block;

    block = (uint *)&ip->dest;       // Add in dest address
    total += *block++;
    total += *block;

    total += protocol;               // Add in the protocol

    fixIP(ip);                       // Back to little endian

    return(total);
}
```

Using pseudoTotal and checkTotal the computation of the UDP checksum is a simple matter. Because UDP is based on both the UDP and IP headers, with one enclosed in the other, a new structure is used here:

```
struct udpIpHdr
{
    struct ipHdr;
    struct udpHdr;
};
```

```
int computeUDP(udpIpHdr *pkt)      // Compute the UDP checksum
{
    int dataLen;            // udp.length, save before swap
    ulong total;            // Running checksum total
    uint saveCk;            // Save/restore checksum
    int i;

    saveCk = pkt->udp.checksum;  // Must zero checksum field
    pkt->udp.checksum = 0;

    // Get the pseudo-header info total
    total = pseudoTotal(&pkt->ip);

    dataLen = pkt->udp.length;

    fixUDP(&pkt->udp);        // Prepare for computation

    // Compute the UDP checksum
    total = checkTotal(&pkt->udp,dataLen,total);

    fixUDP(&pkt->udp);        // Back to internal format

                              // Restore the structure
    pkt->udp.checksum = saveCk;

    i = total;                // Total was done big endian
    swap(&i);
    return (i);
}
```

Building a UDP Header

The UDP checksum is complex, but the header is otherwise simple. Building the UDP header is just a matter of filling in 4 fields:

```
    void buildUDP(
                udpIpHdr *pkt,    // Where's the packet being built?
                int dataLen,      // How much data is there?
                int source,       // Who are we?
                int dest,         // Where's it going?
                int check)        // Should we build a checksum?
{
    pkt->udp.sourcePort  = source;
    pkt->udp.destPort    = dest;
    pkt->udp.length      = dataLen + sizeof(*up);
    pkt->udp.checksum    = check ? computeUDP(pkt) : 0;
}
```

Address Resolution Protocol

It is often the case that an IP address is known, but the underlying hardware address is not. For communication with "well known" machines the lower layer addressing problem may be resolved by system configuration (human intervention). In a more general setting, you must be prepared to handle address resolution dynamically (changing a board in a machine may change its LAN address but would not change its IP address).

This problem is not unique to IP. Generally layer N needs the target address not in terms of layer N, but layer N-1. The technique is:

> Assume layer N needs to communicate with its peer with address AddrN. To send the message point to point the layer N-1 address is required (AddrN1). Layer N determines this by doing a layer N-1 broadcast asking its peer with address AddrN to supply its layer-N-1 address. When the reply is received, layer-N notes the address mapping to resolve future point to point requests.

At first blush it seems absurd to broadcast a message to get an address in order to avoid broadcast, but the resolution is done infrequently and so there should be a net saving.

To resolve addresses a new protocol is used, the Address Resolution Protocol (ARP). For the first IP message to go to a given IP address, address resolution, ideally, goes as follows:

1) the message is queued locally and an ARP is sent.
2) an ARP reply is received, the queued message is sent and the IP/HW mapping is saved in an address cache.

There are several complications:

- the message may become obsolete (when long delays are anticipated for ARP replies, it may be best to discard the message rather than queue it).

- more messages may arrive to be sent to the same address (queue them with the existing message, the existence of a pending message indicates an ARP is pending).

- no ARP reply is ever received (could use an ARP pending timer, on expiration retransmit or, in case the target doesn't exist, purge the pending queue).

If your address table becomes corrupt or goes out of date (a LAN board may be changed after the ARP is done) the network may become flooded with retransmissions and error reports.

The ARP Format

ARP is NOT a higher level protocol, it does NOT go in an IP envelope. The ARP format is:

0	8	16	31
Hardware Type code [1 for Ethernet]		Protocol Type [0X0800 for IP]	
HW Addr Length [6 for Ethernet]	Proto Addr Len [4 for IP]	Operation [1 for request, 2 for response]	
H octets for the sender's HW address P octets for the sender's protocol address H octets for the target HW address P bytes for the target protocol address			

As you might expect, the target HW address is 0 in an ARP request.

For an IP/Ethernet configuration on a 16 bit integer machine, a C structure representing an ARP packet could be written as:

```
    struct arpHdr           // The ARP header
    {
        int hardwareType;
        int protocolType;
        char hLen;
        char pLen;
        int operation;
        etherAddr etherSender;
        ipAddr ipSender;
        etherAddr etherTarget;
        ipAddr ipTarget;
    };
```

Some useful associated constants for an ARP implementation include:

```
    #define ARPrequest          1       // ARP operations
    #define ARPresponse         2
    #define RARPrequest         3       // Reverse ARP operations
    #define RARPresponse        4

    #define EtherHardware       1
```

ARP headers need swapping too:

```
    void fixArp(arpHdr *ap)
    {
        swap(&ap->hardwareType);
        swap(&ap->protocolType);
        swap(&ap->operation);
    }
```

ARP Receipt

The receiver of an ARP request:

1) may build IP/HW entry from the request
2) fills in the missing HW address
3) swaps the sender and target addresses
4) changes operation to reply

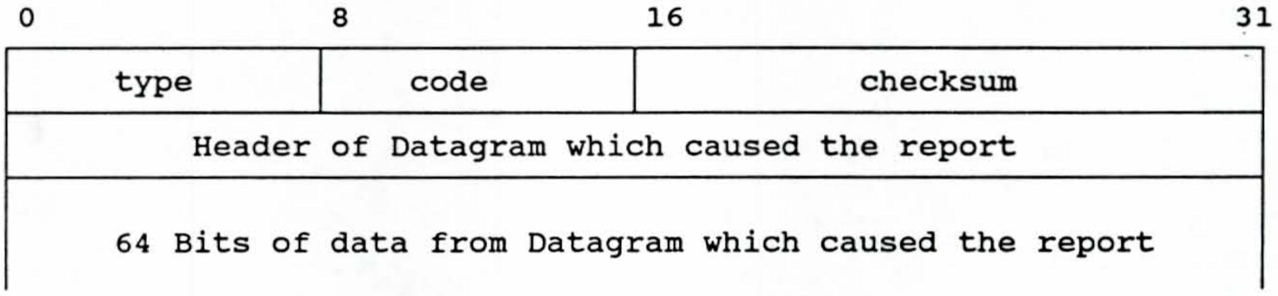
ARPs are broadcast, so everyone seeing an ARP request may build a table entry. Replies are point to point so not everyone sees the reply.

Even if your system resolves addresses through another means, ARP replies are so straight forward you may want to implement them as part of your system.

There exists a "reverse ARP" (RARP) for bootstrap (get an IP address based on a hardware address). RARP has the same format and similar procedures. The operations are 3 (RARP request) and 4 (RARP reply).

Internet Protocol Error and Control Messages

When operating on a legitimate IP net, you should be prepared to recognize Internet Protocol Error and Control Messages (ICMP). The ICMP format begins with a 4 byte header:


0
8
16
31
type
code
checksum
Header of Datagram which caused the report
64 Bits of data from Datagram which caused the report

The ICMP messages indicate problems with packets you have sent. For example, a type field of 3 indicates the router could not find a way to send your packet to the specified address. The code field meaning is dependent on the type field.

The ICMP structure can be represented as:

```
    struct icmpHdr
    {
        char icmpType;
        char icmpCode;
        int  icmpCheck;
    };
```

The active type values follow (some types are obsolete and so are not listed):

Type	Meaning	Notes
0	Echo reply	Seen only if you send an ICMP echo request
3	Destination unreachable	
4	Source Quench	
5	Redirect	Choose a different route for the next transmission to this host.
8	Echo request	PING, are you alive?
11	Time expired	Your time to live expired
12	Parameter problem	Packet header malformed
13	Timestamp request	
14	Timestamp reply	
17	Address Mask Request	
18	Address Mask Reply	

Miscellaneous Topics

- If you are working on a little endian machine and are using Ethernet, don't forget to swap the Ethernet "type."

```
typedef char etherAddr[6];

struct etherHdr             // The ethernet header
{
    etherAddr dest;
    etherAddr source;
    int etherType;
};

void fixEther(etherHdr *ep)
{
    swap(&ep->etherType);
}
```

- When you send IP messages on Ethernet, the type field should be 0X0800. ARP messages have Ethernet type 0X0806.

- Problems paralleling the byte swapping problem can appear in other forms. When defining structures (records) which have bit definitions (fields or packed booleans) your compiler may not assign bits in the order you need them to be. See the definition of the ipHdr for an example of such a problem (the first 2 declarations appear to be out of order).

- On receipt of a PDU all fields should be validated. If a field is not what is expected the PDU should be silently discarded. A basic tenet of protocol design requires this, otherwise corrupt packets may become Chernobylgrams (packets that bring down receivers).

Suggested References

Tanenbaum, Andrew S. [1989], Computer Networks, Second Edition, Prentice-Hall, Englewood Cliffs, New Jersey.

A splendid overview of computer networks. Although the text is not intended as a guide to implementation of any specific network, it gives a good overview of many. Considerable historical information is given. The OSI Reference Model is examined in depth, and is used as the basis for the text's presentation.

Comer, Douglas E. [1991], Internetworking with TCP/IP, Second Edition, Prentice Hall.

A well written text covering the implementation of TCP, UDP, IP, and the various support protocols (ARP, RARP, ICMP, etc.). Contains historical information and compares the TCP/IP suite to the ISO 7-layer Reference Model.

0000124