PE-ARP: Port Enhanced ARP for IPv4 Address Sharing

Manish Karir, Eric Wustrow, Jim Rees
Networking Research and Development
Merit Network Inc.
Ann Arbor, MI 48104 USA
{mkarir, ewust, rees}@merit.edu

***Abstract*— The Internet is rapidly nearing IPv4 address space exhaustion. Current projections predict that within the next two years, all IPv4 address blocks will have been assigned. Various methods such as IPv6 and IPv4 NAT have been proposed to address this scarcity. IPv6 introduces a new set of networking protocols that expands the available number of Internet addresses, while NAT relies on modification of packet headers by the network to implement IPv4 address sharing. In this paper we present a technique that allows multiple end hosts on a network to share a single IP address by relying on the use of a modified hardware address resolution protocol. Using the proposed approach it is possible to allow thousands of end hosts to share a single IPv4 address and at the same time be able to maintain an end-to-end consistent network. Our approach is fundamentally different from current techniques, as it does not require that packets from the end-host be modified at the network layer by an intermediate entity as they transit the network. In addition, each end-host can use a valid routable public IP address. We focus on an expanded use of existing networking protocols such as ARP and DNS to build a novel IPv4 address sharing technique. We have developed an initial implementation of our ideas via a modified Linux kernel which demonstrates the feasibility of our approach.**

I. Introduction

Current projections indicate that within the next two years, all available IPv4 address blocks will have been assigned [1]. The resulting shortage of IP addresses would severely limit innovation on the Internet and even its continued spread throughout the world. While IPv6 would expand the available number of Internet addresses, adoption has been slow, as it requires not only new hardware and software to be pervasively deployed, but also requires network operators to transition entire networks and user bases to this new set of networking protocols. Understandably, this has greatly slowed adoption. Network Address Translation (NAT) has emerged as an approach to share existing IPv4 address among multiple end-hosts by hiding several hosts behind a single public IPv4 address [2]. While to an outside observer, the hosts appear to have the same IP address, they in fact have distinct private IP addresses internally. This approach is fundamentally in conflict with the end-to-end approach to network design, which has been the cornerstone of the development of the Internet.

Our approach, called *Port Enhanced ARP* (PE-ARP), shares a single, publicly routable IP address among a group of hosts. Each host is assigned a unique, contiguous range of TCP and UDP port numbers. The combination of IP address and port range is used to uniquely identify each host. A modified version of the Address Resolution Protocol (ARP) [3] is used to direct incoming packets to the correct end host. No address translation is done and packets are not modified in transit.

The proposed technique has several advantages over existing methods for IPv4 address sharing or even IPv6:

- It does not require a massive global hardware or software upgrade.
- A single site can choose to use this technique and obtain its full benefits while at the same time continuing to be completely interoperable with the rest of the Internet.
- It does not require packet modifications that violate the end-to-end principle.
- It requires only simple software changes to existing hosts.

The rest of this paper is organized as follows: Section II describes the detailed architecture and concepts of PE-ARP; Section III provides details regarding the structure of the standard ARP protocol implementation in the Linux kernel, followed by our modifications; Section IV describes two deployment scenarios and our experiments with using PE-ARP in diferent network configurations; Section V provides a description of some related work; Section VI provides our conclusions and outlines future work.

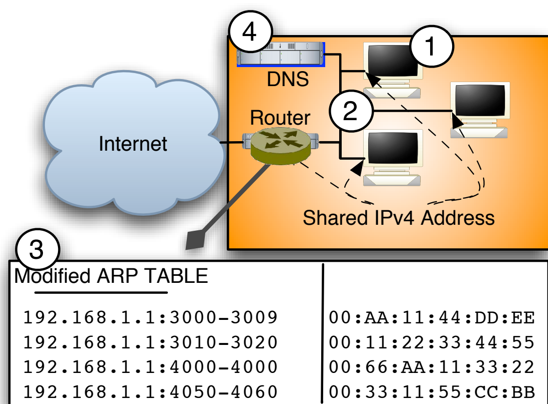

Fig. 1. End-to-End Consistent IPv4 Address Sharing

II. End-to-End Consistent IPv4 Address Sharing

The ideas outlined in this paper are based on three key insights. The first is that each end-host already has a unique hardware address (MAC address) and does not really need a unique IP address as an identifier. All that is necessary is that the IP header contains enough information to ensure that an incoming packet is able to reach the correct end-host. The second important observation is that large portions of the local port range are currently unused at an end host. Last of all, we note that the current Address Resolution Protocol (ARP) mechanism makes an implicit assumption that a single IP address should map to a single MAC address. This one-to-one mapping could be extended such that a single IP address can map to different MAC addresses based on other information available in the IPv4 header.

Based on these insights, we are able to define a new technique that allows a single IPv4 address to be shared across multiple end-hosts. We combine the port and the IP address to create a unique identifier that can be used by the ARP mechanism. PE-ARP can be implemented via a modified ARP table that resolves this combination of IP address and port number to a single unique MAC address, instead of simply resolving an IP address to a MAC address. However, care must now be taken to ensure that the local port ranges used by end-hosts sharing the same IP address do not overlap. This can be implemented via software modules that control which local ports are in use by an end-host and communicate this information to the local edge router.

Figure 1 describes the overall architecture of address sharing with PE-ARP. Labels 1-4 indicate modifications that are needed at a local site. In the subsections below we provide details about how each of these can be implemented.

A. End-Host Source Port Range Management Agent

The first component of the system is the end-host software module. Each application on a network end-host requests and obtains local port values from the operating system. These values can either be explicitly requested by the application or randomly assigned by the Operating System. To enable IPv4 address sharing, the range of local port values that is used by each end-host must be limited. This functionality can be implemented in a number of ways which will most likely be dependent on the specific Operating System.

The purpose of this local port range management agent is to intercept all local port requests from applications and to respond with values from fixed ranges. The ability to limit the range of values from which ports can be used is essential and allows us to reuse the remaining portions on another end-host.

The port range assigned to each end host could be generated in a number of ways. It could, for example, be statically configured at each host, it could be supplied via dhcp, or hosts could even randomly select port ranges and then perform conflict resolution to ensure non-overlapping port allocations.

B. Network Communication Protocol for IP Address and Port Information

The second major component of the system is the network communication protocol to share local port range information with the local network. The ultimate goal of this protocol is to provide every host on the network with enough information to send IPv4 packets to the correct physical host.

We have extended ARP to include both IP addresses and port ranges so it can be used to locate PE-ARP hosts. When the local router needs to know what host is using a particular port, the router sends out a broadcast request for the information (PE-ARP_REQUEST). The correct host responds (PE-ARP_REPLY), providing its hardware address and full local port range.

C. Mapping IPv4 packets to network end-hosts

The third component of the overall PE-ARP architecture is the extension to the local ARP table. When a packet arrives from the Internet to the local router, the router has to determine the packet's end-host destination. In current networks, the local router stores a mapping between IP addresses and physical MAC addresses. However, we now allow multiple end-hosts to share the same IPv4 address, so this information is no longer sufficient to uniquely identify the destination end-host for a packet. Instead, a modified table is used that employs both IP and port information to determine the MAC address of the end-host for which the packets are

intended. The structure of this table is shown in Figure 1.

In addition to the IP address, this table includes two port values, which are used to specify the range of ports that are associated with a given host.

D. Enhanced DNS Look-Up Service: Dissemination of IP and Port Information via DNS

The operation of services on well-known ports is a challenge in an environment where the single unique IP address per end-host restriction has been eliminated. These complications exist in NAT and other current IPv4 address sharing techniques and have not been adequately addressed. Providing services from behind a NAT requires that the NAT be configured to translate the public service port to the end-host's service port.

With PE-ARP no port translation is done, but each host can only provide services on those ports which it has been assigned. If the well-known port of the service is not in that host's range, the host cannot provide the service at that port and must use a different port within its range.

One possible solution is the use of DNS SRV records [4] for locating services on PE-ARP hosts. An SRV record maps a domain name and a service name to a canonical domain name and a port number. A PE-ARP service host can publish its service ports in this way. One problem with this approach is that not all client applications are capable of using SRV records in place of well-known ports.

III. Implementing PE-ARP

We have developed a prototype implementation of PE-ARP based on the Linux kernel version 2.6.29.3. In the following subsections we provide a high-level overview of the existing ARP implemenation in Linux, followed by our changes to this implementation. Next we describe two deployment scenarios for PE-ARP in a bridge and a router based edge network.

A. Background: ARP in the Linux Kernel

The ARP implementation in the Linux kernel relies on the core neighbour structure defined in `arp.h`. A simplified version of the neighbour struct is shown in Figure 2. ARP responses are cached as entries (*neighbours*) in a chaining hash table.

When a packet is passed down the networking stack for transmission, the packet needs to have the link-layer header filled in based on the destination IP. To do this, a lookup is performed on the ARP table. The lookup function hashes the IP address and device arguments and performs a lookup in the ARP hash table. If a valid entry matches, either the `ha[]` field or the `*hh` cache

```
struct neighbour
{
    struct neighbour        *next;
    struct net_device       *dev;
    u8                      nud_state;
    unsigned char           ha[MAX_ADDR_LEN];
    struct hh_cache         *hh;
    struct sk_buff_head     arp_queue;
    u8                      primary_key[0];
};

Where:
struct neighbour *next :pointer to the next neighbour in the hash chain
u8 nud_state           :flag that specifies the state of this entry.
unsigned char ha[]     :hardware address (aligned to a 4 byte boundry)
struct hh_cache *hh    :pointer to the hardware header cache. This
                        is a cache of the entire layer 2 header
                        (Source/Destination MAC and Ethertype)
struct sk_buff_head arp_queue :queue of skbs (packets)
                        that require this neighbour entry
u8 primary_key[0]      :the IP address. Notice that it is
                        of zero length; neighbours must be allocated
                        with sizeof(struct neighbour) += PROTOCOL_ADDR_LEN
```

Fig. 2. The Neighbour Data Structure in the Linux Kernel

of the entry will be used to fill in the link-layer header. If no entry is found in the hash table, one is created with `primary_key` filled in, and state set to incomplete. The packet is then pushed onto the `arp_queue` of the incomplete entry.

Every so often, a timer is triggered to check the state of the neighbour table entries. Incomplete entries prompt ARP requests to be sent to populate this entry. An ARP request is analogous to the English query 'Who (what MAC address) has 10.0.0.1? Tell me at 10.0.0.7 (MAC address 00:12:34:56:78:9A).'

TABLE I
EXISTING ARP STRUCTURE

Bit	0-7	8-15	16-31
0	Hardware Type		Protocol type
32	Hardware Len	Protocol Len	Operation
64	Sender HW Addr		
96	Sender HW Addr		Sender Protocol Addr
128	Sender Protocol Addr		Target HW Addr
160	Target HW Addr		
192	Target Protocol Addr		

Hardware Type: Link-layer protocol. Ethernet is 0x1
Protocol Type: Network-layer protocol. IPv4 is 0x0800
Hardware Len/Protocol Len - size in bytes of the link-layer and network-layer addresses
Operation: Type of ARP packet. ARP_REQUEST, or ARP_REPLY
Sender HW Addr: Link-layer or MAC address of the sender
Sender Protocol Addr: Network-layer or IP address of the sender
Target HW Addr: Link-layer address of the receiver ARP packet.
Target Protocol Addr: Network-layer address of the receiver. This is the IP address that is being looked up

Table I describes the structure of an ARP packet. When the Linux kernel receives an ARP packet, it first determines if the ARP packet is intended for itself, regardless of the operation (request or reply). It determines this by checking if the Target Protocol Address matches one of its own. If the receiver finds the packet

is sent to it, it then performs a lookup in the ARP hash table for the Sender Protocol Address (or creates one if one does not exist). It updates this entry to contain the Sender Hardware Address. If the ARP operation was a request, the receiver then sends a reply, with its Hardware address filled in. The ARP reply is similar to saying '10.0.0.1 is at 00:FE:DC:BA:98:76.' When a neighbour entry is updated by an ARP packet, the arp_queue in the neighbour entry is checked. Any queued packets are pulled off the queue, filled in with the correct link-layer destination and sent to the device for transmission.

B. PE-ARP in the Linux Kernel

Our current proof-of-concept implementation of PE-ARP consists of three of the four components described in the overall architecture in Section II. The first is a simple mechanism to control source port range allocation at an end host, the second is the actual modified ARP table, and the third is the port enabled ARP query/response packets and protocol. We anticipate implementing the fourth DNS integration component in our future work. We are able to fully demonstrate IP address sharing capability of PE-ARP with our current prototype.

While the PE-ARP architecture allows for a range of methods to implement source port range allocation to an end host, for the purposes of our prototype we use the simplest mechanism of manually allocating and configuring these values on our testbed. In Linux the range of available source ports can be set with the `net.ipv4.ip_local_port_range` sysctl.

The addition of port numbers to the ARP table can be implemented by modifying the neighbour struct described in Figure 2. We added an additional member to this data structure to store a port range as shown below.

```
unsigned short    port_range[2];
```

`port_range[0]` is the lowest port number in the range allocated to an end host, and `port_range[1]` is the upper limit (both inclusive). These values are both 0 if the entry applies for all ports. Care must be taken in modifying the neighbour data structure to ensure that the *primary_key* field is always the last component.

In addition to the modified ARP table we also need to modify the ARP query/response packet structures to include port information. Since we wanted to allow for backwards compatibility in our prototype, all of the information was appended to the end of the existing ARP packet format, and a magic number was used to denote the use of PE-ARP. Table II shows the additional fields required by PE-ARP which are simply appended to the regular ARP packet format described in Table I. Both PE-ARP_REQUEST and PE-ARP_REPLY use the same packet format.

TABLE II
NEW ARP STRUCTURE

Bit	0-15	16-31
224	Magic Value (0xEAAB)	Sender Port Range Low
256	Sender Port Range High	Target Port
288	Return Port	

Magic Value: The special designator for PE-ARP packets
Sender Port Range High/Low: The bounds of the sender port range
Target Port: The port we are looking up. This cannot be a range because the sender cannot know what range an arbitrary port falls in. For an ARP_REPLY, this just needs to be a port in the requestor's range, so we choose the Sender Port Range Low from the corresponding request
Return Port: For an ARP_REPLY, this is the Target Port requested in the corresponding ARP_REQUEST. For an ARP_REQUEST, Return Port is set to Sender Port Range Low

```
pe-arp-hostA:~$ arp -n
Address                    HWtype  HWaddress            Iface
192.35.162.2:2000-2999     ether   00:0c:29:9e:8c:ee    eth1
192.35.162.2:3000-3999     ether   00:0c:29:39:44:ab    eth1
198.108.63.1:0-0           ether   00:12:7f:c4:38:d3    eth0
```

Fig. 3. Output of arp -n command showing the PE-ARP table

With the above data structure and packet format changes in place, the next step is to modify the standard ARP processing code in the Linux kernel to make it port aware at each step. At an end host, when we are trying to send packets out, we must first check to see if the destination IP:port range is not in our local ARP cache. As per standard ARP processing, we create an incomplete entry for that IP address in the ARP table, however we also set the port_range (low and high) to be equal to the destination port of the packet we are sending. An PE-ARP request is then sent, which appears as follows: *'Who has 10.0.0.1:22? I have 10.0.0.1:100-199 (MAC 00:12:34:56:78:9A).'*

On receiving the PE-ARP request a node checks to see if the Target Protocol Address matches its own, and if the Target Port is within its configured range. If both are a match, the node will first update its own PE-ARP table by performing a neighbour entry lookup for the Sender Protocol Address and Return Port and updating this entry with the Sender Hardware Address, and Sender port range, or create it if it does not exist and then finally the node will send the PE-ARP response back to the requestor as follows: *'10.0.0.1:22 is really 10.0.0.1:10-99 (MAC 00:FE:DC:BA:98:76).'*

One final set of changes that are required are to the routing code to ensure that the end host does not assume that a packet to an IP address of one of its interfaces is local - it must also check the destination port number against is configured port range. Figure 3 shows an example ARP table. There are multiple entries for the same IP address which are differentiated by port range.

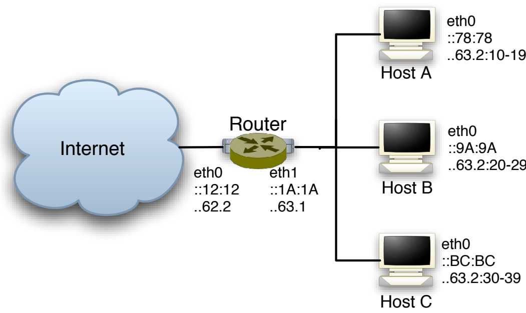

Fig. 4. PE-ARP Router Scenario: Host A, B, and C all share the same IP (198.108.63.2). Each host is limited to a port range. Interfaces are shown with shortened MAC address and IP address/port range (if applicable)

Each entry points to a unique MAC address.

IV. Deployment Scenarios and Experiments

During our development of the PE-ARP prototype we considered two primary deployment scenarios. We have been successfully able to create a shared IP address environment in both these scenarios by using our implementation of PE-ARP. The first is a PE-ARP router and the second is a bridge.

A. *PE-ARP Enabled Router*

In order to forward packets to the correct end host the local last-hop router must also be made port-aware. However, it should be noted that we do not require any changes to the packet forwarding or routing functions of the router. The only changes that are necessary are to the MAC address lookup capability. This PE-ARP capabilty is identical to the implementation on other end hosts described in the previous section. In fact our Linux implementation of this scenario did not require us to make any additional modifications to the Linux based router we used for this purpose. The PE-ARP modifications were sufficient for the router to be able to forward packets to the correct end-host.

The gateway in this configuration has two physical interfaces Each interface has an IP address on different networks that this gateway is routing between. The PE-ARP aware hosts can share one or more IP addresses on the local network (198.108.63.0/24). It should be noted that the router interface IP address on the local subnet itself should not be a shared IP as this IP address will be used as the next-hop for all outgoing traffic. Figure 4 shows this basic setup.

There are three packet forwarding scenarios to consider. Outgoing traffic from the end hosts to the Internet, inbound traffic from the Internet to the hosts, and finally communication between the hosts in the local subnet. In the case of outbound traffic, each end host only allows applications to use ports in its configured port range and then forwards the packets to the gateway. The gateway simply forwards these to the Internet without any additional PE-ARP related processing. Inbound traffic processing is more involved. In this scenario, the gateway performs a PE-ARP lookup to determine which destination MAC address it should forward an incoming packet to. Any communication between hosts in the local network must also use PE-ARP lookups to identify the correct targets before sending packets.

B. *PE-ARP Bridge*

Though our tests used a Linux based router, in general, the PE-ARP enabled router architecture described in the previous section would require that various hardware vendors develop implementations suitable for their platforms before a given site can benefit from the IP address sharing capabilities of PE-ARP. Therefore as an alternative we have also implemented a Linux-based Bridge to allow easier adoption of PE-ARP. Figure 5 shows this deployment scenario.

In a typical bridge, there are two or more physical interfaces. Each interface has a unique MAC address, but no IP is assigned to either. In Linux, the bridged interfaces belong to a virtual interface, which may have an IP address (and whose MAC address is one of the physical interfaces). In normal bridge operation, any packets that come in on one interfaces get re-sent on the other. From a Layer 2 perspective, the gateway in Figure 5 sends packets to the PE-Bridge using standard ARP. However, the PE-Bridge must perform a PE-ARP lookup on the local network to determine which host it should forward the packets to.

The PE-Bridge first checks the destination IP of an incoming packet against the shared IP address. This requires the bridge to know what the shared IP is. This can be implemented via a simple configuration file. For our test scenario, we accomplished this by requiring the bridged interface (`br0`) to have the shared IP assigned to it. Once the PE-Bridge determines the packet is to the shared IP, it performs a lookup based on the destination IP and port. If an entry is found in the PE-ARP table, it is used to re-write the destination MAC address, and the packet continues through the bridge. If no entry is found, the bridge creates a dummy entry in the arp table for `destIP:dport-dport`, and pushes the packet onto the `arp_queue` of that entry. A PE-ARP request then occurs on the local network, and the response is used to populate the entry, and send the waiting packet out. For traffic that does not use ports (non TCP nor UDP), the packet is simply sent through the bridge unaltered.

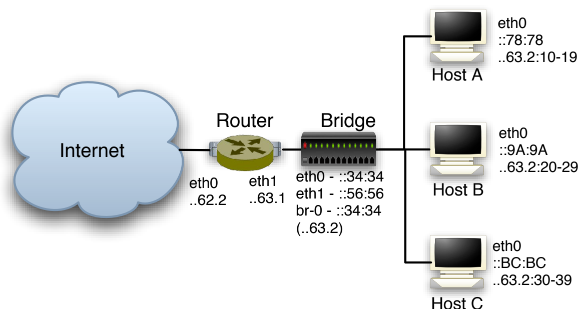

Fig. 5. PE-ARP Bridge Scenario: Host A, B, C and the PE-Bridge all share the same IP (198.108.63.2). Each host is limited to a port range. Interfaces are shown with shortened MAC address and IP address/port range (if applicable)

V. Related Work

There are two broad categories in which related work in the area of IPv4 address space exhaustion can be categorized. The first is the set of techniques that propose to replace IPv4 with IPv6. We believe that though this might be the ultimate long term solution to the IPv4 address depletion problem, current adoption rates indicate that it is unlikely that wide spread IPv6 deployments will be in place prior to IPv4 address space exhaustion. The second set of techniques in this area relate to the use of NAT, proxies and Realm Specific Internet Protocol (RSIP) [5]. All of these essentially attempt to isolate the end host into a private network, which uses non-routable IP addresses for communication with the outside world. An in-line device is then responsible for modifying packet headers and replacing private IP addresses with addresses chosen from a public IP pool before these are transmitted over the Internet [6].

Currently, the primary approach for IP address sharing is Network Address Translation (NAT). NAT is the most widely used technique to allow several network end-hosts share a limited set of IPv4 addresses. This approach relies on devices that modify IP addresses and port numbers in IP packets, as they are transmitted from the network end-host to the Internet. Such transparent modifications of a packet between the source and the destination make it extremely difficult to implement an end-to-end security model. NAT also breaks any protocol that depends on embedded addresses or port numbers, for example FTP, peer-to-peer, or file system protocols that depend on callbacks [7].

The A+P approach [8] of using port numbers to extend the effective IPv4 address is similar in nature to our approach in that they both reclaim unused source port space as a part of the end-host identifier. This was developed largely to address issues and complications of the Carrier Grade NAT technique [9]. However, the A+P scheme continues to rely on the use of a A+P NAT middle device to implement NAT-like capability. Our approach is fundamentally different in that we allow end hosts to use valid public IP addresses, but only a limited range of source ports. This enables them to communicate directly with other services on the Internet without the need for any network-layer packet translations in the middle.

VI. Conclusions and Future Work

We have PE-ARP installed on several test systems on our network. We hope to gain more experience with PE-ARP in a variety of situations including both desktop and server use. We also plan to characterize the scalability of the system and have begun measurements to determine how large a pool of ports is required by a typical host.

Our prototype uses manual configuration of the port ranges. We intend to implement an automatic system for allocating and configuring the ranges, possibly using DHCP [10].

We want to survey popular client applications to see which, if any, are using DNS SRV records now, and modify others to use these records. A new application interface similar to `gethostbyname()` and `getservbyname()` but that takes both a domain and a service name, and returns an IP address and port number, would make this easier. We also would like to implement a way for PE-ARP hosts to register service ports as SRV records using Dynamic DNS [11].

We hope to investigate the use of PE-ARP as part of an IPv6 migration strategy. It should be possible to embed a description of the port ranges into an IPv6 address, which would give us the ability to directly map between PE-ARP host identifiers and IPv6 addresses.

Our prototype runs on Linux, but there is nothing OS-specific about it. We would like to implement prototypes on popular consumer operating systems to investigate portability and scaling on these platforms.

To encourage wider use and investigation of the PE-ARP system we intend to write an Internet Draft to be published within the IETF framework.

References

[1] Geoff Huston. Ipv4 address report. *http://www.potaroo.net/tools/ipv4/index.html*, July 2009.

[2] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), May 1994. Obsoleted by RFC 3022.

[3] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Standard), November 1982. Updated by RFCs 5227, 5494.

[4] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (Proposed Standard), February 2000.

[5] M. Borella, D. Grabelsky, J. Lo, and K. Taniguchi. Realm Specific IP: Protocol Specification. RFC 3103 (Experimental), October 2001.

[6] L. Phifer. The trouble with NAT. *The Internet Protocol Journal*, Dec 2000.
[7] M. Holdrege and P. Srisuresh. Protocol Complications with the IP Network Address Translator. RFC 3027 (Informational), January 2001.
[8] O. Maennel, R. Bush, L. Cittadi, and S.M. Bellovin. A Better Approach than Carrier-Grade-NAT. *Technical Report CUCS-041-80*, Sep 2008.
[9] A. Durand. Managing 100+ Million IP Addreses. *NANOG37: http://nanog.org/mtg-0606/durand.html*, June 2006.
[10] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFCs 3396, 4361, 5494.
[11] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136 (Proposed Standard), April 1997. Updated by RFCs 3007, 4035, 4033, 4034.