A Reverse Address Resolution Protocol

Ross Finlayson, Timothy Mann, Jeffrey Mogul, Marvin Theimer
                   Computer Science Department
                      Stanford University
                         June 1984

Status of this Memo

   This RFC suggests a method for workstations to dynamically find their
   protocol address (e.g., their Internet Address), when they know only
   their hardware address (e.g., their attached physical network
   address).

   This RFC specifies a proposed protocol for the ARPA Internet
   community, and requests discussion and suggestions for improvements.

I. Introduction

   Network hosts such as diskless workstations frequently do not know
   their protocol addresses when booted; they often know only their
   hardware interface addresses. To communicate using higher-level
   protocols like IP, they must discover their protocol address from
   some external source. Our problem is that there is no standard
   mechanism for doing so.

   Plummer's "Address Resolution Protocol" (ARP) [1] is designed to
   solve a complementary problem, resolving a host's hardware address
   given its protocol address. This RFC proposes a "Reverse Address
   Resolution Protocol" (RARP). As with ARP, we assume a broadcast
   medium, such as Ethernet.

II. Design Considerations

   The following considerations guided our design of the RARP protocol.

   A. ARP and RARP are different operations. ARP assumes that every
   host knows the mapping between its own hardware address and protocol
   address(es). Information gathered about other hosts is accumulated
   in a small cache. All hosts are equal in status; there is no
   distinction between clients and servers.

   On the other hand, RARP requires one or more server hosts to maintain
   a database of mappings from hardware address to protocol address and
   respond to requests from client hosts.

B. As mentioned, RARP requires that server hosts maintain large
databases. It is undesirable and in some cases impossible to maintain
such a database in the kernel of a host's operating system. Thus,
most implementations will require some form of interaction with a
program outside the kernel.

C. Ease of implementation and minimal impact on existing host
software are important. It would be a mistake to design a protocol
that required modifications to every host's software, whether or not
it intended to participate.

D. It is desirable to allow for the possibility of sharing code with
existing software, to minimize overhead and development costs.

III. The Proposed Protocol

We propose that RARP be specified as a separate protocol at the
data-link level. For example, if the medium used is Ethernet, then
RARP packets will have an Ethertype (still to be assigned) different
from that of ARP. This recognizes that ARP and RARP are two
fundamentally different operations, not supported equally by all
hosts. The impact on existing systems is minimized; existing ARP
servers will not be confused by RARP packets. It makes RARP a general
facility that can be used for mapping hardware addresses to any
higher level protocol address.

This approach provides the simplest implementation for RARP client
hosts, but also provides the most difficulties for RARP server hosts.
However, these difficulties should not be insurmountable, as is shown
in Appendix A, where we sketch two possible implementations for
4.2BSD Unix.

RARP uses the same packet format that is used by ARP, namely:

    ar$hrd (hardware address space) - 16 bits
    ar$pro (protocol address space) - 16 bits
    ar$hln (hardware address length) - 8 bits
    ar$pln (protocol address length) - 8 bits
    ar$op (opcode) - 16 bits
    ar$sha (source hardware address) - n bytes,
                                        where n is from the ar$hln field.
    ar$spa (source protocol address) - m bytes,
                                        where m is from the ar$pln field.
    ar$tha (target hardware address) - n bytes
    ar$tpa (target protocol address) - m bytes

arhrd, arpro, ar$hln and ar$pln are the same as in regular ARP
(see [1]).

Suppose, for example, that 'hardware' addresses are 48-bit Ethernet
addresses, and 'protocol' addresses are 32-bit Internet Addresses.
That is, we wish to determine Internet Addresses corresponding to
known Ethernet addresses. Then, in each RARP packet, ar$hrd = 1
(Ethernet), ar$pro = 2048 decimal (the Ethertype of IP packets),
ar$hln = 6, and ar$pln = 4.

There are two opcodes: 3 ('request reverse') and 4 ('reply reverse').
An opcode of 1 or 2 has the same meaning as in [1]; packets with such
opcodes may be passed on to regular ARP code. A packet with any
other opcode is undefined. As in ARP, there are no "not found" or
"error" packets, since many RARP servers are free to respond to a
request. The sender of a RARP request packet should timeout if it
does not receive a reply for this request within a reasonable amount
of time.

The arsha, arspa, artha, and ar$tpa fields of the RARP packet are
interpreted as follows:

When the opcode is 3 ('request reverse'):

    ar$sha is the hardware address of the sender of the packet.

    ar$spa is undefined.

    ar$tha is the 'target' hardware address.

        In the case where the sender wishes to determine his own
        protocol address, this, like ar$sha, will be the hardware
        address of the sender.

    ar$tpa is undefined.

When the opcode is 4 ('reply reverse'):

    ar$sha is the hardware address of the responder (the sender of the
    reply packet).

    ar$spa is the protocol address of the responder (see the note
    below).

    ar$tha is the hardware address of the target, and should be the
    same as that which was given in the request.

    ar$tpa is the protocol address of the target, that is, the desired
    address.

  Note that the requirement that ar$spa in opcode 4 packets be filled

   in with the responder's protocol is purely for convenience. For
   instance, if a system were to use both ARP and RARP, then the
   inclusion of the valid protocol-hardware address pair (ar$spa,
   ar$sha) may eliminate the need for a subsequent ARP request.

IV. References

   [1] Plummer, D., "An Ethernet Address Resolution Protocol", RFC 826,
   MIT-LCS, November 1982.

Appendix A. Two Example Implementations for 4.2BSD Unix

   The following implementation sketches outline two different
   approaches to implementing a RARP server under 4.2BSD.

   A. Provide access to data-link level packets outside the kernel. The
   RARP server is implemented completely outside the kernel and
   interacts with the kernel only to receive and send RARP packets. The
   kernel has to be modified to provide the appropriate access for these
   packets; currently the 4.2 kernel allows access only to IP packets.
   One existing mechanism that provides this capability is the CMU
   "packet-filter" pseudo driver. This has been used successfully at
   CMU and Stanford to implement similar sorts of "user-level" network
   servers.

   B. Maintain a cache of database entries inside the kernel. The full
   RARP server database is maintained outside the kernel by a user
   process. The RARP server itself is implemented directly in the
   kernel and employs a small cache of database entries for its
   responses. This cache could be the same as is used for forward ARP.

   The cache gets filled from the actual RARP database by means of two
   new ioctls. (These are like SIOCIFADDR, in that they are not really
   associated with a specific socket.) One means: "sleep until there is
   a translation to be done, then pass the request out to the user
   process"; the other means: "enter this translation into the kernel
   table". Thus, when the kernel can't find an entry in the cache, it
   puts the request on a (global) queue and then does a wakeup(). The
   implementation of the first ioctl is to sleep() and then pull the
   first item off of this queue and return it to the user process.
   Since the kernel can't wait around at interrupt level until the user
   process replies, it can either give up (and assume that the
   requesting host will retransmit the request packet after a second) or
   if the second ioctl passes a copy of the request back into the
   kernel, formulate and send a response at that time.