Network Working Group   David C. Plummer
Request For Comments: 826   (DCP@MIT-MC)
                                                           November 1982

An Ethernet Address Resolution Protocol
-- or --
Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware

Abstract

The implementation of protocol P on a sending host S decides, through protocol P's routing mechanism, that it wants to transmit to a target host T located some place on a connected piece of 10Mbit Ethernet cable. To actually transmit the Ethernet packet a 48.bit Ethernet address must be generated. The addresses of hosts within protocol P are not always compatible with the corresponding Ethernet address (being different lengths or values). Presented here is a protocol that allows dynamic distribution of the information needed to build tables to translate an address A in protocol P's address space into a 48.bit Ethernet address.

Generalizations have been made which allow the protocol to be used for non-10Mbit Ethernet hardware. Some packet radio networks are examples of such hardware.

The protocol proposed here is the result of a great deal of discussion with several other people, most notably J. Noel Chiappa, Yogen Dalal, and James E. Kulp, and helpful comments from David Moon.

[The purpose of this RFC is to present a method of Converting Protocol Addresses (e.g., IP addresses) to Local Network Addresses (e.g., Ethernet addresses). This is a issue of general concern in the ARPA Internet community at this time. The method proposed here is presented for your consideration and comment. This is not the specification of a Internet Standard.]

Notes:

This protocol was originally designed for the DEC/Intel/Xerox 10Mbit Ethernet. It has been generalized to allow it to be used for other types of networks. Much of the discussion will be directed toward the 10Mbit Ethernet. Generalizations, where applicable, will follow the Ethernet-specific discussion.

DOD Internet Protocol will be referred to as Internet.

Numbers here are in the Ethernet standard, which is high byte first. This is the opposite of the byte addressing of machines such as PDP-11s and VAXes. Therefore, special care must be taken with the opcode field (ar$op) described below.

An agreed upon authority is needed to manage hardware name space values (see below). Until an official authority exists, requests should be submitted to

David C. Plummer
Symbolics, Inc.
243 Vassar Street
Cambridge, Massachusetts 02139

Alternatively, network mail can be sent to DCP@MIT-MC.

The Problem:

The world is a jungle in general, and the networking game contributes many animals. At nearly every layer of a network architecture there are several potential protocols that could be used. For example, at a high level, there is TELNET and SUPDUP for remote login. Somewhere below that there is a reliable byte stream protocol, which might be CHAOS protocol, DOD TCP, Xerox BSP or DECnet. Even closer to the hardware is the logical transport layer, which might be CHAOS, DOD Internet, Xerox PUP, or DECnet. The 10Mbit Ethernet allows all of these protocols (and more) to coexist on a single cable by means of a type field in the Ethernet packet header. However, the 10Mbit Ethernet requires 48.bit addresses on the physical cable, yet most protocol addresses are not 48.bits long, nor do they necessarily have any relationship to the 48.bit Ethernet address of the hardware. For example, CHAOS addresses are 16.bits, DOD Internet addresses are 32.bits, and Xerox PUP addresses are 8.bits. A protocol is needed to dynamically distribute the correspondences between a <protocol, address> pair and a 48.bit Ethernet address.

Motivation:

Use of the 10Mbit Ethernet is increasing as more manufacturers supply interfaces that conform to the specification published by DEC, Intel and Xerox. With this increasing availability, more and more software is being written for these interfaces. There are two alternatives: (1) Every implementor invents his/her own method to do some form of address resolution, or (2) every implementor uses a standard so that his/her code can be distributed to other systems without need for modification. This proposal attempts to set the standard.

Definitions:

Define the following for referring to the values put in the TYPE field of the Ethernet packet header:

```
        ether_type$XEROX_PUP,
        ether_type$DOD_INTERNET,
        ether_type$CHAOS,
```

and a new one:

```
        ether_type$ADDRESS_RESOLUTION.
```

Also define the following values (to be discussed later):

```
        ares_op$REQUEST (= 1, high byte transmitted first) and
        ares_op$REPLY   (= 2),
```

and

```
        ares_hrd$Ethernet (= 1).
```

Packet format:

To communicate mappings from <protocol, address> pairs to 48.bit Ethernet addresses, a packet format that embodies the Address Resolution protocol is needed. The format of the packet follows.

```
    Ethernet transmission layer (not necessarily accessible to
         the user):
        48.bit: Ethernet address of destination
        48.bit: Ethernet address of sender
        16.bit: Protocol type = ether_type$ADDRESS_RESOLUTION
    Ethernet packet data:
        16.bit: (ar$hrd) Hardware address space (e.g., Ethernet,
                         Packet Radio Net.)
        16.bit: (ar$pro) Protocol address space.  For Ethernet
                         hardware, this is from the set of type
                         fields ether_typ$<protocol>.
         8.bit: (ar$hln) byte length of each hardware address
         8.bit: (ar$pln) byte length of each protocol address
        16.bit: (ar$op)  opcode (ares_op$REQUEST | ares_op$REPLY)
        nbytes: (ar$sha) Hardware address of sender of this
                         packet, n from the ar$hln field.
        mbytes: (ar$spa) Protocol address of sender of this
                         packet, m from the ar$pln field.
        nbytes: (ar$tha) Hardware address of target of this
                         packet (if known).
        mbytes: (ar$tpa) Protocol address of target.
```

Packet Generation:

As a packet is sent down through the network layers, routing determines the protocol address of the next hop for the packet and on which piece of hardware it expects to find the station with the immediate target protocol address. In the case of the 10Mbit Ethernet, address resolution is needed and some lower layer (probably the hardware driver) must consult the Address Resolution module (perhaps implemented in the Ethernet support module) to convert the <protocol type, target protocol address> pair to a 48.bit Ethernet address. The Address Resolution module tries to find this pair in a table. If it finds the pair, it gives the corresponding 48.bit Ethernet address back to the caller (hardware driver) which then transmits the packet. If it does not, it probably informs the caller that it is throwing the packet away (on the assumption the packet will be retransmitted by a higher network layer), and generates an Ethernet packet with a type field of ether_type$ADDRESS_RESOLUTION. The Address Resolution module then sets the ar$hrd field to ares_hrd$Ethernet, ar$pro to the protocol type that is being resolved, ar$hln to 6 (the number of bytes in a 48.bit Ethernet address), ar$pln to the length of an address in that protocol, ar$op to ares_op$REQUEST, ar$sha with the 48.bit ethernet address of itself, ar$spa with the protocol address of itself, and ar$tpa with the protocol address of the machine that is trying to be accessed. It does not set ar$tha to anything in particular, because it is this value that it is trying to determine. It could set ar$tha to the broadcast address for the hardware (all ones in the case of the 10Mbit Ethernet) if that makes it convenient for some aspect of the implementation. It then causes this packet to be broadcast to all stations on the Ethernet cable originally determined by the routing mechanism.

Packet Reception:

When an address resolution packet is received, the receiving Ethernet module gives the packet to the Address Resolution module which goes through an algorithm similar to the following. Negative conditionals indicate an end of processing and a discarding of the packet.

```
?Do I have the hardware type in ar$hrd?
Yes: (almost definitely)
  [optionally check the hardware length ar$hln]
  ?Do I speak the protocol in ar$pro?
  Yes:
    [optionally check the protocol length ar$pln]
    Merge_flag := false
    If the pair <protocol type, sender protocol address> is
        already in my translation table, update the sender
        hardware address field of the entry with the new
        information in the packet and set Merge_flag to true.
    ?Am I the target protocol address?
    Yes:
      If Merge_flag is false, add the triplet <protocol type,
          sender protocol address, sender hardware address> to
          the translation table.
      ?Is the opcode ares_op$REQUEST?  (NOW look at the opcode!!)
      Yes:
        Swap hardware and protocol fields, putting the local
            hardware and protocol addresses in the sender fields.
        Set the ar$op field to ares_op$REPLY
        Send the packet to the (new) target hardware address on
            the same hardware on which the request was received.
```

Notice that the <protocol type, sender protocol address, sender hardware address> triplet is merged into the table before the opcode is looked at. This is on the assumption that communcation is bidirectional; if A has some reason to talk to B, then B will probably have some reason to talk to A. Notice also that if an entry already exists for the <protocol type, sender protocol address> pair, then the new hardware address supersedes the old one. Related Issues gives some motivation for this.

Generalization: The ar$hrd and ar$hln fields allow this protocol and packet format to be used for non-10Mbit Ethernets. For the 10Mbit Ethernet <arhrd, arhln> takes on the value <1, 6>. For other hardware networks, the ar$pro field may no longer correspond to the Ethernet type field, but it should be associated with the protocol whose address resolution is being sought.

Why is it done this way??

Periodic broadcasting is definitely not desired. Imagine 100 workstations on a single Ethernet, each broadcasting address resolution information once per 10 minutes (as one possible set of parameters). This is one packet every 6 seconds. This is almost reasonable, but what use is it? The workstations aren't generally going to be talking to each other (and therefore have 100 useless entries in a table); they will be mainly talking to a mainframe, file server or bridge, but only to a small number of other workstations (for interactive conversations, for example). The protocol described in this paper distributes information as it is needed, and only once (probably) per boot of a machine.

This format does not allow for more than one resolution to be done in the same packet. This is for simplicity. If things were multiplexed the packet format would be considerably harder to digest, and much of the information could be gratuitous. Think of a bridge that talks four protocols telling a workstation all four protocol addresses, three of which the workstation will probably never use.

This format allows the packet buffer to be reused if a reply is generated; a reply has the same length as a request, and several of the fields are the same.

The value of the hardware field (ar$hrd) is taken from a list for this purpose. Currently the only defined value is for the 10Mbit Ethernet (ares_hrd$Ethernet = 1). There has been talk of using this protocol for Packet Radio Networks as well, and this will require another value as will other future hardware mediums that wish to use this protocol.

For the 10Mbit Ethernet, the value in the protocol field (ar$pro) is taken from the set ether_type$. This is a natural reuse of the assigned protocol types. Combining this with the opcode (ar$op) would effectively halve the number of protocols that can be resolved under this protocol and would make a monitor/debugger more complex (see Network Monitoring and Debugging below). It is hoped that we will never see 32768 protocols, but Murphy made some laws which don't allow us to make this assumption.

In theory, the length fields (ar$hln and ar$pln) are redundant, since the length of a protocol address should be determined by the hardware type (found in ar$hrd) and the protocol type (found in ar$pro). It is included for optional consistency checking, and for network monitoring and debugging (see below).

The opcode is to determine if this is a request (which may cause a reply) or a reply to a previous request. 16 bits for this is overkill, but a flag (field) is needed.

The sender hardware address and sender protocol address are absolutely necessary. It is these fields that get put in a translation table.

The target protocol address is necessary in the request form of the packet so that a machine can determine whether or not to enter the sender information in a table or to send a reply. It is not necessarily needed in the reply form if one assumes a reply is only provoked by a request. It is included for completeness, network monitoring, and to simplify the suggested processing algorithm described above (which does not look at the opcode until AFTER putting the sender information in a table).

The target hardware address is included for completeness and network monitoring. It has no meaning in the request form, since it is this number that the machine is requesting. Its meaning in the reply form is the address of the machine making the request. In some implementations (which do not get to look at the 14.byte ethernet header, for example) this may save some register shuffling or stack space by sending this field to the hardware driver as the hardware destination address of the packet.

There are no padding bytes between addresses. The packet data should be viewed as a byte stream in which only 3 byte pairs are defined to be words (arhrd, arpro and ar$op) which are sent most significant byte first (Ethernet/PDP-10 byte style).

Network monitoring and debugging:

The above Address Resolution protocol allows a machine to gain knowledge about the higher level protocol activity (e.g., CHAOS, Internet, PUP, DECnet) on an Ethernet cable. It can determine which Ethernet protocol type fields are in use (by value) and the protocol addresses within each protocol type. In fact, it is not necessary for the monitor to speak any of the higher level protocols involved. It goes something like this:

When a monitor receives an Address Resolution packet, it always enters the <protocol type, sender protocol address, sender hardware address> in a table. It can determine the length of the hardware and protocol address from the ar$hln and ar$pln fields of the packet. If the opcode is a REPLY the monitor can then throw the packet away. If the opcode is a REQUEST and the target protocol address matches the protocol address of the monitor, the monitor sends a REPLY as it normally would. The monitor will only get one mapping this way, since the REPLY to the REQUEST will be sent directly to the requesting host. The monitor could try sending its own REQUEST, but this could get two monitors into a REQUEST sending loop, and care must be taken.

Because the protocol and opcode are not combined into one field, the monitor does not need to know which request opcode is associated with which reply opcode for the same higher level protocol. The length fields should also give enough information to enable it to "parse" a protocol addresses, although it has no knowledge of what the protocol addresses mean.

A working implementation of the Address Resolution protocol can also be used to debug a non-working implementation. Presumably a hardware driver will successfully broadcast a packet with Ethernet type field of ether_type$ADDRESS_RESOLUTION. The format of the packet may not be totally correct, because initial implementations may have bugs, and table management may be slightly tricky. Because requests are broadcast a monitor will receive the packet and can display it for debugging if desired.

An Example:

Let there exist machines X and Y that are on the same 10Mbit Ethernet cable. They have Ethernet address EA(X) and EA(Y) and DOD Internet addresses IPA(X) and IPA(Y) . Let the Ethernet type of Internet be ET(IP). Machine X has just been started, and sooner or later wants to send an Internet packet to machine Y on the same cable. X knows that it wants to send to IPA(Y) and tells the hardware driver (here an Ethernet driver) IPA(Y). The driver consults the Address Resolution module to convert <ET(IP), IPA(Y)> into a 48.bit Ethernet address, but because X was just started, it does not have this information. It throws the Internet packet away and instead creates an ADDRESS RESOLUTION packet with

```
        (ar$hrd) = ares_hrd$Ethernet
        (ar$pro) = ET(IP)
        (ar$hln) = length(EA(X))
        (ar$pln) = length(IPA(X))
        (ar$op)  = ares_op$REQUEST
        (ar$sha) = EA(X)
        (ar$spa) = IPA(X)
        (ar$tha) = don't care
        (ar$tpa) = IPA(Y)
```

and broadcasts this packet to everybody on the cable.

Machine Y gets this packet, and determines that it understands the hardware type (Ethernet), that it speaks the indicated protocol (Internet) and that the packet is for it ((ar$tpa)=IPA(Y)). It enters (probably replacing any existing entry) the information that <ET(IP), IPA(X)> maps to EA(X). It then notices that it is a request, so it swaps fields, putting EA(Y) in the new sender Ethernet address field (ar$sha), sets the opcode to reply, and sends the packet directly (not broadcast) to EA(X). At this point Y knows how to send to X, but X still doesn't know how to send to Y.

Machine X gets the reply packet from Y, forms the map from <ET(IP), IPA(Y)> to EA(Y), notices the packet is a reply and throws it away. The next time X's Internet module tries to send a packet to Y on the Ethernet, the translation will succeed, and the packet will (hopefully) arrive. If Y's Internet module then wants to talk to X, this will also succeed since Y has remembered the information from X's request for Address Resolution.

Related issue:

It may be desirable to have table aging and/or timeouts. The implementation of these is outside the scope of this protocol. Here is a more detailed description (thanks to MOON@SCRC@MIT-MC).

If a host moves, any connections initiated by that host will work, assuming its own address resolution table is cleared when it moves. However, connections initiated to it by other hosts will have no particular reason to know to discard their old address. However, 48.bit Ethernet addresses are supposed to be unique and fixed for all time, so they shouldn't change. A host could "move" if a host name (and address in some other protocol) were reassigned to a different physical piece of hardware. Also, as we know from experience, there is always the danger of incorrect routing information accidentally getting transmitted through hardware or software error; it should not be allowed to persist forever. Perhaps failure to initiate a connection should inform the Address Resolution module to delete the information on the basis that the host is not reachable, possibly because it is down or the old translation is no longer valid. Or perhaps receiving of a packet from a host should reset a timeout in the address resolution entry used for transmitting packets to that host; if no packets are received from a host for a suitable length of time, the address resolution entry is forgotten. This may cause extra overhead to scan the table for each incoming packet. Perhaps a hash or index can make this faster.

The suggested algorithm for receiving address resolution packets tries to lessen the time it takes for recovery if a host does move. Recall that if the <protocol type, sender protocol address> is already in the translation table, then the sender hardware address supersedes the existing entry. Therefore, on a perfect Ethernet where a broadcast REQUEST reaches all stations on the cable, each station will be get the new hardware address.

Another alternative is to have a daemon perform the timeouts. After a suitable time, the daemon considers removing an entry. It first sends (with a small number of retransmissions if needed) an address resolution packet with opcode REQUEST directly to the Ethernet address in the table. If a REPLY is not seen in a short amount of time, the entry is deleted. The request is sent directly so as not to bother every station on the Ethernet. Just forgetting entries will likely cause useful information to be forgotten, which must be regained.

Since hosts don't transmit information about anyone other than themselves, rebooting a host will cause its address mapping table to be up to date. Bad information can't persist forever by being passed around from machine to machine; the only bad information that can exist is in a machine that doesn't know that some other machine has changed its 48.bit Ethernet address. Perhaps manually resetting (or clearing) the address mapping table will suffice.

This issue clearly needs more thought if it is believed to be important. It is caused by any address resolution-like protocol.