BEHAVE WG   M. Bagnulo
Internet-Draft   UC3M
Intended status: Standards Track   A. Sullivan
Expires: April 4, 2011   Shinkuro
                                                            P. Matthews
                                                         Alcatel-Lucent
                                                        I. van Beijnum
                                                         IMDEA Networks
                                                        October 1, 2010

DNS64: DNS extensions for Network Address Translation from IPv6 Clients
                          to IPv4 Servers
                    draft-ietf-behave-dns64-11

Abstract

   DNS64 is a mechanism for synthesizing AAAA records from A records.
   DNS64 is used with an IPv6/IPv4 translator to enable client-server
   communication between an IPv6-only client and an IPv4-only server,
   without requiring any changes to either the IPv6 or the IPv4 node,
   for the class of applications that work through NATs. This document
   specifies DNS64, and provides suggestions on how it should be
   deployed in conjunction with IPv6/IPv4 translators.

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 4, 2011.

Table of Contents

1. Introduction

   This document specifies DNS64, a mechanism that is part of the
   toolbox for IPv6-IPv4 transition and co-existence. DNS64, used
   together with an IPv6/IPv4 translator such as stateful NAT64
   [I-D.ietf-behave-v6v4-xlate-stateful], allows an IPv6-only client to
   initiate communications by name to an IPv4-only server.

   DNS64 is a mechanism for synthesizing AAAA resource records (RRs)
   from A RRs. A synthetic AAAA RR created by the DNS64 from an
   original A RR contains the same owner name of the original A RR but
   it contains an IPv6 address instead of an IPv4 address. The IPv6
   address is an IPv6 representation of the IPv4 address contained in
   the original A RR. The IPv6 representation of the IPv4 address is
   algorithmically generated from the IPv4 address returned in the A RR
   and a set of parameters configured in the DNS64 (typically, an IPv6
   prefix used by IPv6 representations of IPv4 addresses and optionally
   other parameters).

   Together with an IPv6/IPv4 translator, these two mechanisms allow an
   IPv6-only client to initiate communications to an IPv4-only server
   using the FQDN of the server.

   These mechanisms are expected to play a critical role in the IPv4-
   IPv6 transition and co-existence. Due to IPv4 address depletion, it
   is likely that in the future, many IPv6-only clients will want to
   connect to IPv4-only servers. In the typical case, the approach only
   requires the deployment of IPv6/IPv4 translators that connect an
   IPv6-only network to an IPv4-only network, along with the deployment
   of one or more DNS64-enabled name servers. However, some features
   require performing the DNS64 function directly in the end-hosts
   themselves.

   This document is structured as follows: section 2 provides a non-
   normative overview of the behaviour of DNS64. Section 3 provides a
   non-normative background required to understand the interaction
   between DNS64 and DNSSEC. The normative specification of DNS64 is
   provided in sections 4, 5 and 6. Section 4 defines the terminology,
   section 5 is the actual DNS64 specification and section 6 covers
   deployments issues. Section 7 is non-normative and provides a set of
   examples and typical deployment scenarios.

2. Overview

   This section provides an introduction to the DNS64 mechanism.

   We assume that we have one or more IPv6/IPv4 translator boxes

connecting an IPv4 network and an IPv6 network. The IPv6/IPv4
translator device provides translation services between the two
networks enabling communication between IPv4-only hosts and IPv6-only
hosts. (NOTE: By IPv6-only hosts we mean hosts running IPv6-only
applications, hosts that can only use IPv6, as well as cases where
only IPv6 connectivity is available to the client. By IPv4-only
servers we mean servers running IPv4-only applications, servers that
can only use IPv4, as well as cases where only IPv4 connectivity is
available to the server). Each IPv6/IPv4 translator used in
conjunction with DNS64 must allow communications initiated from the
IPv6-only host to the IPv4-only host.

To allow an IPv6 initiator to do a standard AAAA RR DNS lookup to
learn the address of the responder, DNS64 is used to synthesize a
AAAA record from an A record containing a real IPv4 address of the
responder, whenever the DNS64 cannot retrieve a AAAA record for the
queried name. The DNS64 service appears as a regular DNS server or
resolver to the IPv6 initiator. The DNS64 receives a AAAA DNS query
generated by the IPv6 initiator. It first attempts a resolution for
the requested AAAA records. If there are no AAAA records available
for the target node (which is the normal case when the target node is
an IPv4-only node), DNS64 performs a query for A records. For each A
record discovered, DNS64 creates a synthetic AAAA RR from the
information retrieved in the A RR.

The owner name of a synthetic AAAA RR is the same as that of the
original A RR, but an IPv6 representation of the IPv4 address
contained in the original A RR is included in the AAAA RR. The IPv6
representation of the IPv4 address is algorithmically generated from
the IPv4 address and additional parameters configured in the DNS64.
Among those parameters configured in the DNS64, there is at least one
IPv6 prefix. If not explicitly mentioned, all prefixes are treated
equally and the operations described in this document are performed
using the prefixes available. So as to be general, we will call any
of these prefixes Pref64::/n, and describe the operations made with
the generic prefix Pref64::/n. The IPv6 address representing IPv4
addresses included in the AAAA RR synthesized by the DNS64 contain
Pref64::/n and they also embed the original IPv4 address.

The same algorithm and the same Pref64::/n prefix(es) must be
configured both in the DNS64 device and the IPv6/IPv4 translator(s),
so that both can algorithmically generate the same IPv6
representation for a given IPv4 address. In addition, it is required
that IPv6 packets addressed to an IPv6 destination address that
contains the Pref64::/n be delivered to an IPv6/IPv4 translator that
has that particular Pref64::/n configured, so they can be translated
into IPv4 packets.

Once the DNS64 has synthesized the AAAA RRs, the synthetic AAAA RRs
are passed back to the IPv6 initiator, which will initiate an IPv6
communication with the IPv6 address associated with the IPv4
receiver. The packet will be routed to an IPv6/IPv4 translator which
will forward it to the IPv4 network.

In general, the only shared state between the DNS64 and the IPv6/IPv4
translator is the Pref64::/n and an optional set of static
parameters. The Pref64::/n and the set of static parameters must be
configured to be the same on both; there is no communication between
the DNS64 device and IPv6/IPv4 translator functions. The mechanism
to be used for configuring the parameters of the DNS64 is beyond the
scope of this memo.

The prefixes to be used as Pref64::/n and their applicability are
discussed in [I-D.ietf-behave-address-format]. There are two types
of prefixes that can be used as Pref64::/n.

   The Pref64::/n can be the Well-Known Prefix 64:FF9B::/96 reserved
   by [I-D.ietf-behave-address-format] for the purpose of
   representing IPv4 addresses in IPv6 address space.

   The Pref64::/n can be a Network-Specific Prefix (NSP). An NSP is
   an IPv6 prefix assigned by an organization to create IPv6
   representations of IPv4 addresses.

The main difference in the nature of the two types of prefixes is
that the NSP is a locally assigned prefix that is under control of
the organization that is providing the translation services, while
the Well-Known Prefix is a prefix that has a global meaning since it
has been assigned for the specific purpose of representing IPv4
addresses in IPv6 address space.

The DNS64 function can be performed in any of three places. The
terms below are more formally defined in Section 4.

The first option is to locate the DNS64 function in authoritative
servers for a zone. In this case, the authoritative server provides
synthetic AAAA RRs for an IPv4-only host in its zone. This is one
type of DNS64 server.

Another option is to locate the DNS64 function in recursive name
servers serving end hosts. In this case, when an IPv6-only host
queries the name server for AAAA RRs for an IPv4-only host, the name
server can perform the synthesis of AAAA RRs and pass them back to
the IPv6-only initiator. The main advantage of this mode is that
current IPv6 nodes can use this mechanism without requiring any
modification. This mode is called "DNS64 in DNS recursive resolver

mode". This is a second type of DNS64 server, and it is also one
type of DNS64 resolver.

The last option is to place the DNS64 function in the end hosts,
coupled to the local (stub) resolver. In this case, the stub
resolver will try to obtain (real) AAAA RRs and in case they are not
available, the DNS64 function will synthesize AAAA RRs for internal
usage. This mode is compatible with some functions like DNSSEC
validation in the end host. The main drawback of this mode is its
deployability, since it requires changes in the end hosts. This mode
is called "DNS64 in stub-resolver mode". This is the second type of
DNS64 resolver.

3. Background to DNS64-DNSSEC interaction

DNSSEC ([RFC4033], [RFC4034], [RFC4035]) presents a special challenge
for DNS64, because DNSSEC is designed to detect changes to DNS
answers, and DNS64 may alter answers coming from an authoritative
server.

A recursive resolver can be security-aware or security-oblivious.
Moreover, a security-aware recursive resolver can be validating or
non-validating, according to operator policy. In the cases below,
the recursive resolver is also performing DNS64, and has a local
policy to validate. We call this general case vDNS64, but in all the
cases below the DNS64 functionality should be assumed needed.

DNSSEC includes some signaling bits that offer some indicators of
what the query originator understands.

If a query arrives at a vDNS64 device with the "DNSSEC OK" (DO) bit
set, the query originator is signaling that it understands DNSSEC.
The DO bit does not indicate that the query originator will validate
the response. It only means that the query originator can understand
responses containing DNSSEC data. Conversely, if the DO bit is
clear, that is evidence that the querying agent is not aware of
DNSSEC.

If a query arrives at a vDNS64 device with the "Checking Disabled"
(CD) bit set, it is an indication that the querying agent wants all
the validation data so it can do checking itself. By local policy,
vDNS64 could still validate, but it must return all data to the
querying agent anyway.

Here are the possible cases:

1. A DNS64 (DNSSEC-aware or DNSSEC-oblivious) receives a query with
    the DO bit clear. In this case, DNSSEC is not a concern, because
    the querying agent does not understand DNSSEC responses. The
    DNS64 can do validation of the response, if dictated by its local
    policy.

2. A security-oblivious DNS64 receives a query with the DO bit set,
    and the CD bit clear or set. This is just like the case of a
    non-DNS64 case: the server doesn't support it, so the querying
    agent is out of luck.

3. A security-aware and non-validating DNS64 receives a query with
    the DO bit set and the CD bit clear. Such a resolver is not
    validating responses, likely due to local policy (see [RFC4035],
    section 4.2). For that reason, this case amounts to the same as
    the previous case, and no validation happens.

4. A security-aware and non-validating DNS64 receives a query with
    the DO bit set and the CD bit set. In this case, the DNS64 is
    supposed to pass on all the data it gets to the query initiator
    (see section 3.2.2 of [RFC4035]). This case will not work with
    DNS64, unless the validating resolver is prepared to do DNS64
    itself. If the DNS64 modifies the record, the client will get
    the data back and try to validate it, and the data will be
    invalid as far as the client is concerned.

5. A security-aware and validating DNS64 resolver receives a query
    with the DO bit clear and CD clear. In this case, the resolver
    validates the data. If it fails, it returns RCODE 2 (Server
    failure); otherwise, it returns the answer. This is the ideal
    case for vDNS64. The resolver validates the data, and then
    synthesizes the new record and passes that to the client. The
    client, which is presumably not validating (else it should have
    set DO and CD), cannot tell that DNS64 is involved.

6. A security-aware and validating DNS64 resolver receives a query
    with the DO bit set and CD clear. This works like the previous
    case, except that the resolver should also set the "Authentic
    Data" (AD) bit on the response.

7. A security-aware and validating DNS64 resolver receives a query
    with the DO bit set and CD set. This is effectively the same as
    the case where a security-aware and non-validating recursive
    resolver receives a similar query, and the same thing will
    happen: the downstream validator will mark the data as invalid if
    DNS64 has performed synthesis. The node needs to do DNS64
    itself, or else communication will fail.

4. Terminology

   This section provides definitions for the special terms used in the
   document.

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [RFC2119].

   Authoritative server: A DNS server that can answer authoritatively a
      given DNS request.

   DNS64: A logical function that synthesizes DNS resource records (e.g
      AAAA records containing IPv6 addresses) from DNS resource records
      actually contained in the DNS (e.g., A records containing IPv4
      addresses).

   DNS64 recursive resolver: A recursive resolver that provides the
      DNS64 functionality as part of its operation. This is the same
      thing as "DNS64 in recursive resolver mode".

   DNS64 resolver: Any resolver (stub resolver or recursive resolver)
      that provides the DNS64 function.

   DNS64 server: Any server providing the DNS64 function. This
      includes the server portion of a recursive resolver when it is
      providing the DNS64 function.

   IPv4-only server: Servers running IPv4-only applications, servers
      that can only use IPv4, as well as cases where only IPv4
      connectivity is available to the server.

   IPv6-only hosts: Hosts running IPv6-only applications, hosts that
      can only use IPv6, as well as cases where only IPv6 connectivity
      is available to the client.

   Recursive resolver: A DNS server that accepts requests from one
      resolver, and asks another server (of some description) for the
      answer on behalf of the first resolver. Full discussion of DNS
      recursion is beyond the scope of this document; see [RFC1034] and
      [RFC1035] for full details.

   Synthetic RR: A DNS resource record (RR) that is not contained in
      the authoritative servers' zone data, but which is instead
      synthesized from other RRs in the same zone. An example is a
      synthetic AAAA record created from an A record.

   IPv6/IPv4 translator: A device that translates IPv6 packets to IPv4
      packets and vice-versa. It is only required that the
      communication initiated from the IPv6 side be supported.

   For a detailed understanding of this document, the reader should also
   be familiar with DNS terminology from [RFC1034], [RFC1035] and
   current NAT terminology from [RFC4787]. Some parts of this document
   assume familiarity with the terminology of the DNS security
   extensions outlined in [RFC4035]. It is worth emphasizing that while
   DNS64 is a logical function separate from the DNS, it is nevertheless
   closely associated with that protocol. It depends on the DNS
   protocol, and some behavior of DNS64 will interact with regular DNS
   responses.

5. DNS64 Normative Specification

   DNS64 is a logical function that synthesizes AAAA records from A
   records. The DNS64 function may be implemented in a stub resolver,
   in a recursive resolver, or in an authoritative name server. It
   works within those DNS functions, and appears on the network as
   though it were a "plain" DNS resolver or name server conforming to
   [RFC1034], and [RFC1035].

   The implementation SHOULD support mapping of separate IPv4 address
   ranges to separate IPv6 prefixes for AAAA record synthesis. This
   allows handling of special use IPv4 addresses [RFC5735].

   DNS messages contain several sections. The portion of a DNS message
   that is altered by DNS64 is the Answer section, which is discussed
   below in section Section 5.1. The resulting synthetic answer is put
   together with other sections, and that creates the message that is
   actually returned as the response to the DNS query. Assembling that
   response is covered below in section Section 5.4.

   DNS64 also responds to PTR queries involving addresses containing any
   of the IPv6 prefixes it uses for synthesis of AAAA RRs.

5.1. Resolving AAAA queries and the answer section

   When the DNS64 receives a query for RRs of type AAAA and class IN, it
   first attempts to retrieve non-synthetic RRs of this type and class,
   either by performing a query or, in the case of an authoritative
   server, by examining its own results. The query may be answered from
   a local cache, if one is available. DNS64 operation for classes
   other than IN is undefined, and a DNS64 MUST behave as though no
   DNS64 function is configured.

5.1.1. The answer when there is AAAA data available

   If the query results in one or more AAAA records in the answer
   section, the result is returned to the requesting client as per
   normal DNS semantics, except in the case where any of the AAAA
   records match a special exclusion set of prefixes, considered in
   Section 5.1.4. If there is (non-excluded) AAAA data available, DNS64
   SHOULD NOT include synthetic AAAA RRs in the response (see Appendix A
   for an analysis of the motivations for and the implications of not
   complying with this recommendation). By default DNS64
   implementations MUST NOT synthesize AAAA RRs when real AAAA RRs
   exist.

5.1.2. The answer when there is an error

   If the query results in a response with RCODE other than 0 (No error
   condition), then there are two possibilities. A result with RCODE=3
   (Name Error) is handled according to normal DNS operation (which is
   normally to return the error to the client). This stage is still
   prior to any synthesis having happened, so a response to be returned
   to the client does not need any special assembly than would usually
   happen in DNS operation.

   Any other RCODE is treated as though the RCODE were 0 (see sections
   Section 5.1.6 and Section 5.1.7) and the answer section were empty.
   This is because of the large number of different responses from
   deployed name servers when they receive AAAA queries without a AAAA
   record being available (see [RFC4074]). Note that this means, for
   practical purposes, that several different classes of error in the
   DNS are all treated as though a AAAA record is not available for that
   owner name.

   It is important to note that, as of this writing, some servers
   respond with RCODE=3 to a AAAA query even if there is an A record
   available for that owner name. Those servers are in clear violation
   of the meaning of RCODE 3, and it is expected that they will decline
   in use as IPv6 deployment increases.

5.1.3. Dealing with timeouts

   If the query receives no answer before the timeout (which might be
   the timeout from every authoritative server, depending on whether the
   DNS64 is in recursive resolver mode), it is treated as RCODE=2
   (Server failure).

5.1.4. Special exclusion set for AAAA records

   Some IPv6 addresses are not actually usable by IPv6-only hosts. If
   they are returned to IPv6-only querying agents as AAAA records,
   therefore, the goal of decreasing the number of failure modes will
   not be attained. Examples include AAAA records with addresses in the
   ::ffff:0:0/96 network, and possibly (depending on the context) AAAA
   records with the site's Pref::64/n or the Well-Known Prefix (see
   below for more about the Well-Known Prefix). A DNS64 implementation
   SHOULD provide a mechanism to specify IPv6 prefix ranges to be
   treated as though the AAAA containing them were an empty answer. An
   implementation SHOULD include the ::ffff/96 network in that range by
   default. Failure to provide this facility will mean that clients
   querying the DNS64 function may not be able to communicate with hosts
   that would be reachable from a dual-stack host.

   When the DNS64 performs its initial AAAA query, if it receives an
   answer with only AAAA records containing addresses in the excluded
   range(s), then it MUST treat the answer as though it were an empty
   answer, and proceed accordingly. If it receives an answer with at
   least one AAAA record containing an address outside any of the
   excluded range(s), then it MAY build an answer section for a response
   including only the AAAA record(s) that do not contain any of the
   addresses inside the excluded ranges. That answer section is used in
   the assembly of a response as detailed in Section 5.4.
   Alternatively, it MAY treat the answer as though it were an empty
   answer, and proceed accordingly. It MUST NOT return the offending
   AAAA records as part of a response.

5.1.5. Dealing with CNAME and DNAME

   If the response contains a CNAME or a DNAME, then the CNAME or DNAME
   chain is followed until the first terminating A or AAAA record is
   reached. This may require the DNS64 to ask for an A record, in case
   the response to the original AAAA query is a CNAME or DNAME without a
   AAAA record to follow. The resulting AAAA or A record is treated
   like any other AAAA or A case, as appropriate.

   When assembling the answer section, any chains of CNAME or DNAME RRs
   are included as part of the answer along with the synthetic AAAA (if
   appropriate).

5.1.6. Data for the answer when performing synthesis

   If the query results in no error but an empty answer section in the
   response, the DNS64 attempts to retrieve A records for the name in
   question, either by performing another query or, in the case of an
   authoritative server, by examining its own results. If this new A RR

query results in an empty answer or in an error, then the empty
result or error is used as the basis for the answer returned to the
querying client. If instead the query results in one or more A RRs,
the DNS64 synthesizes AAAA RRs based on the A RRs according to the
procedure outlined in Section 5.1.7. The DNS64 returns the
synthesized AAAA records in the answer section, removing the A
records that form the basis of the synthesis.

5.1.7. Performing the synthesis

   A synthetic AAAA record is created from an A record as follows:

   o The NAME field is set to the NAME field from the A record.

   o The TYPE field is set to 28 (AAAA).

   o The CLASS field is set to the original CLASS field, 1. Under this
      specification, DNS64 for any CLASS other than 1 is undefined.

   o The TTL field is set to the minimum of the TTL of the original A
      RR and the SOA RR for the queried domain. (Note that in order to
      obtain the TTL of the SOA RR, the DNS64 does not need to perform a
      new query, but it can remember the TTL from the SOA RR in the
      negative response to the AAAA query. If the SOA RR was not
      delivered with the negative response to the AAAA query, then the
      DNS64 SHOULD use a the minimum of the TTL of the original A RR and
      600 seconds. It is possible instead to query explicitly for the
      SOA RR and use the result of that query, but this will increase
      query load and time to resolution for little additional benefit.)
      This is in keeping with the approach used in negative caching
      ([RFC2308].

   o The RDLENGTH field is set to 16.

   o The RDATA field is set to the IPv6 representation of the IPv4
      address from the RDATA field of the A record. The DNS64 MUST
      check each A RR against configured IPv4 address ranges and select
      the corresponding IPv6 prefix to use in synthesizing the AAAA RR.
      See Section 5.2 for discussion of the algorithms to be used in
      effecting the transformation.

5.1.8. Querying in parallel

   The DNS64 MAY perform the query for the AAAA RR and for the A RR in
   parallel, in order to minimize the delay.

   Note: Querying in parallel will result in performing unnecessary A RR
   queries in the case where no AAAA RR synthesis is required. A

possible trade-off would be to perform them sequentially but with a
very short interval between them, so if we obtain a fast reply, we
avoid doing the additional query. (Note that this discussion is
relevant only if the DNS64 function needs to perform external queries
t fetch the RR. If the needed RR information is available locally,
as in the case of an authoritative server, the issue is no longer
relevant.)

5.2. Generation of the IPv6 representations of IPv4 addresses

   DNS64 supports multiple algorithms for the generation of the IPv6
   representation of an IPv4 address. The constraints imposed on the
   generation algorithms are the following:

      The same algorithm to create an IPv6 address from an IPv4 address
      MUST be used by both a DNS64 to create the IPv6 address to be
      returned in the synthetic AAAA RR from the IPv4 address contained
      in an original A RR, and by a IPv6/IPv4 translator to create the
      IPv6 address to be included in the source address field of the
      outgoing IPv6 packets from the IPv4 address included in the source
      address field of the incoming IPv4 packet.

      The algorithm MUST be reversible; i.e., it MUST be possible to
      derive the original IPv4 address from the IPv6 representation.

      The input for the algorithm MUST be limited to the IPv4 address,
      the IPv6 prefix (denoted Pref64::/n) used in the IPv6
      representations and optionally a set of stable parameters that are
      configured in the DNS64 and in the NAT64 (such as fixed string to
      be used as a suffix).

         For each prefix Pref64::/n, n MUST be less than or equal to 96.
         If one or more Pref64::/n are configured in the DNS64 through
         any means (such as manually configured, or other automatic
         means not specified in this document), the default algorithm
         MUST use these prefixes (and not use the Well-Known Prefix).
         If no prefix is available, the algorithm MUST use the Well-
         Known Prefix 64:FF9B::/96 defined in
         [I-D.ietf-behave-address-format] to represent the IPv4 unicast
         address range

      [[anchor6: Note in document: The value 64:FF9B::/96 is proposed as
      the value for the Well-Known prefix and needs to be confirmed
      whenis published as RFC.]][I-D.ietf-behave-address-format]

   A DNS64 MUST support the algorithm for generating IPv6
   representations of IPv4 addresses defined in Section 2 of
   [I-D.ietf-behave-address-format]. Moreover, the aforementioned

algorithm MUST be the default algorithm used by the DNS64. While the
normative description of the algorithm is provided in
[I-D.ietf-behave-address-format], a sample description of the
algorithm and its application to different scenarios is provided in
Section 7 for illustration purposes.

5.3. Handling other Resource Records and the Additional Section

5.3.1. PTR Resource Record

If a DNS64 server receives a PTR query for a record in the IP6.ARPA
domain, it MUST strip the IP6.ARPA labels from the QNAME, reverse the
address portion of the QNAME according to the encoding scheme
outlined in section 2.5 of [RFC3596], and examine the resulting
address to see whether its prefix matches any of the locally-
configured Pref64::/n or the default Well-known prefix. There are
two alternatives for a DNS64 server to respond to such PTR queries.
A DNS64 server MUST provide one of these, and SHOULD NOT provide both
at the same time unless different IP6.ARPA zones require answers of
different sorts:

1. The first option is for the DNS64 server to respond
    authoritatively for its prefixes. If the address prefix matches
    any Pref64::/n used in the site, either a NSP or the Well-Known
    Prefix (i.e. 64:FF9B::/96), then the DNS64 server MAY answer the
    query using locally-appropriate RDATA. The DNS64 server MAY use
    the same RDATA for all answers. Note that the requirement is to
    match any Pref64::/n used at the site, and not merely the
    locally-configured Pref64::/n. This is because end clients could
    ask for a PTR record matching an address received through a
    different (site-provided) DNS64, and if this strategy is in
    effect, those queries should never be sent to the global DNS.
    The advantage of this strategy is that it makes plain to the
    querying client that the prefix is one operated by the (DNS64)
    site, and that the answers the client is getting are generated by
    DNS64. The disadvantage is that any useful reverse-tree
    information that might be in the global DNS is unavailable to the
    clients querying the DNS64.

2. The second option is for the DNS64 nameserver to synthesize a
    CNAME mapping the IP6.ARPA namespace to the corresponding IN-
    ADDR.ARPA name. In this case, the DNS64 nameserver SHOULD ensure
    that there is RDATA at the PTR of the corresponding IN-ADDR.ARPA
    name, and that there is not an existing CNAME at that name. This
    is in order to avoid synthesizing a CNAME that makes a CNAME
    chain longer or that does not actually point to anything. The
    rest of the response would be the normal DNS processing. The
    CNAME can be signed on the fly if need be. The advantage of this

approach is that any useful information in the reverse tree is
available to the querying client. The disadvantage is that it
adds additional load to the DNS64 (because CNAMEs have to be
synthesized for each PTR query that matches the Pref64::/n), and
that it may require signing on the fly.

If the address prefix does not match any Pref64::/n, then the DNS64
server MUST process the query as though it were any other query; i.e.
a recursive nameserver MUST attempt to resolve the query as though it
were any other (non-A/AAAA) query, and an authoritative server MUST
respond authoritatively or with a referral, as appropriate.

5.3.2. Handling the additional section

DNS64 synthesis MUST NOT be performed on any records in the
additional section of synthesized answers. The DNS64 MUST pass the
additional section unchanged.

NOTE: It may appear that adding synthetic records to the
additional section is desirable, because clients sometimes use the
data in the additional section to proceed without having to re-
query. There is in general no promise, however, that the
additional section will contain all the relevant records, so any
client that depends on the additional section being able to
satisfy its needs (i.e. without additional queries) is necessarily
broken. An IPv6-only client that needs a AAAA record, therefore,
will send a query for the necessary AAAA record if it is unable to
find such a record in the additional section of an answer it is
consuming. For a correctly-functioning client, the effect would
be no different if the additional section were empty.The
alternative, of removing the A records in the additional section
and replacing them with synthetic AAAA records, may cause a host
behind a NAT64 to query directly a nameserver that is unaware of
the NAT64 in question. The result in this case will be resolution
failure anyway, only later in the resolution operation. The
prohibition on synthetic data in the additional section reduces,
but does not eliminate, the possibility of resolution failures due
to cached DNS data from behind the DNS64. See Section 6.

5.3.3. Other Resource Records

If the DNS64 is in recursive resolver mode, then considerations
outlined in [I-D.ietf-dnsop-default-local-zones] may be relevant.

All other RRs MUST be returned unchanged. This includes responses to
queries for A RRs.

5.4. Assembling a synthesized response to a AAAA query

   A DNS64 uses different pieces of data to build the response returned
   to the querying client.

   The query that is used as the basis for synthesis results either in
   an error, an answer that can be used as a basis for synthesis, or an
   empty (authoritative) answer. If there is an empty answer, then the
   DNS64 responds to the original querying client with the answer the
   DNS64 received to the original (initiator's) query. Otherwise, the
   response is assembled as follows.

   The header fields are set according to the usual rules for recursive
   or authoritative servers, depending on the role that the DNS64 is
   serving. The question section is copied from the original
   (initiator's) query. The answer section is populated according to
   the rules in Section 5.1.7. The authority and additional sections
   are copied from the response to the final query that the DNS64
   performed, and used as the basis for synthesis.

   The final response from the DNS64 is subject to all the standard DNS
   rules, including truncation [RFC1035] and EDNS0 handling [RFC2671].

5.5. DNSSEC processing: DNS64 in validating resolver mode

   We consider the case where a recursive resolver that is performing
   DNS64 also has a local policy to validate the answers according to
   the procedures outlined in [RFC4035] Section 5. We call this general
   case vDNS64.

   The vDNS64 uses the presence of the DO and CD bits to make some
   decisions about what the query originator needs, and can react
   accordingly:

   1. If CD is not set and DO is not set, vDNS64 SHOULD perform
       validation and do synthesis as needed. See the next item for
       rules about how to do validation and synthesis. In this case,
       however, vDNS64 MUST NOT set the AD bit in any response.

   2. If CD is not set and DO is set, then vDNS64 SHOULD perform
       validation. Whenever vDNS64 performs validation, it MUST
       validate the negative answer for AAAA queries before proceeding
       to query for A records for the same name, in order to be sure
       that there is not a legitimate AAAA record on the Internet.
       Failing to observe this step would allow an attacker to use DNS64
       as a mechanism to circumvent DNSSEC. If the negative response
       validates, and the response to the A query validates, then the
       vDNS64 MAY perform synthesis and SHOULD set the AD bit in the

answer to the client. This is acceptable, because [RFC4035],
section 3.2.3 says that the AD bit is set by the name server side
of a security-aware recursive name server if and only if it
considers all the RRSets in the Answer and Authority sections to
be authentic. In this case, the name server has reason to
believe the RRSets are all authentic, so it SHOULD set the AD
bit. If the data does not validate, the vDNS64 MUST respond with
RCODE=2 (Server failure).
A security-aware end point might take the presence of the AD bit
as an indication that the data is valid, and may pass the DNS
(and DNSSEC) data to an application. If the application attempts
to validate the synthesized data, of course, the validation will
fail. One could argue therefore that this approach is not
desirable, but security aware stub resolvers must not place any
reliance on data received from resolvers and validated on their
behalf without certain criteria established by [RFC4035], section
4.9.3. An application that wants to perform validation on its
own should use the CD bit.

3. If the CD bit is set and DO is set, then vDNS64 MAY perform
    validation, but MUST NOT perform synthesis. It MUST return the
    data to the query initiator, just like a regular recursive
    resolver, and depend on the client to do the validation and the
    synthesis itself.
    The disadvantage to this approach is that an end point that is
    translation-oblivious but security-aware and validating will not
    be able to use the DNS64 functionality. In this case, the end
    point will not have the desired benefit of NAT64. In effect,
    this strategy means that any end point that wishes to do
    validation in a NAT64 context must be upgraded to be translation-
    aware as well.

6. Deployment notes

   While DNS64 is intended to be part of a strategy for aiding IPv6
   deployment in an internetworking environment with some IPv4-only and
   IPv6-only networks, it is important to realise that it is
   incompatible with some things that may be deployed in an IPv4-only or
   dual-stack context.

6.1. DNS resolvers and DNS64

   Full-service resolvers that are unaware of the DNS64 function can be
   (mis)configured to act as mixed-mode iterative and forwarding
   resolvers. In a native IPv4 context, this sort of configuration may
   appear to work. It is impossible to make it work properly without it
   being aware of the DNS64 function, because it will likely at some

point obtain IPv4-only glue records and attempt to use them for
resolution. The result that is returned will contain only A records,
and without the ability to perform the DNS64 function the resolver
will be unable to answer the necessary AAAA queries.

6.2. DNSSEC validators and DNS64

An existing DNSSEC validator (i.e. that is unaware of DNS64) might
reject all the data that comes from DNS64 as having been tampered
with (even if it did not set CD when querying). If it is necessary
to have validation behind the DNS64, then the validator must know how
to perform the DNS64 function itself. Alternatively, the validating
host may establish a trusted connection with a DNS64, and allow the
DNS64 recursive resolver to do all validation on its behalf.

6.3. DNS64 and multihomed and dual-stack hosts

6.3.1. IPv6 multihomed hosts

Synthetic AAAA records may be constructed on the basis of the network
context in which they were constructed. If a host sends DNS queries
to resolvers in multiple networks, it is possible that some of them
will receive answers from a DNS64 without all of them being connected
via a NAT64. For instance, suppose a system has two interfaces, i1
and i2. Whereas i1 is connected to the IPv4 Internet via NAT64, i2
has native IPv6 connectivity only. I1 might receive a AAAA answer
from a DNS64 that is configured for a particular NAT64; the IPv6
address contained in that AAAA answer will not connect with anything
via i2.

```
        +---------------+             +------------+
        |      i1 (IPv6)+----NAT64--------+IPv4 Internet|
        |               |             +------------+
        | host          |
        |               |             +------------+
        |      i2 (IPv6)+----------------+IPv6 Internet|
        +---------------+             +------------+
```

Figure 1: IPv6 multihomed hosts

This example illustrates why it is generally preferable that hosts
treat DNS answers from one interface as local to that interface. The
answer received on one interface will not work on the other
interface. Hosts that attempt to use DNS answers globally may
encounter surprising failures in these cases.

Note that the issue is not that there are two interfaces, but that
there are two networks involved. The same results could be achieved

   with a single interface routed to two different networks.

6.3.2. Accidental dual-stack DNS64 use

   Similarly, suppose that i1 has IPv6 connectivity and can connect to
   the IPv4 Internet through NAT64, but i2 has native IPv4 connectivity.
   In this case, i1 could receive an IPv6 address from a synthetic AAAA
   that would better be reached via native IPv4. Again, it is worth
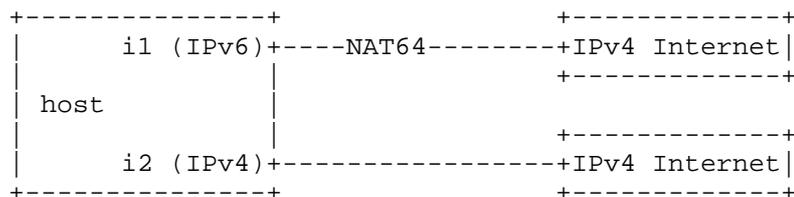   emphasising that this arises because there are two networks involved.

```
        +---------------+                +------------+
        |    i1 (IPv6)+----NAT64--------+IPv4 Internet|
        |             |                +------------+
        | host        |
        |             |                +------------+
        |    i2 (IPv4)+----------------+IPv4 Internet|
        +---------------+                +------------+
```

                 Figure 2: Accidental dual-stack DNS64 use

   The default configuration of dual-stack hosts is that IPv6 is
   preferred over IPv4 ([RFC3484]). In that arrangement the host will
   often use the NAT64 when native IPv4 would be more desirable. For
   this reason, hosts with IPv4 connectivity to the Internet should
   avoid using DNS64. This can be partly resolved by ISPs when
   providing DNS resolvers to clients, but that is not a guarantee that
   the NAT64 will never be used when a native IPv4 connection should be
   used. There is no general-purpose mechanism to ensure that native
   IPv4 transit will always be preferred, because to a DNS64-oblivious
   host, the DNS64 looks just like an ordinary DNS server. Operators of
   a NAT64 should expect traffic to pass through the NAT64 even when it
   is not necessary.

6.3.3. Intentional dual-stack DNS64 use

   Finally, consider the case where the IPv4 connectivity on i2 is only
   with a LAN, and not with the IPv4 Internet. The IPv4 Internet is
   only accessible using the NAT64. In this case, it is critical that
   the DNS64 not synthesize AAAA responses for hosts in the LAN, or else
   that the DNS64 be aware of hosts in the LAN and provide context-
   sensitive answers ("split view" DNS answers) for hosts inside the
   LAN. As with any split view DNS arrangement, operators must be
   prepared for data to leak from one context to another, and for
   failures to occur because nodes accessible from one context are not
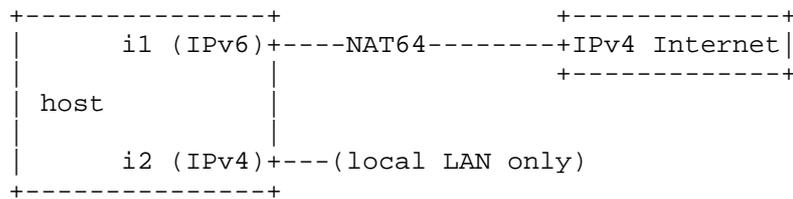   accessible from the other.

```
        +---------------+                +------------+
        |      i1 (IPv6)+----NAT64-------+IPv4 Internet|
        |               |                +------------+
        | host          |
        |               |
        |      i2 (IPv4)+---(local LAN only)
        +---------------+
```

                Figure 3: Intentional dual-stack DNS64 use

   It is important for deployers of DNS64 to realise that, in some
   circumstances, making the DNS64 available to a dual-stack host will
   cause the host to prefer to send packets via NAT64 instead of via
   native IPv4, with the associated loss of performance or functionality
   (or both) entailed by the NAT. At the same time, some hosts are not
   able to learn about DNS servers provisioned on IPv6 addresses, or
   simply cannot send DNS packets over IPv6.

7. Deployment scenarios and examples

   In this section we illustrate how the DNS64 behaves in different
   scenarios that are expected to be common. In particular we will
   consider the following scenarios defined in
   [I-D.ietf-behave-v6v4-framework]: the an-IPv6-network-to-IPv4-
   Internet scenario (both with DNS64 in DNS server mode and in stub-
   resolver mode) and the IPv6-Internet-to-an-IPv4-network setup (with
   DNS64 in DNS server mode only).

   In all the examples below, there is a IPv6/IPv4 translator connecting
   the IPv6 domain to the IPv4 one. Also there is a name server that is
   a dual-stack node, so it can communicate with IPv6 hosts using IPv6
   and with IPv4 nodes using IPv4. In addition, we assume that in the
   examples, the DNS64 function learns which IPv6 prefix it needs to use
   to map the IPv4 address space through manual configuration.

7.1. Example of An-IPv6-network-to-IPv4-Internet setup with DNS64 in
      DNS server mode

   In this example, we consider an IPv6 node located in an IPv6-only
   site that initiates a communication to an IPv4 node located in the
   IPv4 Internet.

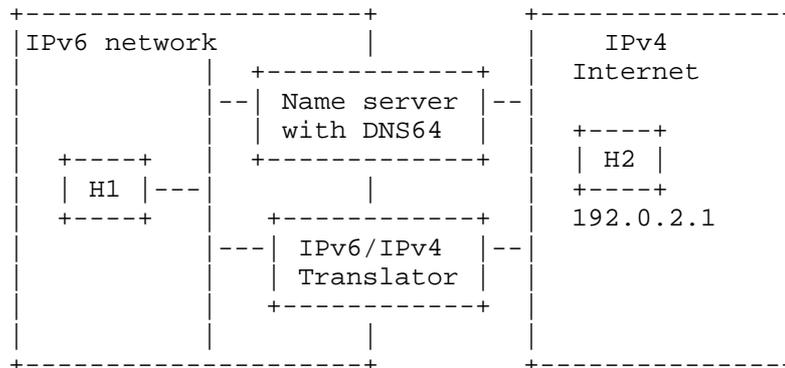   The scenario for this case is depicted in the following figure:

```
         +--------------------+      +--------------+
         |IPv6 network        |      |    IPv4      |
         |         | +-------------+ |   Internet   |
         |         |--| Name server |--|             |
         |         |  | with DNS64  | |  +----+      |
         | +----+  |  +-------------+ |  | H2 |      |
         | | H1 |---|                | |  +----+      |
         | +----+  |  +-----------+  |  192.0.2.1    |
         |         |---| IPv6/IPv4 |--|             |
         |         |  | Translator |  |             |
         |         |  +-----------+  |             |
         |         |        |       |             |
         +--------------------+      +--------------+
```

        Figure 4: An-IPv6-network-to-IPv4-Internet setup with DNS64 in DNS
                            server mode

   The figure shows an IPv6 node H1 and an IPv4 node H2 with IPv4
   address 192.0.2.1 and FQDN h2.example.com.

   The IPv6/IPv4 Translator has an IPv4 address 203.0.113.1 assigned to
   its IPv4 interface and it is using the WKP 64:FF9B::/96 to create
   IPv6 representations of IPv4 addresses. The same prefix is
   configured in the DNS64 function in the local name server.

   For this example, assume the typical DNS situation where IPv6 hosts
   have only stub resolvers, and they are configured with the IP address
   of a name server that they always have to query and that performs
   recursive lookups (henceforth called "the recursive nameserver").

   The steps by which H1 establishes communication with H2 are:

   1. H1 does a DNS lookup for h2.example.com. H1 does this by sending
       a DNS query for a AAAA record for H2 to the recursive name
       server. The recursive name server implements DNS64
       functionality.

   2. The recursive name server resolves the query, and discovers that
       there are no AAAA records for H2.

   3. The recursive name server performs an A-record query for H2 and
       gets back an RRset containing a single A record with the IPv4
       address 192.0.2.1. The name server then synthesizes a AAAA
       record. The IPv6 address in the AAAA record contains the prefix
       assigned to the IPv6/IPv4 Translator in the upper 96 bits and the
       received IPv4 address in the lower 32 bits i.e. the resulting
       IPv6 address is 64:FF9B::192.0.2.1.

4. H1 receives the synthetic AAAA record and sends a packet towards
    H2. The packet is sent to the destination address 64:FF9B::
    192.0.2.1.

5. The packet is routed to the IPv6 interface of the IPv6/IPv4
    translator and the subsequent communication flows by means of the
    IPv6/IPv4 translator mechanisms.

7.2. An example of an-IPv6-network-to-IPv4-Internet setup with DNS64 in
      stub-resolver mode
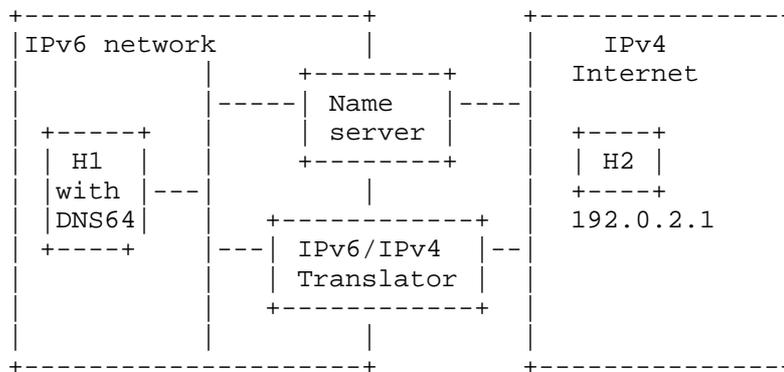
   This case is depicted in the following figure:

```
      +--------------------+       +--------------+
      |IPv6 network        |       |     IPv4     |
      |            |       +--------+       |   Internet    |
      |            |-----|  Name   |----|       |              |
      | +-----+    |     | server  |    |  +----+      |
      | | H1  |    |     +--------+     |  | H2 |      |
      | |with |---|           |        |  +----+      |
      | |DNS64|    |    +------------+   |  192.0.2.1   |
      | +----+    |---| IPv6/IPv4  |--|       |              |
      |            |   | Translator |   |       |              |
      |            |   +------------+   |       |              |
      |            |        |        |       |              |
      +--------------------+       +--------------+
```

          Figure 5: An-IPv6-network-to-IPv4-Internet setup with DNS64 in stub-
                              resolver mode

   The figure shows an IPv6 node H1 implementing the DNS64 function and
   an IPv4 node H2 with IPv4 address 192.0.2.1 and FQDN h2.example.com.

   The IPv6/IPv4 Translator has an IPv4 address 203.0.113.1 assigned to
   its IPv4 interface and it is using the WKP 64:FF9B::/96 to create
   IPv6 representations of IPv4 addresses. The same prefix is
   configured in the DNS64 function in H1.

   For this example, assume the typical DNS situation where IPv6 hosts
   have only stub resolvers, and they are configured with the IP address
   of a name server that they always have to query and that performs
   recursive lookups (henceforth called "the recursive nameserver").
   The recursive name server does not perform the DNS64 function.

   The steps by which H1 establishes communication with H2 are:

1. H1 does a DNS lookup for h2.example.com. H1 does this by sending
    a DNS query for a AAAA record for H2 to the recursive name
    server.

2. The recursive DNS server resolves the query, and returns the
    answer to H1. Because there are no AAAA records in the global
    DNS for H2, the answer is empty.

3. The stub resolver at H1 then queries for an A record for H2 and
    gets back an A record containing the IPv4 address 192.0.2.1. The
    DNS64 function within H1 then synthesizes a AAAA record. The
    IPv6 address in the AAAA record contains the prefix assigned to
    the IPv6/IPv4 translator in the upper 96 bits, then the received
    IPv4 address i.e. the resulting IPv6 address is 64:FF9B::
    192.0.2.1.

4. H1 sends a packet towards H2. The packet is sent to the
    destination address 64:FF9B::192.0.2.1.

5. The packet is routed to the IPv6 interface of the IPv6/IPv4
    translator and the subsequent communication flows using the IPv6/
    IPv4 translator mechanisms.

7.3. Example of IPv6-Internet-to-an-IPv4-network setup DNS64 in DNS
      server mode

   In this example, we consider an IPv6 node located in the IPv6
   Internet that initiates a communication to an IPv4 node located in
   the IPv4 site.

   In some cases, this scenario can be addressed without using any form
   of DNS64 function. This is so because it is possible to assign a
   fixed IPv6 address to each of the IPv4 nodes. Such an IPv6 address
   would be constructed using the address transformation algorithm
   defined in [I-D.ietf-behave-address-format] that takes as input the
   Pref64::/96 and the IPv4 address of the IPv4 node. Note that the
   IPv4 address can be a public or a private address; the latter does
   not present any additional difficulty, since an NSP must be used as
   Pref64::/96 (in this scenario the usage of the Well-Known prefix is
   not supported as discussed in [I-D.ietf-behave-address-format]).
   Once these IPv6 addresses have been assigned to represent the IPv4
   nodes in the IPv6 Internet, real AAAA RRs containing these addresses
   can be published in the DNS under the site's domain. This is the
   recommended approach to handle this scenario, because it does not
   involve synthesizing AAAA records at the time of query.

   However, there are some more dynamic scenarios, where synthesizing
   AAAA RRs in this setup may be needed. In particular, when DNS Update

[RFC2136] is used in the IPv4 site to update the A RRs for the IPv4
nodes, there are two options: One option is to modify the DNS server
that receives the dynamic DNS updates. That would normally be the
authoritative server for the zone. So the authoritative zone would
have normal AAAA RRs that are synthesized as dynamic updates occur.
The other option is modify all the authoritative servers to generate
synthetic AAAA records for a zone, possibly based on additional
constraints, upon the receipt of a DNS query for the AAAA RR. The
first option -- in which the AAAA is synthesized when the DNS update
message is received, and the data published in the relevant zone --
is recommended over the second option (i.e. the synthesis upon
receipt of the AAAA DNS query). This is because it is usually easier
to solve problems of misconfiguration when the DNS responses are not
being generated dynamically. However, it may be the case where the
primary server (that receives all the updates) cannot be upgraded for
whatever reason, but where a secondary can be upgraded in order to
handle the (comparatively small amount) of AAAA queries. In such
case, it is possible to use the DNS64 as described next. The DNS64
behavior that we describe in this section covers the case of
synthesizing the AAAA RR when the DNS query arrives.

The scenario for this case is depicted in the following figure:

```
          +----------+           +----------------------+
          |          |           |  IPv4 site           |
          |  IPv6    |   +-----------+  |   +----+       |
          |  Internet |----|  IPv6/IPv4 |--|---|  H2 |   |
          |          |   |  Translator |  |   +----+       |
          |          |   +-----------+  |              |
          |          |           |      | 192.0.2.1    |
          |          |   +-----------+  |              |
          |          |----|  Name server|--|              |
          |          |   |  with DNS64 |  |              |
          +----------+   +-----------+  |              |
             |               |      |              |
          +----+               |      |              |
          | H1 |           +----------------------+
          +----+
```
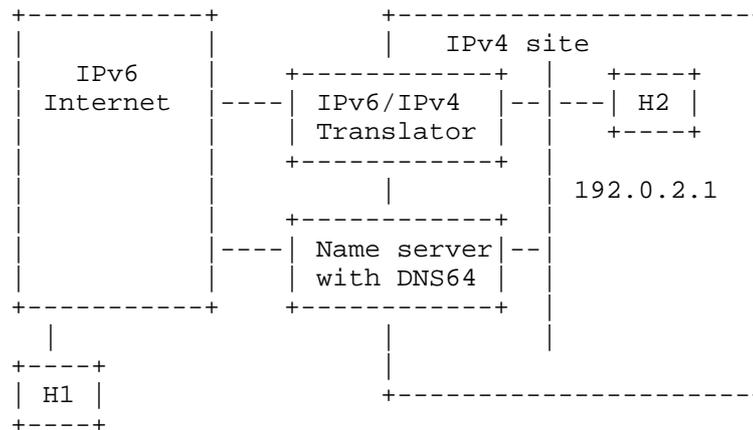
              Figure 6: IPv6-Internet-to-an-IPv4-network setup DNS64 in DNS server
                                  mode

The figure shows an IPv6 node H1 and an IPv4 node H2 with IPv4
address 192.0.2.1 and FQDN h2.example.com.

The IPv6/IPv4 Translator is using a NSP 2001:DB8::/96 to create IPv6
representations of IPv4 addresses. The same prefix is configured in

the DNS64 function in the local name server. The name server that
implements the DNS64 function is the authoritative name server for
the local domain.

The steps by which H1 establishes communication with H2 are:

1. H1 does a DNS lookup for h2.example.com. H1 does this by sending
    a DNS query for a AAAA record for H2. The query is eventually
    forwarded to the server in the IPv4 site.

2. The local DNS server resolves the query (locally), and discovers
    that there are no AAAA records for H2.

3. The name server verifies that h2.example.com and its A RR are
    among those that the local policy defines as allowed to generate
    a AAAA RR from. If that is the case, the name server synthesizes
    a AAAA record from the A RR and the prefix 2001:DB8::/96. The
    IPv6 address in the AAAA record is 2001:DB8::192.0.2.1.

4. H1 receives the synthetic AAAA record and sends a packet towards
    H2. The packet is sent to the destination address 2001:DB8::
    192.0.2.1.

5. The packet is routed through the IPv6 Internet to the IPv6
    interface of the IPv6/IPv4 translator and the communication flows
    using the IPv6/IPv4 translator mechanisms.

8. Security Considerations

   DNS64 operates in combination with the DNS, and is therefore subject
   to whatever security considerations are appropriate to the DNS mode
   in which the DNS64 is operating (i.e. authoritative, recursive, or
   stub resolver mode).

   DNS64 has the potential to interfere with the functioning of DNSSEC,
   because DNS64 modifies DNS answers, and DNSSEC is designed to detect
   such modification and to treat modified answers as bogus. See the
   discussion above in Section 3, Section 5.5, and Section 6.2.

   Additionally, for the correct functioning of the translation
   services, the DNS64 and the NAT64 need to use the same Pref64. If an
   attacker manages to change the Pref64 used by the DNS64, the traffic
   generated by the host that receives the synthetic reply will be
   delivered to the altered Pref64. This can result in either a DoS
   attack (if resulting IPv6 addresses are not assigned to any device)
   or in a flooding attack (if the resulting IPv6 addresses are assigned
   to devices that do not wish to receive the traffic) or in

eavesdropping attack (in case the Pref64 is routed through the attacker).

9. IANA Considerations

This memo makes no request of IANA.

10. Contributors

Dave Thaler

Microsoft

dthaler@windows.microsoft.com

11. Acknowledgements

This draft contains the result of discussions involving many people, including the participants of the IETF BEHAVE Working Group. The following IETF participants made specific contributions to parts of the text, and their help is gratefully acknowledged: Jaap Akkerhuis, Mark Andrews, Jari Arkko, Rob Austein, Timothy Baldwin, Fred Baker, Doug Barton, Marc Blanchet, Cameron Byrne, Brian Carpenter, Zhen Cao, Hui Deng, Francis Dupont, Patrik Faltstrom, David Harrington, Ed Jankiewicz, Peter Koch, Suresh Krishnan, Martti Kuparinen, Ed Lewis, Xing Li, Bill Manning, Matthijs Mekking, Hiroshi Miyata, Simon Perrault, Teemu Savolainen, Jyrki Soini, Dave Thaler, Mark Townsley, Rick van Rein, Stig Venaas, Magnus Westerlund, Jeff Westhead, Florian Weimer, Dan Wing, Xu Xiaohu, Xiangsong Cui.

Marcelo Bagnulo and Iljitsch van Beijnum are partly funded by Trilogy, a research project supported by the European Commission under its Seventh Framework Program.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

   [RFC1035] Mockapetris, P., "Domain names - implementation and
              specification", STD 13, RFC 1035, November 1987.

   [RFC4787] Audet, F. and C. Jennings, "Network Address Translation
              (NAT) Behavioral Requirements for Unicast UDP", BCP 127,
              RFC 4787, January 2007.

   [RFC2671] Vixie, P., "Extension Mechanisms for DNS (EDNS0)",
              RFC 2671, August 1999.

   [I-D.ietf-behave-address-format]
              Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X.
              Li, "IPv6 Addressing of IPv4/IPv6 Translators",
              draft-ietf-behave-address-format-10 (work in progress),
              August 2010.

12.2. Informative References

   [I-D.ietf-behave-v6v4-xlate-stateful]
              Bagnulo, M., Matthews, P., and I. Beijnum, "Stateful
              NAT64: Network Address and Protocol Translation from IPv6
              Clients to IPv4 Servers",
              draft-ietf-behave-v6v4-xlate-stateful-12 (work in
              progress), July 2010.

   [RFC2136] Vixie, P., Thomson, S., Rekhter, Y., and J. Bound,
              "Dynamic Updates in the Domain Name System (DNS UPDATE)",
              RFC 2136, April 1997.

   [RFC2308] Andrews, M., "Negative Caching of DNS Queries (DNS
              NCACHE)", RFC 2308, March 1998.

   [RFC3484] Draves, R., "Default Address Selection for Internet
              Protocol version 6 (IPv6)", RFC 3484, February 2003.

   [RFC3596] Thomson, S., Huitema, C., Ksinant, V., and M. Souissi,
              "DNS Extensions to Support IP Version 6", RFC 3596,
              October 2003.

   [RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S.
              Rose, "DNS Security Introduction and Requirements",
              RFC 4033, March 2005.

   [RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S.
              Rose, "Resource Records for the DNS Security Extensions",
              RFC 4034, March 2005.

   [RFC4035] Arends, R., Austein, R., Larson, M., Massey, D., and S.

                Rose, "Protocol Modifications for the DNS Security
                Extensions", RFC 4035, March 2005.

   [RFC4074]   Morishita, Y. and T. Jinmei, "Common Misbehavior Against
                DNS Queries for IPv6 Addresses", RFC 4074, May 2005.

   [RFC5735]   Cotton, M. and L. Vegoda, "Special Use IPv4 Addresses",
                BCP 153, RFC 5735, January 2010.

   [I-D.ietf-behave-v6v4-framework]
                Baker, F., Li, X., Bao, C., and K. Yin, "Framework for
                IPv4/IPv6 Translation",
                draft-ietf-behave-v6v4-framework-10 (work in progress),
                August 2010.

   [I-D.ietf-dnsop-default-local-zones]
                Andrews, M., "Locally-served DNS Zones",
                draft-ietf-dnsop-default-local-zones-14 (work in
                progress), September 2010.

Appendix A. Motivations and Implications of synthesizing AAAA Resource
             Records when real AAAA Resource Records exist

   The motivation for synthesizing AAAA RRs when real AAAA RRs exist is
   to support the following scenario:

      An IPv4-only server application (e.g. web server software) is
      running on a dual-stack host. There may also be dual-stack server
      applications running on the same host. That host has fully
      routable IPv4 and IPv6 addresses and hence the authoritative DNS
      server has an A and a AAAA record.

      An IPv6-only client (regardless of whether the client application
      is IPv6-only, the client stack is IPv6-only, or it only has an
      IPv6 address) wants to access the above server.

      The client issues a DNS query to a DNS64 resolver.

   If the DNS64 only generates a synthetic AAAA if there's no real AAAA,
   then the communication will fail. Even though there's a real AAAA,
   the only way for communication to succeed is with the translated
   address. So, in order to support this scenario, the administrator of
   a DNS64 service may want to enable the synthesis of AAAA RRs even
   when real AAAA RRs exist.

   The implication of including synthetic AAAA RRs when real AAAA RRs
   exist is that translated connectivity may be preferred over native

connectivity in some cases where the DNS64 is operated in DNS server
mode.

RFC3484 [RFC3484] rules use longest prefix match to select the
preferred destination address to use. So, if the DNS64 resolver
returns both the synthetic AAAA RRs and the real AAAA RRs, then if
the DNS64 is operated by the same domain as the initiating host, and
a global unicast prefix (called an NSP in
[I-D.ietf-behave-address-format]) is used, then a synthetic AAAA RR
is likely to be preferred.

This means that without further configuration:

   In the "An IPv6 network to the IPv4 Internet" scenario, the host
   will prefer translated connectivity if an NSP is used. If the
   Well-Known Prefix defined in [I-D.ietf-behave-address-format] is
   used, it will probably prefer native connectivity.

   In the "IPv6 Internet to an IPv4 network" scenario, it is possible
   to bias the selection towards the real AAAA RR if the DNS64
   resolver returns the real AAAA first in the DNS reply, when an NSP
   is used (the Well-Known Prefix usage is not supported in this
   case)

   In the "An IPv6 network to IPv4 network" scenario, for local
   destinations (i.e., target hosts inside the local site), it is
   likely that the NSP and the destination prefix are the same, so we
   can use the order of RR in the DNS reply to bias the selection
   through native connectivity. If the Well-Known Prefix is used,
   the longest prefix match rule will select native connectivity.

The problem can be solved by properly configuring the RFC3484
[RFC3484] policy table.

Authors' Addresses

   Marcelo Bagnulo
   UC3M
   Av. Universidad 30
   Leganes, Madrid 28911
   Spain

   Phone: +34-91-6249500
   Fax:
   Email: marcelo@it.uc3m.es
   URI: http://www.it.uc3m.es/marcelo

Andrew Sullivan
Shinkuro
4922 Fairmont Avenue, Suite 250
Bethesda, MD 20814
USA

Phone: +1 301 961 3131
Email: ajs@shinkuro.com

Philip Matthews
Unaffiliated
600 March Road
Ottawa, Ontario
Canada

Phone: +1 613-592-4343 x224
Fax:
Email: philip_matthews@magma.ca
URI:

Iljitsch van Beijnum
IMDEA Networks
Av. Universidad 30
Leganes, Madrid 28911
Spain

Phone: +34-91-6246245
Email: iljitsch@muada.com

behave   X. Li
Internet-Draft   C. Bao
Obsoletes: 2765 (if approved)   CERNET Center/Tsinghua University
Intended status: Standards Track   F. Baker
Expires: March 22, 2011   Cisco Systems
                                                  September 18, 2010

                    IP/ICMP Translation Algorithm
                    draft-ietf-behave-v6v4-xlate-23

Abstract

   This document describes the Stateless IP/ICMP Translation Algorithm
   (SIIT), which translates between IPv4 and IPv6 packet headers
   (including ICMP headers). This document obsoletes RFC2765.

Status of this Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on March 22, 2011.

Copyright Notice

Table of Contents

1. Introduction and Motivation

   This document is a product of the 2008-2010 effort to define a
   replacement for NAT-PT [RFC2766]. It is directly derivative from
   Erik Nordmark's "Stateless IP/ICMP Translation Algorithm (SIIT)"
   [RFC2765], which provides stateless translation between IPv4
   [RFC0791] and IPv6 [RFC2460], and between ICMPv4 [RFC0792] and ICMPv6
   [RFC4443]. This document obsoletes RFC2765 [RFC2765]. The changes
   from RFC2765 [RFC2765] are listed in Section 2.

   Readers of this document are expected to have read and understood the
   framework described in [I-D.ietf-behave-v6v4-framework].
   Implementations of this IPv4/IPv6 translation specification MUST also
   support the address translation algorithms in
   [I-D.ietf-behave-address-format]. Implementations MAY also support
   stateful translation [I-D.ietf-behave-v6v4-xlate-stateful].

1.1. IPv4-IPv6 Translation Model

   The translation model consists of two or more network domains
   connected by one or more IP/ICMP translators (XLATs) as shown in
   Figure 1.

```
               ---------         ---------
            //         \\     //         \\
          /           +----+             \
          |           |XLAT|             | XLAT: IP/ICMP
          |   IPv4     +----+    IPv6     |        Translator
          |  Domain    |    |  Domain     |
          |            |    |             |
          \            |    |            /
           \\         //     \\         //
             --------          ---------
```
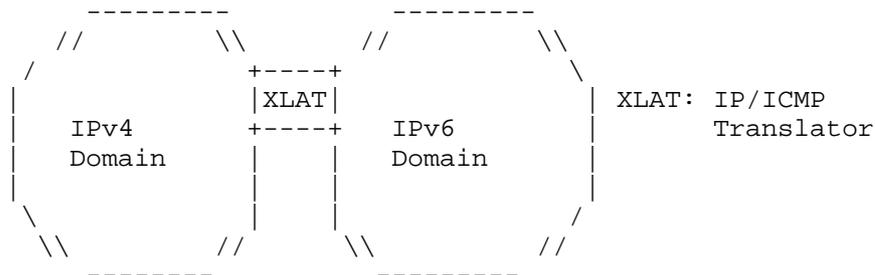
                Figure 1: IPv4-IPv6 Translation Model

   The scenarios of the translation model are discussed in
   [I-D.ietf-behave-v6v4-framework].

1.2. Applicability and Limitations

   This document specifies the translation algorithms between IPv4
   packets and IPv6 packets.

   As with [RFC2765], the translating function specified in this
   document does not translate any IPv4 options and it does not

translate IPv6 extension headers except fragmentation header.

The issues and algorithms in the translation of datagrams containing TCP segments are described in [RFC5382].

Fragmented IPv4 UDP packets that do not contain a UDP checksum (i.e., the UDP checksum field is zero) are not of significant use in the Internet and in general will not be translated by the IP/ICMP translator. However, when the translator is configured to forward the packet without a UDP checksum, the fragmented IPv4 UDP packets will be translated.

Fragmented ICMP/ICMPv6 packets will not be translated by the IP/ICMP translator.

The IP/ICMP header translation specified in this document is consistent with requirements of multicast IP/ICMP headers. However IPv4 multicast addresses [RFC5771] cannot be mapped to IPv6 multicast addresses [RFC3307] based on the unicast mapping rule [I-D.ietf-behave-address-format].

1.3. Stateless vs. Stateful Mode

An IP/ICMP translator has two possible modes of operation: stateless and stateful [I-D.ietf-behave-v6v4-framework]. In both cases, we assume that a system (a node or an application) that has an IPv4 address but not an IPv6 address is communicating with a system that has an IPv6 address but no IPv4 address, or that the two systems do not have contiguous routing connectivity and hence are forced to have their communications translated.

In the stateless mode, a specific IPv6 address range will represent IPv4 systems (IPv4-converted addresses), and the IPv6 systems have addresses (IPv4-translatable addresses) that can be algorithmically mapped to a subset of the service provider's IPv4 addresses. Note that IPv4-translatable addresses is a subset of IPv4-converted addresses. In general, there is no need to concern oneself with translation tables, as the IPv4 and IPv6 counterparts are algorithmically related.

In the stateful mode, a specific IPv6 address range will represent IPv4 systems (IPv4-converted addresses), but the IPv6 systems may use any IPv6 addresses [RFC4291] except in that range. In this case, a translation table is required to bind the IPv6 systems' addresses to the IPv4 addresses maintained in the translator.

The address translation mechanisms for the stateless and the stateful translations are defined in [I-D.ietf-behave-address-format].

1.4. Path MTU Discovery and Fragmentation

   Due to the different sizes of the IPv4 and IPv6 header, which are 20+
   octets and 40 octets respectively, handling the maximum packet size
   is critical for the operation of the IPv4/IPv6 translator. There are
   three mechanisms to handle this issue: path MTU discovery (PMTUD),
   fragmentation, and transport-layer negotiation such as the TCP MSS
   option [RFC0879]. Note that the translator MUST behave as a router,
   i.e. the translator MUST send a "Packet Too Big" error message or
   fragment the packet when the packet size exceeds the MTU of the next
   hop interface.

   "Don't Fragment", ICMP "Packet Too Big", and packet fragmentation are
   discussed in Section 4 and Section 5 of this document. The
   reassembling of fragmented packets in the stateful translator is
   discussed in [I-D.ietf-behave-v6v4-xlate-stateful], since it requires
   state maintenance in the translator.

2. Changes from RFC2765

   The changes from RFC2765 are the following:

   1. Redescribing the network model to map to present and projected
       usage. The scenarios, applicability and limitations originally
       presented in RFC2765 [RFC2765] are moved to framework document
       [I-D.ietf-behave-v6v4-framework].

   2. Moving the address format to the address format document
       [I-D.ietf-behave-address-format], to coordinate with other
       documents on the topic.

   3. Describing the header translation for the stateless and stateful
       operations. The details of the session database and mapping
       table handling of the stateful translation is in stateful
       translation document [I-D.ietf-behave-v6v4-xlate-stateful].

   4. Having refined the header translation, fragmentation handling,
       ICMP translation and ICMP error translation in IPv4 to IPv6
       direction, as well as in IPv6 to IPv4 direction.

   5. Adding more discussion on transport-layer header translation.

   6. Adding Section 5.1.1 for "IPv6 Fragment Processing".

   7. Adding Section 6 for "Special Considerations for ICMPv6 Packet
       Too Big".

   8. Having updated Section 8 for "Security Considerations".

   9. Adding Section 10 "Stateless translation workflow example".

3. Conventions

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in [RFC2119].

4. Translating from IPv4 to IPv6

   When an IP/ICMP translator receives an IPv4 datagram addressed to a
   destination towards the IPv6 domain, it translates the IPv4 header of
   that packet into an IPv6 header. The original IPv4 header on the
   packet is removed and replaced by an IPv6 header and the transport
   checksum updated as needed, if that transport is supported by the
   translator. The data portion of the packet is left unchanged. The
   IP/ICMP translator then forwards the packet based on the IPv6
   destination address.

```
           +-------------+              +-------------+
           |    IPv4     |              |    IPv6     |
           |   Header    |              |   Header    |
           +-------------+              +-------------+
           |  Transport  |              |  Fragment   |
           |   Layer     |    ===>      |   Header    |
           |   Header    |              | (if needed) |
           +-------------+              +-------------+
           |             |              |  Transport  |
           ~    Data     ~              |   Layer     |
           |             |              |   Header    |
           +-------------+              +-------------+
                                        |             |
                                        ~    Data     ~
                                        |             |
                                        +-------------+
```
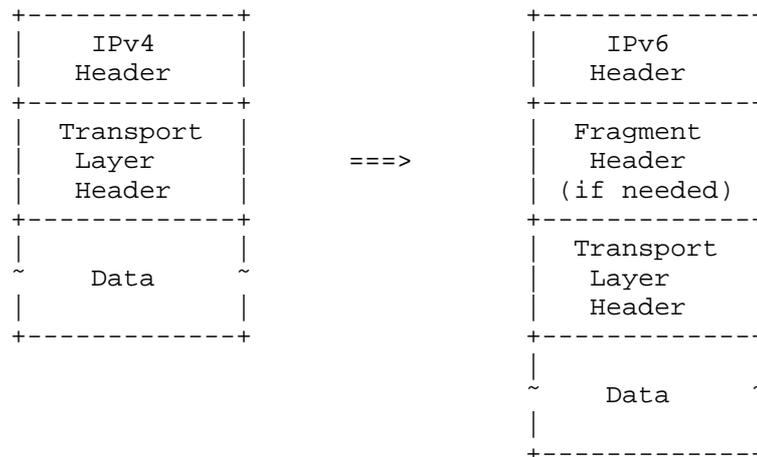
              Figure 2: IPv4-to-IPv6 Translation

   Path MTU discovery is mandatory in IPv6 but it is optional in IPv4.
   IPv6 routers never fragment a packet - only the sender can do
   fragmentation.

   When an IPv4 node performs path MTU discovery (by setting the Don't
   Fragment (DF) bit in the header), path MTU discovery can operate end-

to-end, i.e., across the translator. In this case either IPv4 or
IPv6 routers (including the translator) might send back ICMP "Packet
Too Big" messages to the sender. When the IPv6 routers send these
ICMPv6 errors they will pass through a translator that will translate
the ICMPv6 error to a form that the IPv4 sender can understand. As a
result, an IPv6 fragment header is only included if the IPv4 packet
is already fragmented.

However, when the IPv4 sender does not set the Don't Fragment (DF)
bit, the translator MUST ensure that the packet does not exceed the
path MTU on the IPv6 side. This is done by fragmenting the IPv4
packet (with fragmentation headers) so that it fits in 1280-byte IPv6
packets, since that is the minimum IPv6 MTU. The IPv6 fragmentation
header has been shown to cause operational difficulties in practice
due to limited firewall fragmentation support, etc.. In an
environment where the network owned/operated by the same entity that
owns/operates the translator, the translator MAY provide a
configuration function for the network administrator to adjust the
threshold of the minimum IPv6 MTU to a value that reflects the real
value of the minimum IPv6 MTU in the network (greater than 1280-
byte). This will help reduce the chance of including the
fragmentation header in the packets.

When the IPv4 sender does not set the DF bit the translator SHOULD
always include an IPv6 fragment header to indicate that the sender
allows fragmentation. The translator MAY provide a configuration
function that allows the translator not to include the fragmentation
header for the non-fragmented IPv6 packets.

The rules in Section 4.1 ensure that when packets are fragmented,
either by the sender or by IPv4 routers, the low-order 16 bits of the
fragment identification are carried end-to-end, ensuring that packets
are correctly reassembled. In addition, the rules in Section 4.1 use
the presence of an IPv6 fragment header to indicate that the sender
might not be using path MTU discovery (i.e., the packet should not
have the DF flag set should it later be translated back to IPv4).

Other than the special rules for handling fragments and path MTU
discovery, the actual translation of the packet header consists of a
simple translation as defined below. Note that ICMPv4 packets
require special handling in order to translate the content of ICMPv4
error messages and also to add the ICMPv6 pseudo-header checksum.

The translator SHOULD make sure that the packets belonging to the
same flow leave the translator in the same order in which they
arrived.

4.1. Translating IPv4 Headers into IPv6 Headers

   If the DF flag is not set and the IPv4 packet will result in an IPv6
   packet larger than 1280 bytes, the packet SHOULD be fragmented so the
   resulting IPv6 packet (with Fragment header added to each fragment)
   will be less than or equal to 1280 bytes. For example, if the packet
   is fragmented prior to the translation, the IPv4 packets should be
   fragmented so that their length, excluding the IPv4 header, is at
   most 1232 bytes (1280 minus 40 for the IPv6 header and 8 for the
   Fragment header). The translator MAY provide a configuration
   function for the network administrator to adjust the threshold of the
   minimum IPv6 MTU to a value greater than 1280-byte if the real value
   of the minimum IPv6 MTU in the network is known to the administrator.
   The resulting fragments are then translated independently using the
   logic described below.

   If the DF bit is set and the MTU of the next-hop interface is less
   than the total length value of the IPv4 packet plus 20, the
   translator MUST send an ICMPv4 "Fragmentation Needed" error message
   to the IPv4 source address.

   If the DF bit is set and the packet is not a fragment (i.e., the MF
   flag is not set and the Fragment Offset is equal to zero) then the
   translator SHOULD NOT add a Fragment header to the resulting packet.
   The IPv6 header fields are set as follows:

   Version: 6

   Traffic Class: By default, copied from IP Type Of Service (TOS)
      octet. According to [RFC2474] the semantics of the bits are
      identical in IPv4 and IPv6. However, in some IPv4 environments
      these fields might be used with the old semantics of "Type Of
      Service and Precedence". An implementation of a translator SHOULD
      support an administratively-configurable option to ignore the IPv4
      TOS and always set the IPv6 traffic class (TC) to zero. In
      addition, if the translator is at an administrative boundary, the
      filtering and update considerations of [RFC2475] may be
      applicable.

   Flow Label: 0 (all zero bits)

   Payload Length: Total length value from IPv4 header, minus the size
      of the IPv4 header and IPv4 options, if present.

   Next Header: For ICMPv4 (1) changed to ICMPv6 (58), otherwise
      protocol field MUST be copied from IPv4 header.

   Hop Limit: The hop limit is derived from the TTL value in the IPv4
      header. Since the translator is a router, as part of forwarding
      the packet it needs to decrement either the IPv4 TTL (before the
      translation) or the IPv6 Hop Limit (after the translation). As
      part of decrementing the TTL or Hop Limit the translator (as any
      router) MUST check for zero and send the ICMPv4 "TTL Exceeded" or
      ICMPv6 "Hop Limit Exceeded" error.

   Source Address: The IPv4-converted address derived from the IPv4
      source address per [I-D.ietf-behave-address-format] Section 2.1.

      If the translator gets an illegal source address (e.g. 0.0.0.0,
      127.0.0.1, etc.), the translator SHOULD silently drop the packet
      (as discussed in Section 5.3.7 of [RFC1812]).

   Destination Address: In the stateless mode, which is to say that if
      the IPv4 destination address is within a range of configured IPv4
      stateless translation prefix, the IPv6 destination address is the
      IPv4-translatable address derived from the IPv4 destination
      address per [I-D.ietf-behave-address-format] Section 2.1. A
      workflow example of stateless translation is shown in Section 10
      of this document.

      In the stateful mode, which is to say that if the IPv4 destination
      address is not within the range of any configured IPv4 stateless
      translation prefix, the IPv6 destination address and corresponding
      transport-layer destination port are derived from the Binding
      Information Bases (BIBs) reflecting current session state in the
      translator as described in [I-D.ietf-behave-v6v4-xlate-stateful].

   If any IPv4 options are present in the IPv4 packet, the IPv4 options
   MUST be ignored and the packet translated normally; there is no
   attempt to translate the options. However, if an unexpired source
   route option is present then the packet MUST instead be discarded,
   and an ICMPv4 "Destination Unreachable/Source Route Failed" (Type
   3/Code 5) error message SHOULD be returned to the sender.

   If there is a need to add a Fragment header (the DF bit is not set or
   the packet is a fragment) the header fields are set as above with the
   following exceptions:

   IPv6 fields:

Payload Length: Total length value from IPv4 header, plus 8 for
   the fragment header, minus the size of the IPv4 header and IPv4
   options, if present.

Next Header: Fragment header (44).

Fragment header fields:

Next Header: For ICMPv4 (1) changed to ICMPv6 (58), otherwise
   protocol field MUST be copied from IPv4 header.

Fragment Offset: Fragment Offset copied from the IPv4 header.

M flag: More Fragments bit copied from the IPv4 header.

Identification: The low-order 16 bits copied from the
   Identification field in the IPv4 header. The high-order 16
   bits set to zero.

4.2. Translating ICMPv4 Headers into ICMPv6 Headers

All ICMPv4 messages that are to be translated require that the ICMPv6
checksum field be calculated as part of the translation since ICMPv6,
unlike ICMPv4, has a pseudo-header checksum just like UDP and TCP.

In addition, all ICMPv4 packets MUST have the Type value translated
and, for ICMPv4 error messages, the included IP header also MUST be
translated.

The actions needed to translate various ICMPv4 messages are as
follows:

ICMPv4 query messages:

Echo and Echo Reply (Type 8 and Type 0): Adjust the Type values
   to 128 and 129, respectively, and adjust the ICMP checksum both
   to take the type change into account and to include the ICMPv6
   pseudo-header.

Information Request/Reply (Type 15 and Type 16): Obsoleted in
   ICMPv6. Silently drop.

Timestamp and Timestamp Reply (Type 13 and Type 14): Obsoleted in
   ICMPv6. Silently drop.

Address Mask Request/Reply (Type 17 and Type 18): Obsoleted in
   ICMPv6. Silently drop.

ICMP Router Advertisement (Type 9): Single hop message. Silently
   drop.

ICMP Router Solicitation (Type 10): Single hop message. Silently
   drop.

Unknown ICMPv4 types: Silently drop.

IGMP messages: While the MLD messages [RFC2710][RFC3590][RFC3810]
   are the logical IPv6 counterparts for the IPv4 IGMP messages
   all the "normal" IGMP messages are single-hop messages and
   SHOULD be silently dropped by the translator. Other IGMP
   messages might be used by multicast routing protocols and,
   since it would be a configuration error to try to have router
   adjacencies across IP/ICMP translators those packets SHOULD
   also be silently dropped.

 ICMPv4 error messages:

Destination Unreachable (Type 3): Translate the Code field as
   described below, set the Type field to 1, and adjust the
   ICMP checksum both to take the type/code change into account
   and to include the ICMPv6 pseudo-header.

   Translate the Code field as follows:

   Code 0, 1 (Net, host unreachable): Set Code value to 0 (no
      route to destination).

   Code 2 (Protocol unreachable): Translate to an ICMPv6
      Parameter Problem (Type 4, Code value 1) and make the
      Pointer point to the IPv6 Next Header field.

   Code 3 (Port unreachable): Set Code value to 4 (port
      unreachable).

   Code 4 (Fragmentation needed and DF set): Translate to an
      ICMPv6 Packet Too Big message (Type 2) with Code value
      set to 0. The MTU field MUST be adjusted for the
      difference between the IPv4 and IPv6 header sizes, i.e.
      minimum(advertised MTU+20, MTU_of_IPv6_nexthop,
      (MTU_of_IPv4_nexthop)+20). Note that if the IPv4 router
      set the MTU field to zero, i.e., the router does not
      implement [RFC1191], then the translator MUST use the
      plateau values specified in [RFC1191] to determine a

likely path MTU and include that path MTU in the ICMPv6
packet. (Use the greatest plateau value that is less
than the returned Total Length field.)

See also the requirements in Section 6.

Code 5 (Source route failed): Set Code value to 0 (No route
to destination). Note that this error is unlikely since
source routes are not translated.

Code 6, 7, 8: Set Code value to 0 (No route to
destination).

Code 9, 10 (Communication with destination host
administratively prohibited): Set Code value to 1
(Communication with destination administratively
prohibited)

Code 11, 12: Set Code value to 0 (no route to destination).

Code 13 (Communication Administratively Prohibited): Set
Code value to 1 (Communication with destination
administratively prohibited).

Code 14 (Host Precedence Violation): Silently drop.

Code 15 (Precedence cutoff in effect): Set Code value to 1
(Communication with destination administratively
prohibited).

Other Code values: Silently drop.

Redirect (Type 5): Single hop message. Silently drop.

Alternative Host Address (Type 6): Silently drop.

Source Quench (Type 4): Obsoleted in ICMPv6. Silently drop.

Time Exceeded (Type 11): Set the Type field to 3, and adjust
the ICMP checksum both to take the type change into account
and to include the ICMPv6 pseudo-header. The Code field is
unchanged.

Parameter Problem (Type 12): Set the Type field to 4, and
adjust the ICMP checksum both to take the type/code change
into account and to include the ICMPv6 pseudo-header.

Translate the Code field as follows:

Code 0 (Pointer indicates the error): Set the Code value to
0 (Erroneous header field encountered) and update the
pointer as defined in Figure 3 (If the Original IPv4
Pointer Value is not listed or the Translated IPv6
Pointer Value is listed as "n/a", silently drop the
packet).

Code 1 (Missing a required option): Silently drop

Code 2 (Bad length): Set the Code value to 0 (Erroneous
header field encountered) and update the pointer as
defined in Figure 3 (If the Original IPv4 Pointer Value
is not listed or the Translated IPv6 Pointer Value is
listed as "n/a", silently drop the packet).

Other Code values: Silently drop

Unknown ICMPv4 types: Silently drop.

Original IPv4 Pointer Value	Translated IPv6 Pointer Value
0	Version/IHL	0	Version/Traffic Class
1	Type Of Service	1	Traffic Class/Flow Label
2,3	Total Length	4	Payload Length
4,5	Identification	n/a	
6	Flags/Fragment Offset	n/a	
7	Fragment Offset	n/a	
8	Time to Live	7	Hop Limit
9	Protocol	6	Next Header
10,11	Header Checksum	n/a	
12-15	Source Address	8	Source Address
16-19	Destination Address	24	Destination Address

Figure 3: Pointer value for translating from IPv4 to IPv6

ICMP Error Payload: If the received ICMPv4 packet contains an
ICMPv4 Extension [RFC4884], the translation of the ICMPv4
packet will cause the ICMPv6 packet to change length. When
this occurs, the ICMPv6 Extension length attribute MUST be
adjusted accordingly (e.g., longer due to the translation
from IPv4 to IPv6). If the ICMPv4 Extension exceeds the
maximum size of an ICMPv6 message on the outgoing interface,
the ICMPv4 extension SHOULD be simply truncated. For
extensions not defined in [RFC4884], the translator passes

the extensions as opaque bit strings and those containing
IPv4 address literals will not have those addresses
translated to IPv6 address literals; this may cause problems
with processing of those ICMP extensions.

4.3. Translating ICMPv4 Error Messages into ICMPv6

There are some differences between the ICMPv4 and the ICMPv6 error
message formats as detailed above. The ICMP error messages
containing the packet in error MUST be translated just like a normal
IP packet. If the translation of this "packet in error" changes the
length of the datagram, the Total Length field in the outer IPv6
header MUST be updated.

```
       +-------------+                    +-------------+
       |    IPv4     |                    |    IPv6     |
       |   Header    |                    |   Header    |
       +-------------+                    +-------------+
       |   ICMPv4    |                    |   ICMPv6    |
       |   Header    |                    |   Header    |
       +-------------+                    +-------------+
       |    IPv4     |      ===>          |    IPv6     |
       |   Header    |                    |   Header    |
       +-------------+                    +-------------+
       |   Partial   |                    |   Partial   |
       |  Transport  |                    |  Transport  |
       |    Layer    |                    |    Layer    |
       |   Header    |                    |   Header    |
       +-------------+                    +-------------+
```
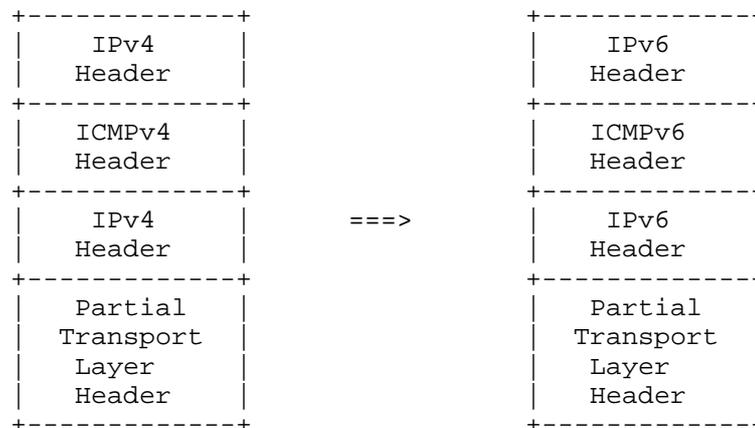
Figure 4: IPv4-to-IPv6 ICMP Error Translation

The translation of the inner IP header can be done by invoking the
function that translated the outer IP headers. This process MUST
stop at the first embedded header and drop the packet if it contains
more.

4.4. Generation of ICMPv4 Error Message

If the IPv4 packet is discarded, then the translator SHOULD be able
to send back an ICMPv4 error message to the original sender of the
packet, unless the discarded packet is itself an ICMPv4 message. The
ICMPv4 message, if sent, has a Type value of 3 (Destination
Unreachable) and a Code value of 13 (Communication Administratively
Prohibited), unless otherwise specified in this document or in
[I-D.ietf-behave-v6v4-xlate-stateful]. The translator SHOULD allow
an administrator to configure whether the ICMPv4 error messages are
sent, rate-limited, or not sent.

4.5. Transport-layer Header Translation

   If the address translation algorithm is not checksum neutral (Section
   4.1 of [I-D.ietf-behave-address-format]), the recalculation and
   updating of the transport-layer headers which contain pseudo headers
   needs to be performed. Translators MUST do this for TCP and ICMP
   packets and for UDP packets that contain a UDP checksum (i.e. the UDP
   checksum field is not zero).

   For UDP packets that do not contain a UDP checksum (i.e. the UDP
   checksum field is zero), the translator SHOULD provide a
   configuration function to allow:

   1. Dropping the packet and generating a system management event
       specifying at least the IP addresses and port numbers of the
       packet.

   2. Calculating an IPv6 checksum and forward the packet (which has
       performance implications).

       A stateless translator cannot compute the UDP checksum of
       fragmented packets, so when a stateless translator receives the
       first fragment of a fragmented UDP IPv4 packet and the checksum
       field is zero, the translator SHOULD drop the packet and generate
       a system management event specifying at least the IP addresses
       and port numbers in the packet.

       For stateful translator, the handling of fragmented UDP IPv4
       packets with a zero checksum is discussed in
       [I-D.ietf-behave-v6v4-xlate-stateful]), Section 3.1.

   Other transport protocols (e.g., DCCP) are OPTIONAL to support. In
   order to ease debugging and troubleshooting, translators MUST forward
   all transport protocols as described in the "Next Header" step of
   Section 4.1.

4.6. Knowing When to Translate

   If the IP/ICMP translator also provides normal forwarding function,
   and the destination IPv4 address is reachable by a more specific
   route without translation, the translator MUST forward it without
   translating it. Otherwise, when an IP/ICMP translator receives an
   IPv4 datagram addressed to an IPv4 destination representing a host in
   the IPv6 domain, the packet MUST be translated to IPv6.

5. Translating from IPv6 to IPv4

   When an IP/ICMP translator receives an IPv6 datagram addressed to a
   destination towards the IPv4 domain, it translates the IPv6 header of
   the received IPv6 packet into an IPv4 header. The original IPv6
   header on the packet is removed and replaced by an IPv4 header.
   Since the ICMPv6 [RFC4443], TCP [RFC0793], UDP [RFC0768] and DCCP
   [RFC4340] headers contain checksums that cover the IP header, if the
   address mapping algorithm is not checksum-neutral, the checksum MUST
   be evaluated before translation and the ICMP and transport-layer
   headers MUST be updated. The data portion of the packet is left
   unchanged. The IP/ICMP translator then forwards the packet based on
   the IPv4 destination address.

```
         +-------------+                   +-------------+
         |    IPv6     |                   |    IPv4     |
         |   Header    |                   |   Header    |
         +-------------+                   +-------------+
         |  Fragment   |                   |  Transport  |
         |   Header    |      ===>         |   Layer     |
         |(if present) |                   |   Header    |
         +-------------+                   +-------------+
         |  Transport  |                   |             |
         |   Layer     |                   ~    Data     ~
         |   Header    |                   |             |
         +-------------+                   +-------------+
         |             |
         ~    Data     ~
         |             |
         +-------------+
```
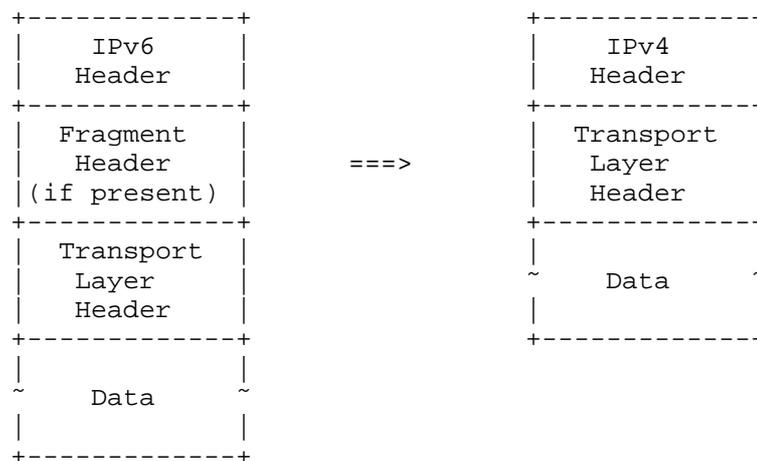
                Figure 5: IPv6-to-IPv4 Translation

   There are some differences between IPv6 and IPv4 in the area of
   fragmentation and the minimum link MTU that affect the translation.
   An IPv6 link has to have an MTU of 1280 bytes or greater. The
   corresponding limit for IPv4 is 68 bytes. Path MTU Discovery across
   a translator relies on ICMP Packet Too Big messages being received
   and processed by IPv6 hosts, including an ICMP Packet Too Big that
   indicates the MTU is less than the IPv6 minimum MTU. This
   requirement is described in Section 5 of [RFC2460] (for IPv6's 1280
   octet minimum MTU) and Section 5 of [RFC1883] (for IPv6's previous
   576 octet minimum MTU).

   In an environment where an ICMPv4 Packet Too Big message is
   translated to an ICMPv6 Packet Too Big messages, and the ICMPv6
   Packet Too Big message is successfully delivered to and correctly
   processed by the IPv6 hosts (e.g., a network owned/operated by the

same entity that owns/operates the translator), the translator can
rely on IPv6 hosts sending subsequent packets to the same IPv6
destination with IPv6 fragment headers. In such an environment, when
the translator receives an IPv6 packet with a fragmentation header,
the translator SHOULD generate the IPv4 packet with a cleared Don't
Fragment bit, and with its identification value from the IPv6
fragment header, for all of the IPv6 fragments (MF=0 or MF=1).

In an environment where an ICMPv4 Packet Too Big message are filtered
(by a network firewall or by the host itself) or not correctly
processed by the IPv6 hosts, the IPv6 host will never generate an
IPv6 packet with the IPv6 fragment header. In such an environment,
the translator SHOULD set the IPv4 Don't Fragment bit. While setting
the Don't Fragment bit may create PMTUD black holes [RFC2923] if
there are IPv4 links smaller than 1260 octets, this is considered
safer than causing IPv4 reassembly errors [RFC4963].

Other than the special rules for handling fragments and path MTU
discovery, the actual translation of the packet header consists of a
simple translation as defined below. Note that ICMPv6 packets
require special handling in order to translate the contents of ICMPv6
error messages and also to remove the ICMPv6 pseudo-header checksum.

The translator SHOULD make sure that the packets belonging to the
same flow leave the translator in the same order in which they
arrived.

5.1. Translating IPv6 Headers into IPv4 Headers

If there is no IPv6 Fragment header, the IPv4 header fields are set
as follows:

Version: 4

Internet Header Length: 5 (no IPv4 options)

Type of Service (TOS) Octet: By default, copied from the IPv6
   Traffic Class (all 8 bits). According to [RFC2474] the semantics
   of the bits are identical in IPv4 and IPv6. However, in some IPv4
   environments, these bits might be used with the old semantics of
   "Type Of Service and Precedence". An implementation of a
   translator SHOULD provide the ability to ignore the IPv6 traffic
   class and always set the IPv4 TOS Octet to a specified value. In
   addition, if the translator is at an administrative boundary, the
   filtering and update considerations of [RFC2475] may be
   applicable.

Total Length: Payload length value from IPv6 header, plus the size
   of the IPv4 header.

Identification: All zero. In order to avoid black holes caused by
   ICMPv4 filtering or non [RFC2460] compatible IPv6 hosts (a
   workaround discussed in Section 6), the translator MAY provide a
   function such as if the packet size is equal to or smaller than
   1280 bytes and greater than 88 bytes, generate the identification
   value. The translator SHOULD provide a method for operators to
   enable or disable this function.

Flags: The More Fragments flag is set to zero. The Don't Fragments
   flag is set to one. In order to avoid black holes caused by
   ICMPv4 filtering or non [RFC2460] compatible IPv6 hosts (a
   workaround discussed in Section 6), the translator MAY provide a
   function such as if the packet size is equal to or smaller than
   1280 bytes and greater than 88 bytes, the Don't Fragments (DF)
   flag is set to zero, otherwise the Don't Fragments (DF) flag is
   set to one. The translator SHOULD provide a method for operators
   to enable or disable this function.

Fragment Offset: All zeros.

Time to Live: Time to Live is derived from Hop Limit value in IPv6
   header. Since the translator is a router, as part of forwarding
   the packet it needs to decrement either the IPv6 Hop Limit (before
   the translation) or the IPv4 TTL (after the translation). As part
   of decrementing the TTL or Hop Limit the translator (as any
   router) MUST check for zero and send the ICMPv4 "TTL Exceeded" or
   ICMPv6 "Hop Limit Exceeded" error.

Protocol: The IPv6-Frag (44) header is handled as discussed in
   Section 5.1.1. ICMPv6 (58) is changed to ICMPv4 (1), and the
   payload is translated as discussed in Section 5.2. The IPv6
   headers HOPOPT (0), IPv6-Route (43), and IPv6-Opts (60) are
   skipped over during processing as they have no meaning in IPv4.
   For the first 'next header' that does not match one of the cases
   above, its next header value (which contains the transport
   protocol number) is copied to the protocol field in the IPv4
   header. This means that all transport protocols are translated.

Note: Some translated protocols will fail at the receiver for
   various reasons: some are known to fail when translated (e.g.,
   IPsec AH (51)), and others will fail checksum validation if the
   address translation is not checksum neutral
   [I-D.ietf-behave-address-format] and the translator does not
   update the transport protocol's checksum (because the
   translator doesn't support recalculating the checksum for that

      transport protocol, see Section 5.5).

   Header Checksum: Computed once the IPv4 header has been created.

   Source Address: In the stateless mode, which is to say that if the
      IPv6 source address is within the range of a configured IPv6
      translation prefix, the IPv4 source address is derived from the
      IPv6 source address per [I-D.ietf-behave-address-format] Section
      2.1. Note that the original IPv6 source address is an IPv4-
      translatable address. A workflow example of stateless translation
      is shown in Appendix of this document. If the translator only
      supports stateless mode and if the IPv6 source address is not
      within the range of configured IPv6 prefix(es), the translator
      SHOULD drop the packet and respond with an ICMPv6 Type=1, Code=5
      (Destination Unreachable, Source address failed ingress/egress
      policy).

      In the stateful mode, which is to say that if the IPv6 source
      address is not within the range of any configured IPv6 stateless
      translation prefix, the IPv4 source address and transport-layer
      source port corresponding to the IPv4-related IPv6 source address
      and source port are derived from the Binding Information Bases
      (BIBs) as described in [I-D.ietf-behave-v6v4-xlate-stateful].

      In stateless and stateful modes, if the translator gets an illegal
      source address (e.g. ::1, etc.), the translator SHOULD silently
      drop the packet.

   Destination Address: The IPv4 destination address is derived from
      the IPv6 destination address of the datagram being translated per
      [I-D.ietf-behave-address-format] Section 2.1. Note that the
      original IPv6 destination address is an IPv4-converted address.

   If a Routing header with a non-zero Segments Left field is present
   then the packet MUST NOT be translated, and an ICMPv6 "parameter
   problem/erroneous header field encountered" (Type 4/Code 0) error
   message, with the Pointer field indicating the first byte of the
   Segments Left field, SHOULD be returned to the sender.

5.1.1. IPv6 Fragment Processing

   If the IPv6 packet contains a Fragment header, the header fields are
   set as above with the following exceptions:

   Total Length: Payload length value from IPv6 header, minus 8 for the
      Fragment header, plus the size of the IPv4 header.

   Identification: Copied from the low-order 16-bits in the
      Identification field in the Fragment header.

   Flags: The IPv4 More Fragments (MF) flag is copied from the M flag
      in the IPv6 Fragment header. The IPv4 Don't Fragments (DF) flag
      is cleared (set to zero) allowing this packet to be further
      fragmented by IPv4 routers.

   Fragment Offset: Copied from the Fragment Offset field of the IPv6
      Fragment header.

   Protocol: For ICMPv6 (58) changed to ICMPv4 (1), otherwise skip
      extension headers, Next Header field copied from the last IPv6
      header.

   If a translated packet with DF set to 1 will be larger than the MTU
   of the next-hop interface, then the translator MUST drop the packet
   and send the ICMPv6 "Packet Too Big" (Type 2/Code 0) error message to
   the IPv6 host with an adjusted MTU in the ICMPv6 message.

5.2. Translating ICMPv6 Headers into ICMPv4 Headers

   If a non-checksum neutral translation address is being used, ICMPv6
   messages MUST have their ICMPv4 checksum field be updated as part of
   the translation since ICMPv6 (unlike ICMPv4) includes a pseudo-header
   in the checksum just like UDP and TCP.

   In addition all ICMP packets MUST have the Type value translated and,
   for ICMP error messages, the included IP header also MUST be
   translated. Note that the IPv6 addresses in the IPv6 header may not
   be IPv4-translatable addresses and there will be no corresponding
   IPv4 addresses representing this IPv6 address. In this case, the
   translator can do stateful translation. A mechanism by which the
   translator can instead do stateless translation of this address is
   left for future work.

   The actions needed to translate various ICMPv6 messages are:

   ICMPv6 informational messages:

      Echo Request and Echo Reply (Type 128 and 129): Adjust the Type
         values to 8 and 0, respectively, and adjust the ICMP checksum
         both to take the type change into account and to exclude the
         ICMPv6 pseudo-header.

MLD Multicast Listener Query/Report/Done (Type 130, 131, 132):
   Single hop message. Silently drop.

Neighbor Discover messages (Type 133 through 137): Single hop
   message. Silently drop.

Unknown informational messages: Silently drop.

ICMPv6 error messages:

Destination Unreachable (Type 1) Set the Type field to 3, and
   adjust the ICMP checksum both to take the type/code change into
   account and to exclude the ICMPv6 pseudo-header.

   Translate the Code field as follows:

   Code 0 (no route to destination): Set Code value to 1 (Host
      unreachable).

   Code 1 (Communication with destination administratively
   prohibited): Set Code value to 10 (Communication with
      destination host administratively prohibited).

   Code 2 (Beyond scope of source address): Set Code value to 1
      (Host unreachable). Note that this error is very unlikely
      since an IPv4-translatable source address is typically
      considered to have global scope.

   Code 3 (Address unreachable): Set Code value to 1 (Host
      unreachable).

   Code 4 (Port unreachable): Set Code value to 3 (Port
      unreachable).

   Other Code values: Silently drop.

Packet Too Big (Type 2): Translate to an ICMPv4 Destination
   Unreachable (Type 3) with Code value equal to 4, and adjust the
   ICMPv4 checksum both to take the type change into account and
   to exclude the ICMPv6 pseudo-header. The MTU field MUST be
   adjusted for the difference between the IPv4 and IPv6 header
   sizes taking into account whether or not the packet in error
   includes a Fragment header, i.e. minimum(advertised MTU-20,
   MTU_of_IPv4_nexthop, (MTU_of_IPv6_nexthop)-20).

See also the requirements in Section 6.

Time Exceeded (Type 3): Set the Type value to 11, and adjust the
    ICMPv4 checksum both to take the type change into account and
    to exclude the ICMPv6 pseudo-header. The Code field is
    unchanged.

Parameter Problem (Type 4): Translate the Type and Code field as
    follows, and adjust the ICMPv4 checksum both to take the type/
    code change into account and to exclude the ICMPv6 pseudo-
    header.

    Translate the Code field as follows:

    Code 0 (Erroneous header field encountered): Set Type 12, Code
        0 and update the pointer as defined in Figure 6 (If the
        Original IPv6 Pointer Value is not listed or the Translated
        IPv4 Pointer Value is listed as "n/a", silently drop the
        packet).

    Code 1 (Unrecognized Next Header type encountered): Translate
        this to an ICMPv4 protocol unreachable (Type 3, Code 2).

    Code 2 (Unrecognized IPv6 option encountered): Silently drop.

Unknown error messages: Silently drop.

Original IPv6 Pointer Value	Translated IPv4 Pointer Value
0	Version/Traffic Class	0	Version/IHL, Type Of Ser
1	Traffic Class/Flow Label	1	Type Of Service
2,3	Flow Label	n/a	
4,5	Payload Length	2	Total Length
6	Next Header	9	Protocol
7	Hop Limit	8	Time to Live
8-23	Source Address	12	Source Address
24-39	Destination Address	16	Destination Address

Figure 6: Pointer Value for translating from IPv6 to IPv4

ICMP Error Payload: If the received ICMPv6 packet contains an
   ICMPv6 Extension [RFC4884], the translation of the ICMPv6
   packet will cause the ICMPv4 packet to change length. When
   this occurs, the ICMPv6 Extension length attribute MUST be
   adjusted accordingly (e.g., shorter due to the translation from
   IPv6 to IPv4). For extensions not defined in [RFC4884], the
   translator passes the extensions as opaque bit strings and
   those containing IPv6 address literals will not have those
   addresses translated to IPv4 address literals; this may cause
   problems with processing of those ICMP extensions.

5.3. Translating ICMPv6 Error Messages into ICMPv4

   There are some differences between the ICMPv4 and the ICMPv6 error
   message formats as detailed above. The ICMP error messages
   containing the packet in error MUST be translated just like a normal
   IP packet. The translation of this "packet in error" is likely to
   change the length of the datagram thus the Total Length field in the
   outer IPv4 header MUST be updated.

```
        +-------------+                   +-------------+
        |   IPv6      |                   |   IPv4      |
        |   Header    |                   |   Header    |
        +-------------+                   +-------------+
        |   ICMPv6    |                   |   ICMPv4    |
        |   Header    |                   |   Header    |
        +-------------+                   +-------------+
        |   IPv6      |      ===>         |   IPv4      |
        |   Header    |                   |   Header    |
        +-------------+                   +-------------+
        |   Partial   |                   |   Partial   |
        |   Transport |                   |   Transport |
        |   Layer     |                   |   Layer     |
        |   Header    |                   |   Header    |
        +-------------+                   +-------------+
```
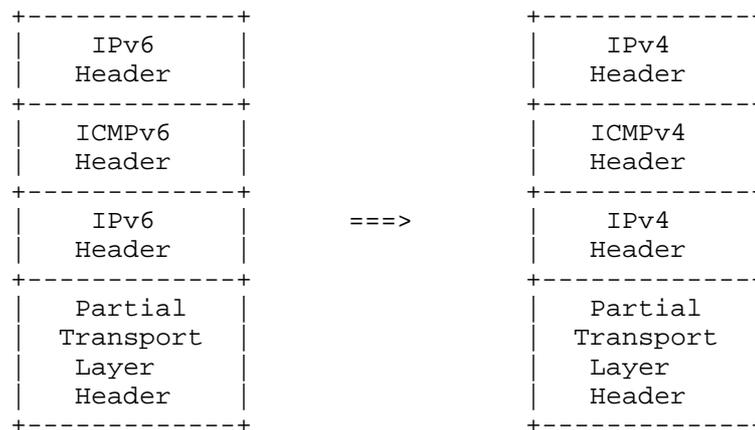
              Figure 7: IPv6-to-IPv4 ICMP Error Translation

   The translation of the inner IP header can be done by invoking the
   function that translated the outer IP headers. This process MUST
   stop at first embedded header and drop the packet if it contains
   more. Note that the IPv6 addresses in the IPv6 header may not be
   IPv4-translatable addresses and there will be no corresponding IPv4
   addresses. In this case, the translator can do stateful translation.
   A mechanism by which the translator can instead do stateless
   translation is left for future work.

5.4. Generation of ICMPv6 Error Message

   If the IPv6 packet is discarded, then the translator SHOULD send back
   an ICMPv6 error message to the original sender of the packet, unless
   the discarded packet is itself an ICMPv6 message.

   If the ICMPv6 error message is being sent because the IPv6 source
   address is not an IPv4-translatable address and the translator is
   stateless, the ICMPv6 message, if sent, MUST have a Type value of 1
   and Code value of 5 (Source address failed ingress/egress policy).
   In other cases, the ICMPv6 message MUST have a Type value of 1
   (Destination Unreachable) and a Code value of 1 (Communication with
   destination administratively prohibited), unless otherwise specified
   in this document or [I-D.ietf-behave-v6v4-xlate-stateful]. The
   translator SHOULD allow an administrator to configure whether the
   ICMPv6 error messages are sent, rate-limited, or not sent.

5.5. Transport-layer Header Translation

   If the address translation algorithm is not checksum neutral (Section
   4.1 of [I-D.ietf-behave-address-format]), the recalculation and
   updating of the transport-layer headers which contain pseudo headers
   need to be performed. Translators MUST do this for TCP, UDP and
   ICMP.

   Other transport protocols (e.g., DCCP) are OPTIONAL to support. In
   order to ease debugging and troubleshooting, translators MUST forward
   all transport protocols as described in the "Protocol" step of
   Section 5.1.

5.6. Knowing When to Translate

   If the IP/ICMP translator also provides a normal forwarding function,
   and the destination address is reachable by a more specific route
   without translation, the router MUST forward it without translating
   it. When an IP/ICMP translator receives an IPv6 datagram addressed
   to an IPv6 address representing a host in the IPv4 domain, the IPv6
   packet MUST be translated to IPv4.

6. Special Considerations for ICMPv6 Packet Too Big

   Two recent studies analyzed the behavior of IPv6-capable web servers
   on the Internet and found that approximately 95% responded as
   expected to an IPv6 Packet Too Big that indicated MTU=1280, but only
   43% responded as expected to an IPv6 Packet Too Big that indicated an
   MTU < 1280. It is believed firewalls violating Section 4.3.1 of
   [RFC4890] are at fault. These failures will both cause Path MTU

Discovery (PMTUD) black holes [RFC2923]. Unfortunately the
translator cannot improve the failure rate of the first case (MTU =
1280), but the translator can improve the failure rate of the second
case (MTU < 1280). There are two approaches to resolving the problem
with sending ICMPv6 messages indicating an MTU < 1280. It SHOULD be
possible to configure a translator for either of the two approaches.

The first approach is to constrain the deployment of the IPv6/IPv4
translator by observing that four of the scenarios intended for
stateless IPv6/IPv4 translators do not have IPv6 hosts on the
Internet (Scenarios 1, 2, 5 and 6 described in
[I-D.ietf-behave-v6v4-framework], which refer to "An IPv6 network").
In these scenarios IPv6 hosts, IPv6 host-based firewalls, and IPv6
network firewalls can be administered in compliance with Section
4.3.1 of [RFC4890] and therefore avoid the problem witnessed with
IPv6 hosts on the Internet.

The second approach is necessary if the translator has IPv6 hosts,
IPv6 host-based firewalls, or IPv6 network firewalls that do not (or
cannot) comply with Section 5 of [RFC2460] -- such as IPv6 hosts on
the Internet. This approach requires the translator to do the
following:

1. in the IPv4 to IPv6 direction: if the MTU value of ICMPv4 Packet
    Too Big messages is less than 1280, change it to 1280. This is
    intended to cause the IPv6 host and IPv6 firewall to process the
    ICMP PTB message and generate subsequent packets to this
    destination with an IPv6 fragmentation header.

    Note: Based on recent studies, this is effective for 95% of IPv6
    hosts on the Internet.

2. in the IPv6 to IPv4 direction:

    A. if there is a Fragment header in the IPv6 packet, the last 16
        bits of its value MUST be used for the IPv4 identification
        value.

    B. if there is no Fragment header in the IPv6 packet:

        a. if the packet is less than or equal to 1280 bytes:

            - the translator SHOULD set DF to 0 and generate an IPv4
               identification value.

            - To avoid the problems described in [RFC4963], it is
               RECOMMENDED the translator maintain 3-tuple state for

generating the IPv4 identification value.

   b. if the packet is greater than 1280 bytes, the translator
       SHOULD set the IPv4 DF bit to 1.

7. IANA Considerations

   This memo adds no new IANA considerations.

   Note to RFC Editor: This section will have served its purpose if it
   correctly tells IANA that no new assignments or registries are
   required, or if those assignments or registries are created during
   the RFC publication process. From the author's perspective, it may
   therefore be removed upon publication as an RFC at the RFC Editor's
   discretion.

8. Security Considerations

   The use of stateless IP/ICMP translators does not introduce any new
   security issues beyond the security issues that are already present
   in the IPv4 and IPv6 protocols and in the routing protocols that are
   used to make the packets reach the translator.

   There are potential issues that might arise by deriving an IPv4
   address from an IPv6 address - particularly addresses like broadcast
   or loopback addresses and the non IPv4-translatable IPv6 addresses,
   etc. The [I-D.ietf-behave-address-format] addresses these issues.

   As with network address translation of IPv4 to IPv4, the IPsec
   Authentication Header [RFC4302] cannot be used across an IPv6 to IPv4
   translator.

   As with network address translation of IPv4 to IPv4, packets with
   tunnel mode ESP can be translated since tunnel mode ESP does not
   depend on header fields prior to the ESP header. Similarly,
   transport mode ESP will fail with IPv6 to IPv4 translation unless
   checksum neutral addresses are used. In both cases, the IPsec ESP
   endpoints will normally detect the presence of the translator and
   encapsulate ESP in UDP packets [RFC3948].

9. Acknowledgements

   This is under development by a large group of people. Those who have
   posted to the list during the discussion include Alexey Melnikov,
   Andrew Sullivan, Andrew Yourtchenko, Brian Carpenter, Dan Wing, Dave

Thaler, David Harrington, Ed Jankiewicz, Hiroshi Miyata, Iljitsch van
Beijnum, Jari Arkko, Jerry Huang, John Schnizlein, Jouni Korhonen,
Kentaro Ebisawa, Kevin Yin, Magnus Westerlund, Marcelo Bagnulo Braun,
Margaret Wasserman, Masahito Endo, Phil Roberts, Philip Matthews,
Reinaldo Penno, Remi Denis-Courmont, Remi Despres, Sean Turner,
Senthil Sivakumar, Simon Perreault, Stewart Bryant, Tim Polk, Tero
Kivinen and Zen Cao.

10. Appendix: Stateless translation workflow example

   A stateless translation workflow example is depicted in the following
   figure. The documentation address blocks 2001:db8::/32 [RFC3849],
   192.0.2.0/24 and 198.51.100.0/24 [RFC5737] are used in this example.

```
      +--------------+               +--------------+
      | IPv4 network |               | IPv6 network |
      |              |    +-------+   |              |
      |    +----+    |----| XLAT  |---- |    +----+    |
      |    | H4 |-----|    +-------+   |--| H6 |    |
      |    +----+    |               |    +----+    |
      +--------------+               +--------------+
```
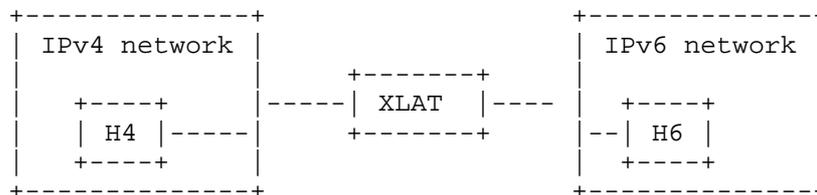
                              Figure 8

   A translator (XLAT) connects the IPv6 network to the IPv4 network.
   This XLAT uses the Network Specific Prefix (NSP) 2001:db8:100::/40
   defined in [I-D.ietf-behave-address-format] to represent IPv4
   addresses in the IPv6 address space (IPv4-converted addresses) and to
   represent IPv6 addresses (IPv4-translatable addresses) in the IPv4
   address space. In this example, 192.0.2.0/24 is the IPv4 block of
   the corresponding IPv4-translatable addresses.

   Based on the address mapping rule, the IPv6 node H6 has an IPv4-
   translatable IPv6 address 2001:db8:1c0:2:21:: (address mapping from
   192.0.2.33). The IPv4 node H4 has IPv4 address 198.51.100.2.

   The IPv6 routing is configured in such a way that the IPv6 packets
   addressed to a destination address in 2001:db8:100::/40 are routed to
   the IPv6 interface of the XLAT.

   The IPv4 routing is configured in such a way that the IPv4 packets
   addressed to a destination address in 192.0.2.0/24 are routed to the
   IPv4 interface of the XLAT.

10.1. H6 establishes communication with H4

   The steps by which H6 establishes communication with H4 are:

   1. H6 performs the destination address mapping, so the IPv4-
       converted address 2001:db8:1c6:3364:200:: is formed from
       198.51.100.2 based on the address mapping algorithm
       [I-D.ietf-behave-address-format].

   2. H6 sends a packet to H4. The packet is sent from a source
       address 2001:db8:1c0:2:21:: to a destination address
       2001:db8:1c6:3364:200::.

   3. The packet is routed to the IPv6 interface of the XLAT (since
       IPv6 routing is configured that way).

   4. The XLAT receives the packet and performs the following actions:

       * The XLAT translates the IPv6 header into an IPv4 header using
          the IP/ICMP Translation Algorithm defined in this document.

       * The XLAT includes 192.0.2.33 as source address in the packet
          and 198.51.100.2 as destination address in the packet. Note
          that 192.0.2.33 and 198.51.100.2 are extracted directly from
          the source IPv6 address 2001:db8:1c0:2:21:: (IPv4-translatable
          address) and destination IPv6 address 2001:db8:1c6:3364:200::
          (IPv4-converted address) of the received IPv6 packet that is
          being translated.

   5. The XLAT sends the translated packet out its IPv4 interface and
       the packet arrives at H4.

   6. H4 node responds by sending a packet with destination address
       192.0.2.33 and source address 198.51.100.2.

   7. The packet is routed to the IPv4 interface of the XLAT (since
       IPv4 routing is configured that way). The XLAT performs the
       following operations:

       * The XLAT translates the IPv4 header into an IPv6 header using
          the IP/ICMP Translation Algorithm defined in this document.

       * The XLAT includes 2001:db8:1c0:2:21:: as destination address
          in the packet and 2001:db8:1c6:3364:200:: as source address in
          the packet. Note that 2001:db8:1c0:2:21:: and
          2001:db8:1c6:3364:200::
          are formed directly from the destination IPv4
          address 192.0.2.33 and source IPv4 address 198.51.100.2 of the
          received IPv4 packet that is being translated.

8. The translated packet is sent out the IPv6 interface to H6.

The packet exchange between H6 and H4 continues until the session is finished.

10.2. H4 establishes communication with H6

The steps by which H4 establishes communication with H6 are:

1. H4 performs the destination address mapping, so 192.0.2.33 is formed from IPv4-translatable address 2001:db8:1c0:2:21:: based on the address mapping algorithm [I-D.ietf-behave-address-format].

2. H4 sends a packet to H6. The packet is sent from a source address 198.51.100.2 to a destination address 192.0.2.33.

3. The packet is routed to the IPv4 interface of the XLAT (since IPv4 routing is configured that way).

4. The XLAT receives the packet and performs the following actions:

    * The XLAT translates the IPv4 header into an IPv6 header using the IP/ICMP Translation Algorithm defined in this document.

    * The XLAT includes 2001:db8:1c6:3364:200:: as source address in the packet and 2001:db8:1c0:2:21:: as destination address in the packet. Note that 2001:db8:1c6:3364:200:: (IPv4-converted address) and 2001:db8:1c0:2:21:: (IPv4-translatable address) are obtained directly from the source IPv4 address 198.51.100.2 and destination IPv4 address 192.0.2.33 of the received IPv4 packet that is being translated.

5. The XLAT sends the translated packet out its IPv6 interface and the packet arrives at H6.

6. H6 node responds by sending a packet with destination address 2001:db8:1c6:3364:200:: and source address 2001:db8:1c0:2:21::.

7. The packet is routed to the IPv6 interface of the XLAT (since IPv6 routing is configured that way). The XLAT performs the following operations:

    * The XLAT translates the IPv6 header into an IPv4 header using the IP/ICMP Translation Algorithm defined in this document.

    * The XLAT includes 198.51.100.2 as destination address in the packet and 192.0.2.33 as source address in the packet. Note

that 198.51.100.2 and 192.0.2.33 are formed directly from the
destination IPv6 address 2001:db8:1c6:3364:200:: and source
IPv6 address 2001:db8:1c0:2:21:: of the received IPv6 packet
that is being translated.

8. The translated packet is sent out the IPv4 interface to H4.

The packet exchange between H4 and H6 continues until the session
finished.

11. References

11.1. Normative References

[I-D.ietf-behave-address-format]
          Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X.
          Li, "IPv6 Addressing of IPv4/IPv6 Translators",
          draft-ietf-behave-address-format-10 (work in progress),
          August 2010.

[I-D.ietf-behave-v6v4-xlate-stateful]
          Bagnulo, M., Matthews, P., and I. Beijnum, "Stateful
          NAT64: Network Address and Protocol Translation from IPv6
          Clients to IPv4 Servers",
          draft-ietf-behave-v6v4-xlate-stateful-12 (work in
          progress), July 2010.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768,
          August 1980.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791,
          September 1981.

[RFC0792] Postel, J., "Internet Control Message Protocol", STD 5,
          RFC 792, September 1981.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7,
          RFC 793, September 1981.

[RFC1812] Baker, F., "Requirements for IP Version 4 Routers",
          RFC 1812, June 1995.

[RFC1883] Deering, S. and R. Hinden, "Internet Protocol, Version 6
          (IPv6) Specification", RFC 1883, December 1995.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
          Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6
              (IPv6) Specification", RFC 2460, December 1998.

   [RFC2765] Nordmark, E., "Stateless IP/ICMP Translation Algorithm
              (SIIT)", RFC 2765, February 2000.

   [RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M.
              Stenberg, "UDP Encapsulation of IPsec ESP Packets",
              RFC 3948, January 2005.

   [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing
              Architecture", RFC 4291, February 2006.

   [RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram
              Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

   [RFC4443] Conta, A., Deering, S., and M. Gupta, "Internet Control
              Message Protocol (ICMPv6) for the Internet Protocol
              Version 6 (IPv6) Specification", RFC 4443, March 2006.

   [RFC4884] Bonica, R., Gan, D., Tappan, D., and C. Pignataro,
              "Extended ICMP to Support Multi-Part Messages", RFC 4884,
              April 2007.

   [RFC5382] Guha, S., Biswas, K., Ford, B., Sivakumar, S., and P.
              Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142,
              RFC 5382, October 2008.

   [RFC5771] Cotton, M., Vegoda, L., and D. Meyer, "IANA Guidelines for
              IPv4 Multicast Address Assignments", BCP 51, RFC 5771,
              March 2010.

11.2. Informative References

   [I-D.ietf-behave-v6v4-framework]
              Baker, F., Li, X., Bao, C., and K. Yin, "Framework for
              IPv4/IPv6 Translation",
              draft-ietf-behave-v6v4-framework-10 (work in progress),
              August 2010.

   [RFC0879] Postel, J., "TCP maximum segment size and related topics",
              RFC 879, November 1983.

   [RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191,
              November 1990.

   [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black,
              "Definition of the Differentiated Services Field (DS

Field) in the IPv4 and IPv6 Headers", RFC 2474,
December 1998.

[RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z.,
and W. Weiss, "An Architecture for Differentiated
Services", RFC 2475, December 1998.

[RFC2710] Deering, S., Fenner, W., and B. Haberman, "Multicast
Listener Discovery (MLD) for IPv6", RFC 2710,
October 1999.

[RFC2766] Tsirtsis, G. and P. Srisuresh, "Network Address
Translation - Protocol Translation (NAT-PT)", RFC 2766,
February 2000.

[RFC2923] Lahey, K., "TCP Problems with Path MTU Discovery",
RFC 2923, September 2000.

[RFC3307] Haberman, B., "Allocation Guidelines for IPv6 Multicast
Addresses", RFC 3307, August 2002.

[RFC3590] Haberman, B., "Source Address Selection for the Multicast
Listener Discovery (MLD) Protocol", RFC 3590,
September 2003.

[RFC3810] Vida, R. and L. Costa, "Multicast Listener Discovery
Version 2 (MLDv2) for IPv6", RFC 3810, June 2004.

[RFC3849] Huston, G., Lord, A., and P. Smith, "IPv6 Address Prefix
Reserved for Documentation", RFC 3849, July 2004.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302,
December 2005.

[RFC4890] Davies, E. and J. Mohacsi, "Recommendations for Filtering
ICMPv6 Messages in Firewalls", RFC 4890, May 2007.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly
Errors at High Data Rates", RFC 4963, July 2007.

[RFC5737] Arkko, J., Cotton, M., and L. Vegoda, "IPv4 Address Blocks
Reserved for Documentation", RFC 5737, January 2010.

Authors' Addresses

    Xing Li
    CERNET Center/Tsinghua University
    Room 225, Main Building, Tsinghua University
    Beijing,   100084
    China

    Phone: +86 10-62785983
    Email: xing@cernet.edu.cn

    Congxiao Bao
    CERNET Center/Tsinghua University
    Room 225, Main Building, Tsinghua University
    Beijing,   100084
    China

    Phone: +86 10-62785983
    Email: congxiao@cernet.edu.cn

    Fred Baker
    Cisco Systems
    Santa Barbara, California 93117
    USA

    Phone: +1-408-526-4257
    Email: fred@cisco.com

BEHAVE WG   M. Bagnulo
Internet-Draft   UC3M
Intended status: Standards Track   P. Matthews
Expires: January 11, 2011   Alcatel-Lucent
                                                           I. van Beijnum
                                                           IMDEA Networks
                                                            July 10, 2010

        Stateful NAT64: Network Address and Protocol Translation from IPv6
                           Clients to IPv4 Servers
                  draft-ietf-behave-v6v4-xlate-stateful-12

   Abstract

      This document describes stateful NAT64 translation, which allows
      IPv6-only clients to contact IPv4 servers using unicast UDP, TCP, or
      ICMP. The public IPv4 address can be shared among several IPv6-only
      clients. When the stateful NAT64 is used in conjunction with DNS64
      no changes are usually required in the IPv6 client or the IPv4
      server.

   Status of this Memo

      This Internet-Draft is submitted in full conformance with the
      provisions of BCP 78 and BCP 79.

      Internet-Drafts are working documents of the Internet Engineering
      Task Force (IETF). Note that other groups may also distribute
      working documents as Internet-Drafts. The list of current Internet-
      Drafts is at http://datatracker.ietf.org/drafts/current/.

      Internet-Drafts are draft documents valid for a maximum of six months
      and may be updated, replaced, or obsoleted by other documents at any
      time. It is inappropriate to use Internet-Drafts as reference
      material or to cite them other than as "work in progress."

      This Internet-Draft will expire on January 11, 2011.

Table of Contents

1. Introduction

   This document specifies stateful NAT64, a mechanism for IPv4-IPv6
   transition and co-existence. Together with DNS64
   [I-D.ietf-behave-dns64], these two mechanisms allow an IPv6-only
   client to initiate communications to an IPv4-only server. They also
   enable peer-to-peer communication between an IPv4 and an IPv6 node,
   where the communication can be initiated by either end using
   existing, NAT-traversal, peer-to-peer communication techniques, such
   as ICE [RFC5245]. Stateful NAT64 also supports IPv4-initiated
   communications to a subset of the IPv6 hosts through statically
   configured bindings in the stateful NAT64.

   Stateful NAT64 is a mechanism for translating IPv6 packets to IPv4
   packets and vice-versa. The translation is done by translating the
   packet headers according to the IP/ICMP Translation Algorithm defined
   in [I-D.ietf-behave-v6v4-xlate]. The IPv4 addresses of IPv4 hosts
   are algorithmically translated to and from IPv6 addresses by using
   the algorithm defined in [I-D.ietf-behave-address-format] and a
   prefix assigned to the stateful NAT64 for this specific purpose. The
   IPv6 addresses of IPv6 hosts are translated to and from IPv4
   addresses by installing mappings in the normal NAPT manner [RFC3022].
   The current specification only defines how stateful NAT64 translates
   unicast packets carrying TCP, UDP and ICMP traffic. Multicast
   packets and other protocols, including SCTP, DCCP and IPsec are out
   of the scope of this specification.

   DNS64 is a mechanism for synthesizing AAAA resource records (RR) from
   A RR. The IPv6 address contained in the synthetic AAAA RR is
   algorithmically generated from the IPv4 address and the IPv6 prefix
   assigned to a NAT64 device by using the same algorithm defined in
   [I-D.ietf-behave-address-format].

   Together, these two mechanisms allow an IPv6-only client (i.e. either
   a host with only IPv6 stack, or a host with both IPv4 and IPv6 stack,
   but only with IPv6 connectivity or a host running an IPv6 only
   application) to initiate communications to an IPv4-only server
   (analogous meaning to the IPv6-only host above).

   These mechanisms are expected to play a critical role in the IPv4-
   IPv6 transition and co-existence. Due to IPv4 address depletion, it
   is likely that in the future, the new clients will be IPv6-only and
   they will want to connect to the existent IPv4-only servers. The
   stateful NAT64 and DNS64 mechanisms are easily deployable, since they
   require no changes to either the IPv6 client nor the IPv4 server.
   For basic functionality, the approach only requires the deployment of
   the stateful NAT64 function in the devices connecting an IPv6-only
   network to the IPv4-only network, along with the deployment of a few

DNS64-enabled name servers accessible to the IPv6-only hosts. An
analysis of the application scenarios can be found in
[I-D.ietf-behave-v6v4-framework].

For brevity, in the rest of the document, we will refer to the
stateful NAT64 either as stateful NAT64 or simply as NAT64.

1.1. Features of stateful NAT64

The features of NAT64 are:

o NAT64 is compliant with the recommendations for how NATs should
   handle UDP [RFC4787], TCP [RFC5382], and ICMP [RFC5508]. As such,
   NAT64 only supports Endpoint-Independent mappings and supports
   both Endpoint-Independent and Address-Dependent Filtering.
   Because of the compliance with the aforementioned requirements,
   NAT64 is compatible with current NAT traversal techniques, such as
   ICE [RFC5245] and compatible with other non-IETF-standard NAT
   traversal techniques.

o In the absence of any state in NAT64, only IPv6 nodes can initiate
   sessions to IPv4 nodes. This works for roughly the same class of
   applications that work through IPv4-to-IPv4 NATs.

o Depending on the filtering policy used (Endpoint-Independent, or
   Address-Dependent), IPv4-nodes might be able to initiate sessions
   to a given IPv6 node, if the NAT64 somehow has an appropriate
   mapping (i.e.,state) for an IPv6 node, via one of the following
   mechanisms:

   * The IPv6 node has recently initiated a session to the same or
      another IPv4 node. This is also the case if the IPv6 node has
      used a NAT-traversal technique (such as ICE) .

   * If a statically configured mapping exists for the IPv6 node.

o IPv4 address sharing: NAT64 allows multiple IPv6-only nodes to
   share an IPv4 address to access the IPv4 Internet. This helps
   with IPv4 forthcoming exhaustion.

o As currently defined in this NAT64 specification, only TCP/UDP/
   ICMP are supported. Support for other protocols such as other
   transport protocols and IPsec are to be defined in separate
   documents.

1.2. Overview

   This section provides a non-normative introduction to NAT64. This is
   achieved by describing the NAT64 behavior involving a simple setup,
   that involves a single NAT64 device, a single DNS64 and a simple
   network topology. The goal of this description is to provide the
   reader with a general view of NAT64. It is not the goal of this
   section to describe all possible configurations nor to provide a
   normative specification of the NAT64 behavior. So, for the sake of
   clarity, only TCP and UDP are described in this overview; the details
   of ICMP, fragmentation, and other aspects of translation are
   purposefully avoided in this overview. The normative specification
   of NAT64 is provided in Section 3.

   The NAT64 mechanism is implemented in a device which has (at least)
   two interfaces, an IPv4 interface connected to the IPv4 network, and
   an IPv6 interface connected to the IPv6 network. Packets generated
   in the IPv6 network for a receiver located in the IPv4 network will
   be routed within the IPv6 network towards the NAT64 device. The
   NAT64 will translate them and forward them as IPv4 packets through
   the IPv4 network to the IPv4 receiver. The reverse takes place for
   packets generated by hosts connected to the IPv4 network for an IPv6
   receiver. NAT64, however, is not symmetric. In order to be able to
   perform IPv6-IPv4 translation, NAT64 requires state, binding an IPv6
   address and TCP/UDP port (hereafter called an IPv6 transport address)
   to an IPv4 address and TCP/UDP port (hereafter called an IPv4
   transport address).

   Such binding state is either statically configured in the NAT64 or it
   is created when the first packet flowing from the IPv6 network to the
   IPv4 network is translated. After the binding state has been
   created, packets flowing in both directions on that particular flow
   are translated. The result is that, in the general case, NAT64 only
   supports communications initiated by the IPv6-only node towards an
   IPv4-only node. Some additional mechanisms (like ICE) or static
   binding configuration, can be used to provide support for
   communications initiated by an IPv4-only node to an IPv6-only node.

1.2.1. Stateful NAT64 solution elements

   In this section we describe the different elements involved in the
   NAT64 approach.

   The main component of the proposed solution is the translator itself.
   The translator has essentially two main parts, the address
   translation mechanism and the protocol translation mechanism.

   Protocol translation from IPv4 packet header to IPv6 packet header

and vice-versa is performed according to the IP/ICMP Translation
Algorithm [I-D.ietf-behave-v6v4-xlate].

Address translation maps IPv6 transport addresses to IPv4 transport
addresses and vice-versa. In order to create these mappings the
NAT64 has two pools of addresses: an IPv6 address pool (to represent
IPv4 addresses in the IPv6 network) and an IPv4 address pool (to
represent IPv6 addresses in the IPv4 network).

The IPv6 address pool is one or more IPv6 prefixes assigned to the
translator itself. Hereafter we will call the IPv6 address pool as
Pref64::/n; in the case there are more than one prefix assigned to
the NAT64, the comments made about Pref64::/n apply to each of them.
Pref64::/n will be used by the NAT64 to construct IPv4-Converted IPv6
addresses as defined in [I-D.ietf-behave-address-format]. Due to the
abundance of IPv6 address space, it is possible to assign one or more
Pref64::/n, each of them being equal to or even bigger than the size
of the whole IPv4 address space. This allows each IPv4 address to be
mapped into a different IPv6 address by simply concatenating a
Pref64::/n with the IPv4 address being mapped and a suffix. The
provisioning of the Pref64::/n as well as the address format are
defined in [I-D.ietf-behave-address-format].

The IPv4 address pool is a set of IPv4 addresses, normally a prefix
assigned by the local administrator. Since IPv4 address space is a
scarce resource, the IPv4 address pool is small and typically not
sufficient to establish permanent one-to-one mappings with IPv6
addresses. So, except for the static/manually created ones, mappings
using the IPv4 address pool will be created and released dynamically.
Moreover, because of the IPv4 address scarcity, the usual practice
for NAT64 is likely to be the binding of IPv6 transport addresses
into IPv4 transport addresses, instead of IPv6 addresses into IPv4
addresses directly, enabling a higher utilization of the limited IPv4
address pool. This implies that NAT64 performs both address and port
translation.

Because of the dynamic nature of the IPv6 to IPv4 address mapping and
the static nature of the IPv4 to IPv6 address mapping, it is far
simpler to allow communications initiated from the IPv6 side toward
an IPv4 node, whose address is algorithmically mapped into an IPv6
address, than communications initiated from IPv4-only nodes to an
IPv6 node in which case an IPv4 address needs to be associated with
the IPv6 node's address dynamically.

Using a mechanisms such as DNS64, an IPv6 client obtains an IPv6
address that embeds the IPv4 address of the IPv4 server, and sends a
packet to that IPv6 address. The packets are intercepted by the
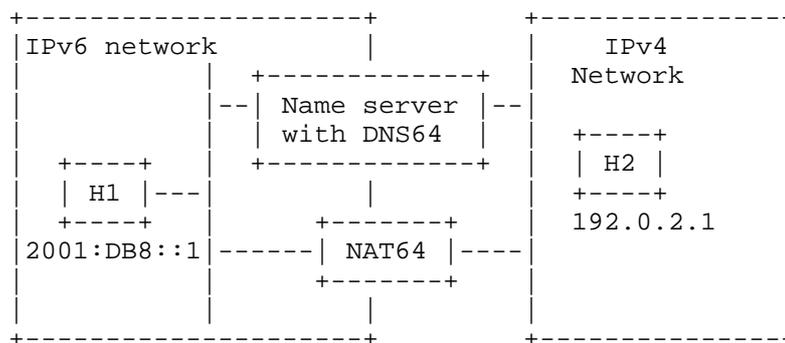NAT64 device, which associates an IPv4 transport address of its IPv4

pool to the IPv6 transport address of the initiator, creating binding
state, so that reply packets can be translated and forwarded back to
the initiator. The binding state is kept while packets are flowing.
Once the flow stops, and based on a timer, the IPv4 transport address
is returned to the IPv4 address pool so that it can be reused for
other communications.

To allow an IPv6 initiator to do a DNS lookup to learn the address of
the responder, DNS64 [I-D.ietf-behave-dns64] is used to synthesize
AAAA RRs from the A RRs. The IPv6 addresses contained in the
synthetic AAAA RRs contain a Pref64::/n assigned to the NAT64 and the
IPv4 address of the responder. The synthetic AAAA RRs are passed
back to the IPv6 initiator, which will initiate an IPv6 communication
with an IPv6 address associated to the IPv4 receiver. The packet
will be routed to the NAT64 device, which will create the IPv6 to
IPv4 address mapping as described before.

1.2.2. Stateful NAT64 Behaviour Walkthrough

In this section we provide a simple example of the NAT64 behaviour.
We consider an IPv6 node located in an IPv6-only site that initiates
a TCP connection to an IPv4-only node located in the IPv4 network.

The scenario for this case is depicted in the following figure:

```
        +--------------------+      +--------------+
        |IPv6 network        |      |    IPv4      |
        |          |  +-------------+ |  Network    |
        |          |--| Name server |--|           |
        |          |  | with DNS64  | |  +----+    |
        | +----+   |  +-------------+ |  | H2 |    |
        | | H1 |---|       |          |  +----+    |
        | +----+   |     +-------+    |  192.0.2.1 |
        |2001:DB8::1|-----| NAT64 |----|           |
        |          |     +-------+    |           |
        |          |       |          |           |
        +--------------------+      +--------------+
```

The figure above shows an IPv6 node H1 with an IPv6 address 2001:
DB8::1 and an IPv4 node H2 with IPv4 address 192.0.2.1 H2 has
h2.example.com as FQDN.

A NAT64 connects the IPv6 network to the IPv4 network. This NAT64
uses the Well-Known Prefix 64:FF9B::/96 defined in
[I-D.ietf-behave-address-format] to represent IPv4 addresses in the
IPv6 address space and a single IPv4 address 203.0.113.1 assigned to
its IPv4 interface. The routing is configured in such a way that the

IPv6 packets addressed to a destination address in 64:FF9B::/96 are
routed to the IPv6 interface of the NAT64 device.

Also shown is a local name server with DNS64 functionality. The
local name server uses the Well-Known prefix 64:FF9B::/96 to create
the IPv6 addresses in the synthetic RRs.

For this example, assume the typical DNS situation where IPv6 hosts
have only stub resolvers and the local name server does the recursive
lookups.

The steps by which H1 establishes communication with H2 are:

1. H1 performs a DNS query for h2.example.com and receives the
    synthetic AAAA RR from the local name server that implements the
    DNS64 functionality. The AAAA record contains an IPv6 address
    formed by the Well-Known Prefix and the IPv4 address of H2 (i.e.
    64:FF9B::192.0.2.1).

2. H1 sends a TCP SYN packet to H2. The packet is sent from a
    source transport address of (2001:DB8::1,1500) to a destination
    transport address of (64:FF9B::192.0.2.1,80), where the ports are
    set by H1.

3. The packet is routed to the IPv6 interface of the NAT64 (since
    IPv6 routing is configured that way).

4. The NAT64 receives the packet and performs the following actions:

    * The NAT64 selects an unused port (e.g. 2000) on its IPv4
       address 203.0.113.1 and creates the mapping entry (2001:DB8::
       1,1500) <--> (203.0.113.1,2000)

    * The NAT64 translates the IPv6 header into an IPv4 header using
       the IP/ICMP Translation Algorithm
       [I-D.ietf-behave-v6v4-xlate].

    * The NAT64 includes (203.0.113.1,2000) as source transport
       address in the packet and (192.0.2.1,80) as destination
       transport address in the packet. Note that 192.0.2.1 is
       extracted directly from the destination IPv6 address of the
       received IPv6 packet that is being translated. The
       destination port 80 of the translated packet is the same as
       the destination port of the received IPv6 packet.

5. The NAT64 sends the translated packet out its IPv4 interface and
    the packet arrives at H2.

   6. H2 node responds by sending a TCP SYN+ACK packet with destination
       transport address (203.0.113.1,2000) and source transport address
       (192.0.2.1,80).

   7. Since the IPv4 address 203.0.113.1 is assigned to the IPv4
       interface of the NAT64 device, the packet is routed to the NAT64
       device, which will look for an existing mapping containing
       (203.0.113.1,2000). Since the mapping (2001:DB8::1,1500) <-->
       (203.0.113.1,2000) exists, the NAT64 performs the following
       operations:

       * The NAT64 translates the IPv4 header into an IPv6 header using
          the IP/ICMP Translation Algorithm
          [I-D.ietf-behave-v6v4-xlate].

       * The NAT64 includes (2001:DB8::1,1500) as destination transport
          address in the packet and (64:FF9B::192.0.2.1,80) as source
          transport address in the packet. Note that 192.0.2.1 is
          extracted directly from the source IPv4 address of the
          received IPv4 packet that is being translated. The source
          port 80 of the translated packet is the same as the source
          port of the received IPv4 packet.

   8. The translated packet is sent out the IPv6 interface to H1.

   The packet exchange between H1 and H2 continues and packets are
   translated in the different directions as previously described.

   It is important to note that the translation still works if the IPv6
   initiator H1 learns the IPv6 representation of H2's IPv4 address
   (i.e., 64:FF9B::192.0.2.1) through some scheme other than a DNS
   look-up. This is because the DNS64 processing does NOT result in any
   state installed in the NAT64 and because the mapping of the IPv4
   address into an IPv6 address is the result of concatenating the Well-
   Known Prefix to the original IPv4 address.

1.2.3. Filtering

   NAT64 may do filtering, which means that it only allows a packet in
   through an interface under certain circumstances. The NAT64 can
   filter IPv6 packets based on the administrative rules to create
   entries in the binding and session tables. The filtering can be
   flexible and general but the idea of the filtering is to provide the
   administrators necessary control to avoid DoS attacks that would
   result in exhaustion of the NAT64's IPv4 address, port, memory and
   CPU resources. Filtering techniques of incoming IPv6 packets are not
   specific to the NAT64 and therefore are not described in this
   specification.

Filtering of IPv4 packets on the other hand is tightly coupled to the NAT64 state and therefore is described in this specification. In this document, we consider that the NAT64 may do no filtering, or it may filter incoming IPv4 packets.

NAT64 filtering of incoming IPv4 packets is consistent with the recommendations of [RFC4787], and the ones of [RFC5382]. Because of that, the NAT64 as specified in this document, supports both Endpoint-Independent Filtering and Address-Dependent Filtering, both for TCP and UDP as well as filtering of ICMP packets.

If a NAT64 performs Endpoint-Independent Filtering of incoming IPv4 packets, then an incoming IPv4 packet is dropped unless the NAT64 has state for the destination transport address of the incoming IPv4 packet.

If a NAT64 performs Address-Dependent Filtering of incoming IPv4 packets, then an incoming IPv4 packet is dropped unless the NAT64 has state involving the destination transport address of the IPv4 incoming packet and the particular source IP address of the incoming IPv4 packet.

2. Terminology

This section provides a definitive reference for all the terms used in this document.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

The following additional terms are used in this document:

3-Tuple: The tuple (source IP address, destination IP address, ICMP Identifier). A 3-tuple uniquely identifies an ICMP Query session. When an ICMP Query session flows through a NAT64, each session has two different 3-tuples: one with IPv4 addresses and one with IPv6 addresses.

5-Tuple: The tuple (source IP address, source port, destination IP address, destination port, transport protocol). A 5-tuple uniquely identifies a UDP/TCP session. When a UDP/TCP session flows through a NAT64, each session has two different 5-tuples: one with IPv4 addresses and one with IPv6 addresses.

BIB: Binding Information Base. A table of bindings kept by a NAT64.
   Each NAT64 has a BIB for each translated protocol. An
   implementation compliant to this document would have a BIB for
   TCP, one for UDP and one for ICMP Queries. Additional BIBs would
   be added to support other protocols, such as SCTP.

Endpoint-Independent Mapping: In NAT64, using the same mapping for
   all the sessions involving a given IPv6 transport address of an
   IPv6 host (irrespectively of the transport address of the IPv4
   host involved in the communication). Endpoint-independent Mapping
   is important for peer-to-peer communication. See [RFC4787] for
   the definition of the different types of mappings in IPv4-to-IPv4
   NATs.

Filtering, Endpoint-Independent: The NAT64 only filters incoming
   IPv4 packets destined to a transport address for which there is no
   state in the NAT64, regardless of the source IPv4 transport
   address. The NAT forwards any packets destined to any transport
   address for which it has state. In other words, having state for
   a given transport address is sufficient to allow any packets back
   to the internal endpoint. See [RFC4787] for the definition of the
   different types of filtering in IPv4-to-IPv4 NATs.

Filtering, Address-Dependent: The NAT64 filters incoming IPv4
   packets destined to a transport address for which there is no
   state (similar to the Endpoint-Independent Filtering).
   Additionally, the NAT64 will filter out incoming IPv4 packets
   coming from a given IPv4 address X and destined for a transport
   address that it has state for if the NAT64 has not sent packets to
   X previously (independently of the port used by X). In other
   words, for receiving packets from a specific IPv4 endpoint, it is
   necessary for the IPv6 endpoint to send packets first to that
   specific IPv4 endpoint's IP address.

Hairpinning: Having a packet do a "U-turn" inside a NAT and come
   back out the same side as it arrived on. If the destination IPv6
   address and its embedded IPv4 address are both assigned to the
   NAT64 itself, then the packet is being sent to another IPv6 host
   connected to the same NAT64. Such a packet is called a 'hairpin
   packet'. A NAT64 that forwards hairpin packets, back to the IPv6
   host are defined as supporting "hairpinning". Hairpinning support
   is important for peer-to-peer applications, as there are cases
   when two different hosts on the same side of a NAT can only
   communicate using sessions that hairpin through the NAT. Hairpin
   packets can be either TCP or UDP. More detailed explanation of
   hairpinning and examples for the UDP case can be found in
   [RFC4787].

ICMP Query packet: ICMP packets that are not ICMP error messages.
   For ICMPv6, ICMPv6 Query Messages are the ICMPv6 Informational
   messages as defined in [RFC4443]. For ICMPv4, ICMPv4 Query
   messages are all ICMPv4 messages that are not ICMPv4 error
   messages.

Mapping or Binding: A mapping between an IPv6 transport address and
   a IPv4 transport address or a mapping between an (IPv6 address,
   ICMPv6 Identifier) pair and an (IPv4 address, ICMPv4 Identifier)
   pair. Used to translate the addresses and ports/ICMP Identifiers
   of packets flowing between the IPv6 host and the IPv4 host. In
   NAT64, the IPv4 address and port/ICMPv4 Identifier is always one
   assigned to the NAT64 itself, while the IPv6 address and port/
   ICMPv6 Identifier belongs to some IPv6 host.

Session: The flow of packets between two different hosts. This may
   be TCP, UDP or ICMP Queries. In NAT64, typically one host is an
   IPv4 host, and the other one is an IPv6 host. However, due to
   hairpinning, both hosts might be IPv6 hosts.

Session table: A table of sessions kept by a NAT64. Each NAT64 has
   three session tables, one for TCP, one for UDP and one for ICMP
   Queries.

Stateful NAT64: A function that has per-flow state which translates
   IPv6 packets to IPv4 packets and vice-versa, for TCP, UDP, and
   ICMP. The NAT64 uses binding state to perform the translation
   between IPv6 and IPv4 addresses. In this document we also refer
   to stateful NAT64 simply as NAT64.

Stateful NAT64 device: The device where the NAT64 function is
   executed. In this document we also refer to stateful NAT64 device
   simply as NAT64 device.

Transport Address: The combination of an IPv6 or IPv4 address and a
   port. Typically written as (IP address, port)- e.g. (192.0.2.15,
   8001).

Tuple: Refers to either a 3-Tuple or a 5-tuple as defined above.

For a detailed understanding of this document, the reader should also
be familiar with NAT terminology [RFC4787].

3. Stateful NAT64 Normative Specification

   A NAT64 is a device with at least one IPv6 interface and at least one
   IPv4 interface. Each NAT64 device MUST have at least one unicast /n

   IPv6 prefix assigned to it, denoted Pref64::/n. Additional
   considerations about the Pref64::/n are presented in Section 3.5.4.
   A NAT64 MUST have one or more unicast IPv4 addresses assigned to it.

   A NAT64 uses the following conceptual dynamic data structures:

   o UDP Binding Information Base

   o UDP Session Table

   o TCP Binding Information Base

   o TCP Session Table

   o ICMP Query Binding Information Base

   o ICMP Query Session Table

   These tables contain information needed for the NAT64 processing.
   The actual division of the information into six tables is done in
   order to ease the description of the NAT64 behaviour. NAT64
   implementations are free to use different data structures but they
   MUST store all the required information and the externally visible
   outcome MUST be the same as the one described in this document.

   The notation used is the following: upper case letters are IPv4
   addresses; upper case letters with a prime(') are IPv6 addresses;
   lower case letters are ports; IPv6 prefixes of length n are indicated
   by "P::/n", mappings are indicated as "(X,x) <--> (Y',y)".

3.1. Binding Information Bases

   A NAT64 has three Binding Information Bases (BIBs): one for TCP, one
   for UDP and one for ICMP Queries. In the case of UDP and TCP BIBs,
   each BIB entry specifies a mapping between an IPv6 transport address
   and an IPv4 transport address:

      (X',x) <--> (T,t)

   where X' is some IPv6 address, T is an IPv4 address, and x and t are
   ports. T will always be one of the IPv4 addresses assigned to the
   NAT64. The BIB has then two columns: the BIB IPv6 transport address
   and the BIB IPv4 transport address. A given IPv6 or IPv4 transport
   address can appear in at most one entry in a BIB: for example, (2001:
   db8::17, 49832) can appear in at most one TCP and at most one UDP BIB
   entry. TCP and UDP have separate BIBs because the port number space
   for TCP and UDP are distinct. If the BIBs are implemented as
   specified in this document, it results in Endpoint-Independent

Mappings in the NAT64. The information in the BIBs is also used to
implement Endpoint-Independent Filtering. (Address-Dependent
Filtering is implemented using the session tables described below.)

In the case of the ICMP Query BIB, each ICMP Query BIB entry
specifies a mapping between an (IPv6 address, ICMPv6 Identifier) pair
and an (IPv4 address, ICMPv4 Identifier) pair.

    (X',i1) <--> (T,i2)

where X' is some IPv6 address, T is an IPv4 address, i1 is an ICMPv6
Identifier and i2 is an ICMPv4 Identifier. T will always be one of
the IPv4 addresses assigned to the NAT64. A given (IPv6 or IPv4
address, ICMPv6 or ICMPv4 Identifier) pair can appear in at most one
entry in the ICMP Query BIB.

Entries in any of the three BIBs can be created dynamically as the
result of the flow of packets as described in Section 3.5 but they
can also be created manually by an administrator. NAT64
implementations SHOULD support manually configured BIB entries for
any of the three BIBs. Dynamically-created entries are deleted from
the corresponding BIB when the last session associated with the BIB
entry is removed from the session table. Manually-configured BIB
entries are not deleted when there is no corresponding session table
entry and can only be deleted by the administrator.

3.2. Session Tables

A NAT64 also has three session tables: one for TCP sessions, one for
UDP sessions, and one for ICMP Query sessions. Each entry keeps
information on the state of the corresponding session. In the TCP
and UDP session tables, each entry specifies a mapping between a pair
of IPv6 transport addresses and a pair of IPv4 transport addresses:

    (X',x),(Y',y) <--> (T,t),(Z,z)

where X' and Y' are IPv6 addresses, T and Z are IPv4 addresses, and
x, y, z and t are ports. T will always be one of the IPv4 addresses
assigned to the NAT64. Y' is always the IPv6 representation of the
IPv4 address Z, so Y' is obtained from Z using the algorithm applied
by the NAT64 to create IPv6 representations of IPv4 addresses. y will
always be equal to z.

For each TCP or UDP Session Table Entry (STE), there are then five
columns. The terminology used for the session table entry columns is
from the perspective of an incoming IPv6 packet being translated into
an outgoing IPv4 packet. The columns are::

The STE source IPv6 transport address, (X',x) in the example
above,

The STE destination IPv6 transport address, (Y',y) in the example
above,

The STE source IPv4 transport address, (T,t) in the example above,
and,

The STE destination IPv4 transport address, (Z,z) in the example
above.

The STE lifetime.

In the ICMP query session table, each entry specifies a mapping
between a 3-tuple of IPv6 source address, IPv6 destination address
and ICMPv6 Identifier and a 3-tuple of IPv4 source address, IPv4
destination address and ICMPv4 Identifier:

   (X',Y',i1) <--> (T,Z,i2)

where X' and Y' are IPv6 addresses, T and Z are IPv4 addresses, i1 is
an ICMPv6 Identifier and i2 is an ICMPv4 Identifier. T will always
be one of the IPv4 addresses assigned to the NAT64. Y' is always the
IPv6 representation of the IPv4 address Z, so Y' is obtained from Z
using the algorithm applied by the NAT64 to create IPv6
representations of IPv4 addresses.

For each ICMP Query Session Table Entry (STE), there are then seven
columns:

The STE source IPv6 address, X' in the example above,

The STE destination IPv6 address, Y' in the example above,

The STE ICMPv6 Identifier, i1 in the example above,

The STE source IPv4 address, T in the example above,

The STE destination IPv4 address, Z in the example above, and,

The STE ICMPv4 Identifier, i2 in the example above.

The STE lifetime.

3.3. Packet Processing Overview

   The NAT64 uses the session state information to determine when the
   session is completed, and also uses session information for Address-
   Dependent Filtering. A session can be uniquely identified by either
   an incoming tuple or an outgoing tuple.

   For each TCP or UDP session, there is a corresponding BIB entry,
   uniquely specified by either the source IPv6 transport address (in
   the IPv6 --> IPv4 direction) or the destination IPv4 transport
   address (in the IPv4 --> IPv6 direction). For each ICMP Query
   session, there is a corresponding BIB entry, uniquely specified by
   either the source IPv6 address and ICMPv6 Identifier (in the IPv6 -->
   IPv4 direction) or the destination IPv4 address and the ICMPv4
   Identifier (in the IPv4 --> IPv6 direction). However, for all the
   BIBs, a single BIB entry can have multiple corresponding sessions.
   When the last corresponding session is deleted, if the BIB entry was
   dynamically created, the BIB entry is deleted.

   The NAT64 will receive packets through its interfaces. These packets
   can be either IPv6 packets or IPv4 packets and they may carry TCP
   traffic, UDP traffic or ICMP traffic. The processing of the packets
   will be described next. In the case that the processing is common to
   all the aforementioned types of packets, we will refer to the packet
   as the incoming IP packet in general. In case that the processing is
   specific to IPv6 packets, we will refer to the incoming IPv6 packet
   and similarly to the IPv4 packets.

   The processing of an incoming IP packet takes the following steps:

   1. Determining the incoming tuple

   2. Filtering and updating binding and session information

   3. Computing the outgoing tuple

   4. Translating the packet

   5. Handling hairpinning

   The details of these steps are specified in the following
   subsections.

   This breakdown of the NAT64 behavior into processing steps is done
   for ease of presentation. A NAT64 MAY perform the steps in a
   different order, or MAY perform different steps, but the externally
   visible outcome MUST be the same as the one described in this
   document.

3.4. Determining the Incoming tuple

   This step associates an incoming tuple with every incoming IP packet
   for use in subsequent steps. In the case of TCP, UDP and ICMP error
   packets, the tuple is a 5-tuple consisting of source IP address,
   source port, destination IP address, destination port, transport
   protocol. In case of ICMP Queries, the tuple is a 3-tuple consisting
   of the source IP address, destination IP address and ICMP Identifier.

   If the incoming IP packet contains a complete (un-fragmented) UDP or
   TCP protocol packet, then the 5-tuple is computed by extracting the
   appropriate fields from the received packet.

   If the incoming packet is a complete (un-fragmented) ICMP query
   message (i.e., an ICMPv4 Query message or an ICMPv6 Informational
   message), the 3-tuple is the source IP address, the destination IP
   address and the ICMP Identifier.

   If the incoming IP packet contains a complete (un-fragmented) ICMP
   error message containing a UDP or a TCP packet, then the incoming
   5-tuple is computed by extracting the appropriate fields from the IP
   packet embedded inside the ICMP error message. However, the role of
   source and destination is swapped when doing this: the embedded
   source IP address becomes the destination IP address in the incoming
   5-tuple, the embedded source port becomes the destination port in the
   incoming 5-tuple, etc. If it is not possible to determine the
   incoming 5-tuple (perhaps because not enough of the embedded packet
   is reproduced inside the ICMP message), then the incoming IP packet
   MUST be silently discarded.

   If the incoming IP packet contains a complete (un-fragmented) ICMP
   error message containing a ICMP error message, then the packet is
   silently discarded.

   If the incoming IP packet contains a complete (un-fragmented) ICMP
   error message containing an ICMP Query message, then the incoming
   3-tuple is computed by extracting the appropriate fields from the IP
   packet embedded inside the ICMP error message. However, the role of
   source and destination is swapped when doing this: the embedded
   source IP address becomes the destination IP address in the incoming
   3-tuple, the embedded destination IP address becomes the source
   address in the incoming 3-tuple and the embedded ICMP Identifier is
   used as the ICMP Identifier of the incoming 3-tuple. If it is not
   possible to determine the incoming 3-tuple (perhaps because not
   enough of the embedded packet is reproduced inside the ICMP message),
   then the incoming IP packet MUST be silently discarded.

   If the incoming IP packet contains a fragment, then more processing

may be needed. This specification leaves open the exact details of
how a NAT64 handles incoming IP packets containing fragments, and
simply requires that the external behavior of the NAT64 is compliant
with the following conditions:

   The NAT64 MUST handle fragments. In particular, NAT64 MUST handle
   fragments arriving out-of-order , conditioned on the following:

   * The NAT64 MUST limit the amount of resources devoted to the
      storage of fragmented packets in order to protect from DoS
      attacks.

   * As long as the NAT64 has available resources, the NAT64 MUST
      allow the fragments to arrive over a time interval. The time
      interval SHOULD be configurable and the default value MUST be
      of at least FRAGMENT_MIN.

   * The NAT64 MAY require that the UDP, TCP, or ICMP header be
      completely contained within the fragment that contains fragment
      offset equal to zero.

   For incoming packets carrying TCP or UDP fragments with non-null
   checksum, NAT64 MAY elect to queue the fragments as they arrive
   and translate all fragments at the same time. In this case, the
   incoming tuple is determined as documented above to the un-
   fragmented packets. Alternatively, a NAT64 MAY translate the
   fragments as they arrive, by storing information that allows it to
   compute the 5-tuple for fragments other than the first. In the
   latter case, subsequent fragments may arrive before the first and
   the rules about how the NAT64 handles (out-of-order) fragments
   described in the bulleted list above apply.

   For incoming IPv4 packets carrying UDP packets with null checksum,
   if the NAT64 has enough resources, the NAT64 MUST reassemble the
   packets and MUST calculate the checksum. If the NAT64 does not
   have enough resources, then it MUST silently discard the packets.

   Implementers of NAT64 should be aware that there are a number of
   well-known attacks against IP fragmentation; see [RFC1858] and
   [RFC3128]. Implementers should also be aware of additional issues
   with reassembling packets at high rates, described in [RFC4963].

If the incoming packet is an IPv6 packet that contains a protocol
other than TCP, UDP or ICMPv6 in the last Next Header, then the
packet SHOULD be discarded and, if the security policy permits, the
NAT64 SHOULD send an ICMPv6 Destination Unreachable error message
with Code 3 (Destination Unreachable) to the source address of the
received packet. NOTE: This behaviour may be updated by future

documents that define how other protocols such as SCTP or DCCP are
processed by NAT64.

If the incoming packet is an IPv4 packet that contains a protocol
other than TCP, UDP or ICMPv4, then the packet SHOULD discarded and,
if the security policy permits, the NAT64 SHOULD send an ICMPv4
Destination Unreachable error message with Code 2 (Protocol
Unreachable) to the source address of the received packet. NOTE:
This behaviour may be updated by future documents that define how
other protocols such as SCTP or DCCP are processed by NAT64.

3.5. Filtering and Updating Binding and Session Information

This step updates binding and session information stored in the
appropriate tables. This step may also filter incoming packets, if
desired.

The details of this step depend on the protocol i.e. UDP, TCP or
ICMP. The behaviour for UDP is described in Section 3.5.1, for TCP
is described in Section 3.5.2 and for ICMP Queries is described in
Section 3.5.3. For the case of ICMP error messages, they do not
affect in any way neither the BIBs nor the session tables, so, there
is no processing resulting from these messages in this section. ICMP
error message processing continues in Section 3.6.

Irrespective of the transport protocol used, the NAT64 MUST silently
discard all incoming IPv6 packets containing a source address that
contains the Pref64::/n. This is required in order to prevent
hairpinning loops as described in Section 5. In addition, the NAT64
MUST only process incoming IPv6 packets that contain a destination
address that contains Pref64::/n. Likewise, the NAT64 MUST only
process incoming IPv4 packets that contain a destination address that
belong to the IPv4 pool assigned to the NAT64.

3.5.1. UDP Session Handling

The following state information is stored for a UDP session:

Binding:(X',x),(Y',y) <--> (T,t),(Z,z)

Lifetime: a timer that tracks the remaining lifetime of the UDP
session. When the timer expires, the UDP session is deleted. If
all the UDP sessions corresponding to a dynamically created UDP
BIB entry are deleted, then the UDP BIB entry is also deleted.

An IPv6 incoming packet with an incoming tuple with source transport
address (X',x) and destination transport address (Y',y) is processed
as follows:

The NAT64 searches for a UDP BIB entry that contains the BIB IPv6
transport address that matches the IPv6 source transport address
(X',x). If such an entry does not exist, the NAT64 tries to
create a new entry (if resources and policy permit). The source
IPv6 transport address of the packet (X',x) is used as BIB IPv6
transport address, and the BIB IPv4 transport address is set to
(T,t) which is allocated using the rules defined in
Section 3.5.1.1. The result is a BIB entry as follows: (X',x)
<--> (T,t).

The NAT64 searches for the session table entry corresponding to
the incoming 5-tuple. If no such entry is found, the NAT64 tries
to create a new entry (if resources and policy permit). The
information included in the session table is as follows:

*   The STE source IPv6 transport address is set to (X',x), the
    source IPv6 transport addresses contained in the received IPv6
    packet,

*   The STE destination IPv6 transport address is set to (Y',y),
    the destination IPv6 transport addresses contained in the
    received IPv6 packet,

*   The STE source IPv4 transport address is extracted from the
    corresponding UDP BIB entry i.e. it is set to (T,t),

*   The STE destination IPv4 transport is set to (Z(Y'),y), y being
    the same port as the STE destination IPv6 transport address and
    Z(Y') being algorithmically generated from the IPv6 destination
    address (i.e. Y') using the reverse algorithm (see
    Section 3.5.4).

The result is a Session table entry as follows: (X',x),(Y',y) <-->
(T,t),(Z(Y'),y)

The NAT64 sets (or resets) the timer in the Session Table Entry to
the maximum session lifetime. The maximum session lifetime MAY be
configurable and the default SHOULD be at least UDP_DEFAULT. The
maximum session lifetime MUST NOT be less than UDP_MIN. The
packet is translated and forwarded as described in the following
sections.

An IPv4 incoming packet, with an incoming tuple with source IPv4
transport address (W,w) and destination IPv4 transport address (T,t)
is processed as follows:

The NAT64 searches for a UDP BIB entry that contains the BIB IPv4
transport address matching (T,t), (i.e., the IPv4 destination

transport address in the incoming IPv4 packet). If such an entry
does not exist, the packet MUST be dropped. An ICMP error message
with type of 3 (Destination Unreachable) MAY be sent to the
original sender of the packet.

If the NAT64 applies Address-Dependent Filters on its IPv4
interface, then the NAT64 checks to see if the incoming packet is
allowed according to the Address-Dependent Filtering rule. To do
this, it searches for a session table entry with an STE source
IPv4 transport address equal to (T,t), (i.e., the destination IPv4
transport address in the incoming packet) and STE destination IPv4
address equal to W, (i.e., the source IPv4 address in the incoming
packet). If such an entry is found (there may be more than one),
packet processing continues. Otherwise, the packet is discarded.
If the packet is discarded, then an ICMP error message MAY be sent
to the original sender of the packet. The ICMP error message, if
sent, has a type of 3 (Destination Unreachable) and a code of 13
(Communication Administratively Prohibited).

In case the packet is not discarded in the previous processing
(either because the NAT64 is not filtering or because the packet
is compliant with the Address-Dependent Filtering rule), then the
NAT64 searches for the session table entry containing the STE
source IPv4 transport address equal to (T,t) and the STE
destination IPv4 transport address equal to (W,w). If no such
entry is found, the NAT64 tries to create a new entry (if
resources and policy permit). In case a new UDP session table
entry is created, it contains the following information:

* The STE source IPv6 transport address is extracted from the
   corresponding UDP BIB entry.

* The STE destination IPv6 transport address is set to (Y'(W),w),
   w being the same port w than the source IPv4 transport address
   and Y'(W) being the IPv6 representation of W, generated using
   the algorithm described in Section 3.5.4.

* The STE source IPv4 transport address is set to (T,t) the
   destination IPv4 transport addresses contained in the received
   IPv4 packet.

* The STE destination IPv4 transport is set to (W,w), the source
   IPv4 transport addresses contained in the received IPv4 packet.

The NAT64 sets (or resets) the timer in the Session Table Entry to
the maximum session lifetime. The maximum session lifetime MAY be
configurable and the default SHOULD be at least UDP_DEFAULT. The
maximum session lifetime MUST NOT be less than UDP_MIN. The

packet is translated and forwarded as described in the following
sections.

3.5.1.1. Rules for Allocation of IPv4 Transport Addresses for UDP

When a new UDP BIB entry is created for a source transport address of
(S',s), then the NAT64 allocates an IPv4 transport address for this
BIB entry as follows:

> If there exists some other BIB entry containing S' as the IPv6
> address and mapping it to some IPv4 address T, then the NAT64
> SHOULD use T as the IPv4 address. Otherwise, use any IPv4 address
> of the IPv4 pool assigned to the NAT64 to be used for translation.

> If the port s is in the Well-Known port range 0-1023, and the
> NAT64 has an available port t in the same port range, then the
> NAT64 SHOULD allocate the port t. If the NAT64 does not have a
> port available in the same range, the NAT64 MAY assign a port t
> from other range where it has an available port. (This behavior
> is recommended in REQ 3-a of [RFC4787].)

> If the port s is in the range 1024-65535, and the NAT64 has an
> available port t in the same port range, then the NAT64 SHOULD
> allocate the port t. If the NAT64 does not have a port available
> in the same range, the NAT64 MAY assign a port t from other range
> where it has an available port. (this behavior is recommended in
> REQ 3-a of [RFC4787])

> The NAT64 SHOULD preserve the port parity (odd/even), as per
> Section 4.2.2 of [RFC4787]).

> In all cases, the allocated IPv4 transport address (T,t) MUST NOT
> be in use in another entry in the same BIB, but MAY be in use in
> the other BIB (referring to the UDP and TCP BIBs).

If it is not possible to allocate an appropriate IPv4 transport
address or create a BIB entry, then the packet is discarded. The
NAT64 SHOULD send an ICMPv6 Destination Unreachable/Address
unreachable (Code 3) message.

3.5.2. TCP Session Handling

In this section we describe how the TCP BIB and Session table are
populated. We do so by defining the state machine of the NAT64 uses
for TCP. We first describe the states and the information contained
in them and then we describe the actual state machine and state
transitions.

3.5.2.1. State definition

   The following state information is stored for a TCP session:

      Binding:(X',x),(Y',y) <--> (T,t),(Z,z)

      Lifetime: a timer that tracks the remaining lifetime of the TCP
      session. When the timer expires, the TCP session is deleted. If
      all the TCP sessions corresponding to a TCP BIB entry are deleted,
      then the dynamically created TCP BIB entry is also deleted.

   Because the TCP session inactivity lifetime is set to at least 2
   hours and 4 min (as per [RFC5382]), it is important that each TCP
   session table entry corresponds to an existent TCP session. In order
   to do that, for each TCP session established through it, it tracks
   the corresponding state machine as follows.

   The states are the following ones:

      CLOSED: Analogous to [RFC0793], CLOSED is a fictional state
      because it represents the state when there is no state for this
      particular 5-tuple, and therefore, no connection.

      V4 INIT: An IPv4 packet containing a TCP SYN was received by the
      NAT64, implying that a TCP connection is being initiated from the
      IPv4 side. The NAT64 is now waiting for a matching IPv6 packet
      containing the TCP SYN in the opposite direction.

      V6 INIT: An IPv6 packet containing a TCP SYN was received,
      translated and forwarded by the NAT64, implying that a TCP
      connection is being initiated from the IPv6 side. The NAT64 is
      now waiting for a matching IPv4 packet containing the TCP SYN in
      the opposite direction.

      ESTABLISHED: Represents an open connection, with data able to flow
      in both directions.

      V4 FIN RCV: An IPv4 packet containing a TCP FIN was received by
      the NAT64, data can still flow in the connection, and the NAT64 is
      waiting for a matching TCP FIN in the opposite direction.

      V6 FIN RCV: An IPv6 packet containing a TCP FIN was received by
      the NAT64, data can still flow in the connection, and the NAT64 is
      waiting for a matching TCP FIN in the opposite direction.

      V6 FIN + V4 FIN RCV: Both an IPv4 packet containing a TCP FIN and
      an IPv6 packet containing an TCP FIN for this connection were
      received by the NAT64. The NAT64 keeps the connection state alive

and forwards packets in both directions for a short period of time
to allow remaining packets (in particular the ACKs) to be
delivered.

TRANS: The lifetime of the state for the connection is set to
TCP_TRANS minutes either because a packet containing a TCP RST was
received by the NAT64 for this connection or simply because the
lifetime of the connection has decreased and there are only
TCP_TRANS minutes left. The NAT64 will keep the state for the
connection for TCP_TRANS min. and if no other data packets for
that connection are received, the state for this connection is
then terminated.

3.5.2.2. State machine for TCP processing in the NAT64

The state machine used by the NAT64 for the TCP session processing is
depicted next. The described state machine handles all TCP segments
received through the IPv6 and IPv4 interface. There is one state
machine per TCP connection that is potentially established through
the NAT64. After bootstrapping of the NAT64 device, all TCP sessions
are in CLOSED state. As we mention above, the CLOSED state is a
fictional state when is no state for that particular connection in
the NAT64. It should be noted that there is one state machine per
connection, so only packets belonging to a given connection are
inputs to the state machine associated to that connection. In other
words, when in the state machine below we state that a packet is
received, it is implicit that the incoming 5-tuple of the data packet
matches to the one of the state machine.

A TCP segment with the SYN flag set that is received through the IPv6
interface is called a V6 SYN, similarly, V4 SYN, V4 FIN, V6 FIN, V6
FIN + V4 FIN, V6 RST and V4 RST.

```
                          +---------------------------+
                          |                           |
                          V                           |
           V6       +------+         V4                |
         +----SYN------|CLOSED|-----SYN------+          |
         |            +------+              |          |
         |               ^                  |          |
         |               |TCP_TRANS T.O.    |          |
         V               |                  V          |
     +-------+        +-------+         +-------+       |
     |V6 INIT|        | TRANS |         |V4 INIT|       |
     +-------+        +-------+         +-------+       |
         |               |   ^             |           |
         |         data pkt  |             |           |
         |               |   V4 or V6 RST  |           |
         |               |   TCP_EST T.O.  |           |
     V4 SYN              V   |          V6 SYN          |
         |        +--------------+        |             |
         +-------->|  ESTABLISHED |<---------+          |
                  +--------------+                      |
                     |          |                       |
                  V4 FIN      V6 FIN                     |
                     |          |                        |
                     V          V                        |
              +---------+   +----------+                  |
              | V4 FIN  |   |  V6 FIN  |                  |
              |  RCV    |   |   RCV    |                  |
              +---------+   +----------+                  |
                   |            |                         |
                V6 FIN       V4 FIN              TCP_TRANS |
                   |            |                   T.O.   |
                   V            V                      |   |
              +--------------------+                   |   |
              | V4 FIN + V6 FIN RCV |-------------------+   
              +--------------------+
```
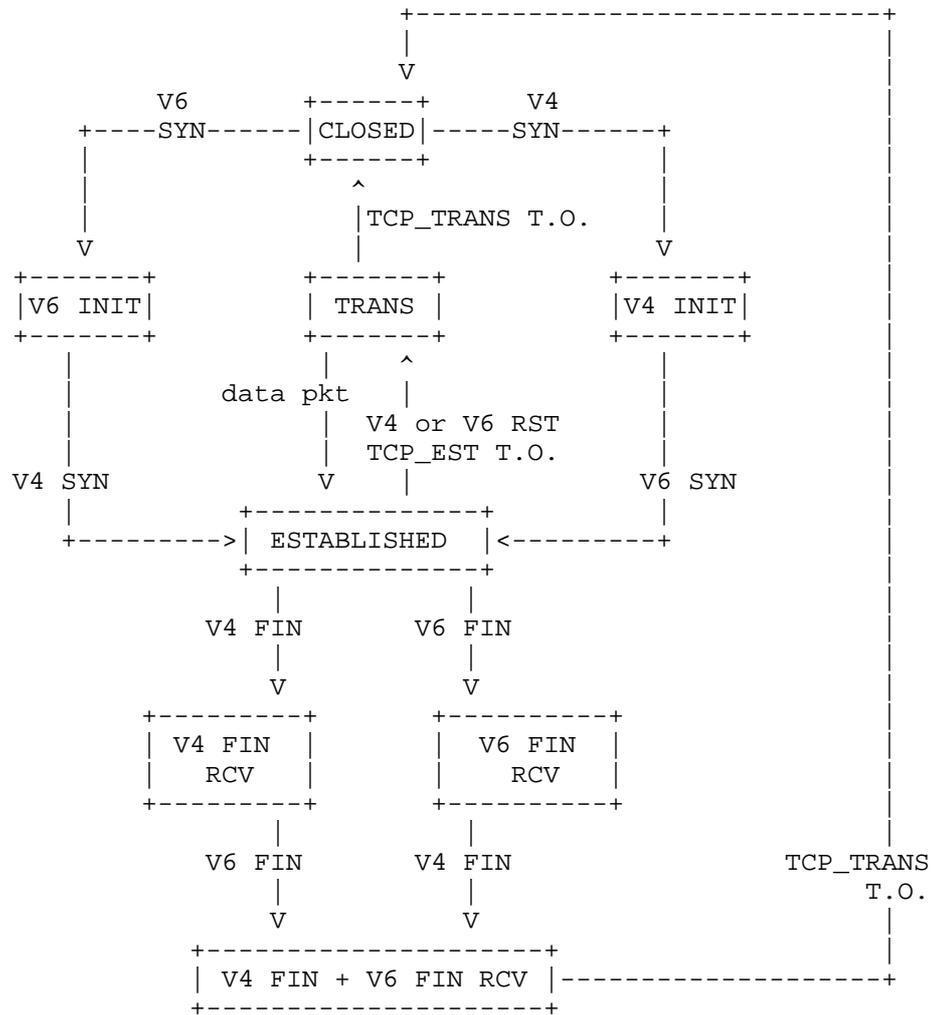
We next describe the state information and the transitions.

*** CLOSED ***

If a V6 SYN is received with an incoming tuple with source transport
address (X',x) and destination transport address (Y',y) (this is the
case of a TCP connection initiated from the IPv6 side), the
processing is as follows:

1. The NAT64 searches for a TCP BIB entry that matches the IPv6
    source transport address (X',x).

    If such an entry does not exist, the NAT64 tries to create a
    new BIB entry (if resources and policy permit). The BIB IPv6
    transport address is set to (X',x) (i.e., the source IPv6
    transport address of the packet). The BIB IPv4 transport
    address is set to an IPv4 transport address allocated using
    the rules defined in Section 3.5.2.3. The processing of the
    packet continues as described in bullet 2.

    If the entry already exists, then the processing continues as
    described in bullet 2.

2. Then the NAT64 tries to create a new TCP session entry in the TCP
    session table (if resources and policy permit). The information
    included in the session table is as follows:

    The STE source IPv6 transport address is set to (X',x) (i.e.
    the source transport address contained in the received V6 SYN
    packet,

    The STE destination IPv6 transport address is set to (Y',y)
    (i.e. the destination transport address contained in the
    received V6 SYN packet.

    The STE source IPv4 transport address is set to the BIB IPv4
    transport address of the corresponding TCP BIB entry.

    The STE destination IPv4 transport address contains the port y
    (i.e., the same port as the IPv6 destination transport
    address) and the IPv4 address that is algorithmically
    generated from the IPv6 destination address (i.e. Y') using
    the reverse algorithm as specified in Section 3.5.4.

    The lifetime of the TCP session table entry is set to at least
    to TCP_TRANS (the transitory connection idle timeout as
    defined in [RFC5382]).

3. The state of the session is moved to V6 INIT.

4. The NAT64 translates and forwards the packet as described in the
    following sections.

If a V4 SYN packet is received with an incoming tuple with source
IPv4 transport address (Y,y) and destination IPv4 transport address
(X,x) (this is the case of a TCP connection initiated from the IPv4
side), the processing is as follows:

If the security policy requires silently dropping externally initiated TCP connections, then the packet is silently discarded, else,

If the destination transport address contained in the incoming V4 SYN (i.e., X,x) is not in use in the TCP BIB, then:

The NAT64 tries to create a new session table entry in the TCP session table (if resources and policy permit), containing the following information:

+ The STE source IPv4 transport address is set to (X,x) (i.e. the destination transport address contained in the V4 SYN)

+ The STE destination IPv4 transport address is set to (Y,y) (i.e. the source transport address contained in the V4 SYN)

+ The STE transport IPv6 source address is left unspecified and may be populated by other protocols out of the scope of this specification.

+ The STE destination IPv6 transport address contains the port y (i.e. the same port as the STE destination IPv4 transport address) and the IPv6 representation of Y (i.e. the IPv4 address of the STE destination IPv4 transport address), generated using the algorithm described in Section 3.5.4.

The state is moved to V4 INIT.

The lifetime of the STE entry is set to TCP_INCOMING_SYN as per [RFC5382] and the packet is stored. The result is that the NAT64 will not drop the packet based on the filtering, nor create a BIB entry. Instead, the NAT64 will only create the session table entry and store the packet. The motivation for this is to support simultaneous open of TCP connections.

If the destination transport address contained in the incoming V4 SYN (i.e., X,x) is in use in the TCP BIB, then:

The NAT64 tries to create a new session table entry in the TCP session table (if resources and policy permit), containing the following information:

+ The STE source IPv4 transport address is set to (X,x) (i.e. the destination transport address contained in the V4 SYN)

+ The STE destination IPv4 transport address is set to (Y,y) (i.e. the source transport address contained in the V4 SYN)

+   The STE transport IPv6 source address is set to the IPv6
    transport address contained in the corresponding TCP BIB
    entry.

+   The STE destination IPv6 transport address contains the port
    y (i.e. the same port as the STE destination IPv4 transport
    address) and the IPv6 representation of Y (i.e. the IPv4
    address of the STE destination IPv4 transport address),
    generated using the algorithm described in Section 3.5.4.

The state is moved to V4 INIT.

If the NAT64 is performing Address-Dependent Filtering, the
lifetime of the STE entry is set to TCP_INCOMING_SYN as per
[RFC5382] and the packet is stored. The motivation for
creating the session table entry and storing the packet
(instead of simply dropping the packet based on the filtering)
is to support simultaneous open of TCP connections.

If the NAT64 is not performing Address-Dependent Filtering, the
lifetime of the STE is set to at least to TCP_TRANS (the
transitory connection idle timeout as defined in [RFC5382]) and
it translates and forwards the packet as described in the
following sections.

For any other packet belonging to this connection:

If there is a corresponding entry in the TCP BIB other packets
SHOULD be translated and forwarded if the security policy allows
to do so. The state remains unchanged.

If there is no corresponding entry in the TCP BIB the packet is
silently discarded.

*** V4 INIT ***

If a V6 SYN is received with incoming tuple with source transport
address (X',x) and destination transport address (Y',y). The
lifetime of the TCP session table entry is set to at least to the
maximum session lifetime. The value for the maximum session lifetime
MAY be configurable but it MUST NOT be less than TCP_EST (the
established connection idle timeout as defined in [RFC5382]). The
default value for the maximum session lifetime SHOULD be set to
TCP_EST. The packet is translated and forwarded. The state is moved
to ESTABLISHED.

If the lifetime expires, an ICMP Port Unreachable error (Type 3, Code
3) containing the IPv4 SYN packet stored is sent back to the source

of the v4 SYN, the session table entry is deleted and, the state is
moved to CLOSED.

For any other packet, other packets SHOULD be translated and
forwarded if the security policy allows to do so. The state remains
unchanged.

*** V6 INIT ***

If a V4 SYN is received (with or without the ACK flag set), with an
incoming tuple with source IPv4 transport address (Y,y) and
destination IPv4 transport address (X,x), then the state is moved to
ESTABLISHED. The lifetime of the TCP session table entry is set to
at least to the maximum session lifetime. The value for the maximum
session lifetime MAY be configurable but it MUST NOT be less than
TCP_EST (the established connection idle timeout as defined in
[RFC5382]). The default value for the maximum session lifetime
SHOULD be set to TCP_EST. The packet is translated and forwarded.

If the lifetime expires, the session table entry is deleted and the
state is moved to CLOSED.

If a V6 SYN packet is received, the packet is translated and
forwarded. The lifetime of the TCP session table entry is set to at
least to TCP_TRANS. The state remains unchanged.

For any other packet, other packets SHOULD be translated and
forwarded if the security policy allows to do so. The state remains
unchanged.

*** ESTABLISHED ***

If a V4 FIN packet is received, the packet is translated and
forwarded. The state is moved to V4 FIN RCV.

If a V6 FIN packet is received, the packet is translated and
forwarded. The state is moved to V6 FIN RCV.

If a V4 RST or a V6 RST packet is received, the packet is translated
and forwarded. The lifetime is set to TCP_TRANS and the state is
moved to TRANS. (Since the NAT64 is uncertain whether the peer will
accept the RST packet, instead of moving the state to CLOSED, it
moves to TRANS, which has a shorter lifetime. If no other packets
are received for this connection during the short timer, the NAT64
assumes that the peer has accepted the RST packet and moves to
CLOSED. If packets keep flowing, the NAT64 assumes that the peer has
not accepted the RST packet and moves back to the ESTABLISHED state.
This is described below in the TRANS state processing description.)

If any other packet is received, the packet is translated and
forwarded. The lifetime of the TCP session table entry is set to at
least to the maximum session lifetime. The value for the maximum
session lifetime MAY be configurable but it MUST NOT be less than
TCP_EST (the established connection idle timeout as defined in
[RFC5382]). The default value for the maximum session lifetime
SHOULD be set to TCP_EST. The state remains unchanged as
ESTABLISHED.

If the lifetime expires then the NAT64 SHOULD send a probe packet (as
defined next) to at least one of the endpoints of the TCP connection.
The probe packet is a TCP segment for the connection with no data.
The sequence number and the acknowledgment number are set to zero.
All flags but the ACK flag are reset. The state is moved to TRANS.

   Upon the reception of this probe packet, the endpoint will reply
   with an ACK containing the expected sequence number for that
   connection. It should be noted that, for an active connection,
   each of these probe packets will generate one packet from each end
   involved in the connection, since the reply of the first point to
   the probe packet will generate a reply from the other endpoint.

*** V4 FIN RCV ***

If a V6 FIN packet is received, the packet is translated and
forwarded. The lifetime is set to TCP_TRANS. The state is moved to
V6 FIN + V4 FIN RCV.

If any packet other than the V6 FIN is received, the packet is
translated and forwarded. The lifetime of the TCP session table
entry is set to at least to the maximum session lifetime. The value
for the maximum session lifetime MAY be configurable but it MUST NOT
be less than TCP_EST (the established connection idle timeout as
defined in [RFC5382]). The default value for the maximum session
lifetime SHOULD be set to TCP_EST. The state remains unchanged as V4
FIN RCV.

If the lifetime expires, the session table entry is deleted and the
state is moved to CLOSED.

*** V6 FIN RCV ***

If a V4 FIN packet is received, the packet is translated and
forwarded. The lifetime is set to TCT_TRANS. The state is moved to
V6 FIN + V4 FIN RCV.

If any packet other than the V4 FIN is received, the packet is
translated and forwarded. The lifetime of the TCP session table

entry is set to at least to the maximum session lifetime. The value
for the maximum session lifetime MAY be configurable but it MUST NOT
be less than TCP_EST (the established connection idle timeout as
defined in [RFC5382]). The default value for the maximum session
lifetime SHOULD be set to TCP_EST. The state remains unchanged as V6
FIN RCV.

If the lifetime expires, the session table entry is deleted and the
state is moved to CLOSED.

*** V6 FIN + V4 FIN RCV ***

All packets are translated and forwarded.

If the lifetime expires, the session table entry is deleted and the
state is moved to CLOSED.

*** TRANS ***

If a packet other than a RST packet is received, the lifetime of the
TCP session table entry is set to at least to the maximum session
lifetime. The value for the maximum session lifetime MAY be
configurable but it MUST NOT be less than TCP_EST (the established
connection idle timeout as defined in [RFC5382]). The default value
for the maximum session lifetime SHOULD be set to TCP_EST. The state
is moved to ESTABLISHED.

If the lifetime expires, the session table entry is deleted and the
state is moved to CLOSED.

3.5.2.3. Rules for allocation of IPv4 transport addresses for TCP

When a new TCP BIB entry is created for a source transport address of
(S',s), then the NAT64 allocates an IPv4 transport address for this
BIB entry as follows:

   If there exists some other BIB entry containing S' as the IPv6
   address and mapping it to some IPv4 address T, then T SHOULD be
   used as the IPv4 address. Otherwise, use any IPv4 address of the
   IPv4 pool assigned to the NAT64 to be used for translation.

   If the port s is in the Well-Known port range 0-1023, and the
   NAT64 has an available port t in the same port range, then the
   NAT64 SHOULD allocate the port t. If the NAT64 does not have a
   port available in the same range, the NAT64 MAY assign a port t
   from another range where it has an available port.

If the port s is in the range 1024-65535, and the NAT64 has an
available port t in the same port range, then the NAT64 SHOULD
allocate the port t. If the NAT64 does not have a port available
in the same range, the NAT64 MAY assign a port t from another
range where it has an available port.

In all cases, the allocated IPv4 transport address (T,t) MUST NOT
be in use in another entry in the same BIB, but MAY be in use in
the other BIB (referring to the UDP and TCP BIBs).

If it is not possible to allocate an appropriate IPv4 transport
address or create a BIB entry, then the packet is discarded. The
NAT64 SHOULD send an ICMPv6 Destination Unreachable/Address
unreachable (Code 3) message.

3.5.3. ICMP Query Session Handling

The following state information is stored for an ICMP Query session
in the ICMP Query session table:

Binding:(X',Y',i1) <--> (T,Z,i2)

Lifetime: a timer that tracks the remaining lifetime of the ICMP
Query session. When the timer expires, the session is deleted.
If all the ICMP Query sessions corresponding to a dynamically
created ICMP Query BIB entry are deleted, then the ICMP Query BIB
entry is also deleted.

An incoming ICMPv6 Informational packet with IPv6 source address X',
IPv6 destination address Y' and ICMPv6 Identifier i1, is processed as
follows:

If the local security policy determines that ICMPv6 Informational
packets are to be filtered, the packet is silently discarded.
Else, the NAT64 searches for an ICMP Query BIB entry that matches
the (X',i1) pair. If such entry does not exist, the NAT64 tries
to create a new entry (if resources and policy permit) with the
following data:

* The BIB IPv6 address is set to X' (i.e. the source IPv6 address
   of the IPv6 packet).

* The BIB ICMPv6 Identifier is set to i1 (i.e. the ICMPv6
   Identifier).

* If there exists another BIB entry containing the same IPv6
   address X' and mapping it to an IPv4 address T, then use T as
   the BIB IPv4 address for this new entry. Otherwise, use any

IPv4 address assigned to the IPv4 interface.

* As the BIB ICMPv4 Identifier use any available value i.e. any
  identifier value for which no other entry exists with the same
  (IPv4 address, ICMPv4 Identifier) pair.

The NAT64 searches for an ICMP query session table entry
corresponding to the incoming 3-tuple (X',Y',i1). If no such
entry is found, the NAT64 tries to create a new entry (if
resources and policy permit). The information included in the new
session table entry is as follows:

* The STE IPv6 source address is set to X' (i.e. the address
  contained in the received IPv6 packet),

* The STE IPv6 destination address is set to Y' (i.e. the address
  contained in the received IPv6 packet),

* The STE ICMPv6 Identifier is set to i1 (i.e. the identifier
  contained in the received IPv6 packet),

* The STE IPv4 source address is set to the IPv4 address
  contained in the corresponding BIB entry,

* The STE ICMPv4 Identifier is set to the IPv4 identifier
  contained in the corresponding BIB entry,

* The STE IPv4 destination address is algorithmically generated
  from Y' using the reverse algorithm as specified in
  Section 3.5.4.

The NAT64 sets (or resets) the timer in the session table entry to
the maximum session lifetime. By default, the maximum session
lifetime is ICMP_DEFAULT. The maximum lifetime value SHOULD be
configurable. The packet is translated and forwarded as described
in the following sections.

An incoming ICMPv4 Query packet with source IPv4 address Y,
destination IPv4 address X and ICMPv4 Identifier i2 is processed as
follows:

The NAT64 searches for an ICMP Query BIB entry that contains X as
IPv4 address and i2 as the ICMPv4 Identifier. If such an entry
does not exist, the packet is dropped. An ICMP error message MAY
be sent to the original sender of the packet. The ICMP error
message, if sent, has a type of 3 (Destination Unreachable).

If the NAT64 filters on its IPv4 interface, then the NAT64 checks
to see if the incoming packet is allowed according to the Address-
Dependent Filtering rule. To do this, it searches for a session
table entry with an STE source IPv4 address equal to X, an STE
ICMPv4 Identifier equal to i2 and a STE destination IPv4 address
equal to Y. If such an entry is found (there may be more than
one), packet processing continues. Otherwise, the packet is
discarded. If the packet is discarded, then an ICMP error message
MAY be sent to the original sender of the packet. The ICMP error
message, if sent, has a type of 3 (Destination Unreachable) and a
code of 13 (Communication Administratively Prohibited).

In case the packet is not discarded in the previous processing
steps (either because the NAT64 is not filtering or because the
packet is compliant with the Address-dependent Filtering rule),
then the NAT64 searches for a session table entry with an STE
source IPv4 address equal to X, an STE ICMPv4 Identifier equal to
i2 and a STE destination IPv4 address equal to Y. If no such entry
is found, the NAT64 tries to create a new entry (if resources and
policy permit) with the following information:

* The STE source IPv4 address is set to X,

* The STE ICMPv4 Identifier is set to i2,

* The STE destination IPv4 address is set to Y,

* The STE source IPv6 address is set to the IPv6 address of the
   corresponding BIB entry,

* The STE ICMPv6 Identifier is set to the ICMPv6 Identifier of
   the corresponding BIB entry, and,

* The STE destination IPv6 address is set to the IPv6
   representation of the IPv4 address of Y, generated using the
   algorithm described in Section 3.5.4.

* The NAT64 sets (or resets) the timer in the session table entry
   to the maximum session lifetime. By default, the maximum
   session lifetime is ICMP_DEFAULT. The maximum lifetime value
   SHOULD be configurable. The packet is translated and forwarded
   as described in the following sections.

3.5.4. Generation of the IPv6 Representations of IPv4 Addresses

   NAT64 supports multiple algorithms for the generation of the IPv6
   representation of an IPv4 address and vice-versa. The constraints
   imposed on the generation algorithms are the following:

The algorithm MUST be reversible, i.e. it MUST be possible to derive the original IPv4 address from the IPv6 representation.

The input for the algorithm MUST be limited to the IPv4 address, the IPv6 prefix (denoted Pref64::/n) used in the IPv6 representations and optionally a set of stable parameters that are configured in the NAT64 (such as fixed string to be used as a suffix).

If we note n the length of the prefix Pref64::/n, then n MUST be less or equal than 96. If a Pref64::/n is configured through any means in the NAT64 (such as manually configured, or other automatic means not specified in this document), the default algorithm MUST use this prefix. If no prefix is available, the algorithm SHOULD use the Well-Known Prefix (64: FF9B::/96) defined in [I-D.ietf-behave-address-format]

NAT64 MUST support the algorithm for generating IPv6 representations of IPv4 addresses defined in Section 2.1 of [I-D.ietf-behave-address-format]. The aforementioned algorithm SHOULD be used as default algorithm.

3.6. Computing the Outgoing Tuple

This step computes the outgoing tuple by translating the IP addresses and port numbers or ICMP Identifier in the incoming tuple.

In the text below, a reference to a BIB means either the TCP BIB, the UDP BIB or the ICMP Query BIB as appropriate.

NOTE: Not all addresses are translated using the BIB. BIB entries are used to translate IPv6 source transport addresses to IPv4 source transport addresses, and IPv4 destination transport addresses to IPv6 destination transport addresses. They are NOT used to translate IPv6 destination transport addresses to IPv4 destination transport addresses, nor to translate IPv4 source transport addresses to IPv6 source transport addresses. The latter cases are handled applying the algorithmic transformation described in Section 3.5.4. This distinction is important; without it, hairpinning doesn't work correctly.

3.6.1. Computing the Outgoing 5-tuple for TCP, UDP and for ICMP Error messages containing a TCP or UDP packets.

The transport protocol in the outgoing 5-tuple is always the same as that in the incoming 5-tuple.

When translating in the IPv6 --> IPv4 direction, let the source and

destination transport addresses in the incoming 5-tuple be (S',s) and
(D',d) respectively. The outgoing source transport address is
computed as follows: if the BIB contains a entry (S',s) <--> (T,t),
then the outgoing source transport address is (T,t).

The outgoing destination address is computed algorithmically from D'
using the address transformation described in Section 3.5.4.

When translating in the IPv4 --> IPv6 direction, let the source and
destination transport addresses in the incoming 5-tuple be (S,s) and
(D,d) respectively. The outgoing source transport address is
computed as follows:

   The outgoing source transport address is generated from S using
   the address transformation algorithm described in Section 3.5.4.

   The BIB table is searched for an entry (X',x) <--> (D,d), and if
   one is found, the outgoing destination transport address is set to
   (X',x).

3.6.2. Computing the Outgoing 3-tuple for ICMP Query Messages and for
        ICMP Error messages containing an ICMP Query.

When translating in the IPv6 --> IPv4 direction, let the source and
destination addresses in the incoming 3-tuple be S' and D'
respectively and the ICMPv6 Identifier be i1. The outgoing source
address is computed as follows: the BIB contains an entry (S',i1)
<--> (T,i2), then the outgoing source address is T and the ICMPv4
Identifier is i2.

The outgoing IPv4 destination address is computed algorithmically
from D' using the address transformation described in Section 3.5.4.

When translating in the IPv4 --> IPv6 direction, let the source and
destination addresses in the incoming 3-tuple be S and D respectively
and the ICMPv4 Identifier is i2. The outgoing source address is
generated from S using the address transformation algorithm described
in Section 3.5.4. The BIB is searched for an entry containing
(X',i1) <--> (D,i2) and if found the outgoing destination address is
X' and the outgoing ICMPv6 Identifier is i1.

3.7. Translating the Packet

   This step translates the packet from IPv6 to IPv4 or vice-versa.

   The translation of the packet is as specified in Section 3 and
   Section 4 of the IP/ICMP Translation Algorithm
   [I-D.ietf-behave-v6v4-xlate], with the following modifications:

o When translating an IP header (Sections 3.1 and 4.1), the source
   and destination IP address fields are set to the source and
   destination IP addresses from the outgoing tuple as determined in
   Section 3.6.

o When the protocol following the IP header is TCP or UDP, then the
   source and destination ports are modified to the source and
   destination ports from the outgoing 5-tuple. In addition, the TCP
   or UDP checksum must also be updated to reflect the translated
   addresses and ports; note that the TCP and UDP checksum covers the
   pseudo-header which contains the source and destination IP
   addresses. An algorithm for efficiently updating these checksums
   is described in [RFC3022].

o When the protocol following the IP header is ICMP and it is an
   ICMP Query message, the ICMP Identifier is set to the one from the
   outgoing 3-tuple as determined in Section 3.6.2.

o When the protocol following the IP header is ICMP and it is an
   ICMP error message, the source and destination transport addresses
   in the embedded packet are set to the destination and source
   transport addresses from the outgoing 5-tuple (note the swap of
   source and destination).

The size of outgoing packets as well and the potential need for
fragmentation is done according to the behavior defined in the IP/
ICMP Translation Algorithm [I-D.ietf-behave-v6v4-xlate]

3.8. Handling Hairpinning

If the destination IP address of the translated packet is an IPv4
address assigned to the NAT64 itself then the packet is a hairpin
packet. Hairpin packets are processed as follows:

o The outgoing 5-tuple becomes the incoming 5-tuple, and,

o the packet is treated as if it was received on the outgoing
   interface.

o Processing of the packet continues at step 2 - Filtering and
   updating binding and session information described in Section 3.5.

4. Protocol Constants

UDP_MIN 2 minutes (as defined in [RFC4787])

UDP_DEFAULT 5 minutes (as defined in [RFC4787])

    TCP_TRANS 4 minutes (as defined in [RFC5382])

    TCP_EST 2 hours (the minimum lifetime for an established TCP session
    defined in [RFC5382] is 2 hrs and 4 minutes, which is achieved adding
    the 2 hours with this timer and the 4 minutes with the TCP_TRANS
    timer)

    TCP_INCOMING_SYN 6 seconds (as defined in [RFC5382])

    FRAGMENT_MIN 2 seconds

    ICMP_DEFAULT 60 seconds (as defined in [RFC5508])

5. Security Considerations

5.1. Implications on end-to-end security

    Any protocols that protect IP header information are essentially
    incompatible with NAT64. This implies that end-to-end IPsec
    verification will fail when AH is used (both transport and tunnel
    mode) and when ESP is used in transport mode. This is inherent in
    any network-layer translation mechanism. End-to-end IPsec protection
    can be restored, using UDP encapsulation as described in [RFC3948].
    The actual extensions to support IPsec are out of the scope of this
    document.

5.2. Filtering

    NAT64 creates binding state using packets flowing from the IPv6 side
    to the IPv4 side. In accordance with the procedures defined in this
    document following the guidelines defined in [RFC4787] a NAT64 MUST
    offer "Endpoint-Independent Mapping". This means:

       for any IPv6 packet with source (S'1,s1) and destination (Pref64::
       D1,d1) that creates an external mapping to (S1,s1v4), (D1,d1), for
       any subsequent packet from (S'1,s1) to (Pref64::D2,d2) that
       creates an external mapping to (S2,s2v4), (D2,d2), within a given
       binding timer window,

       (S1,s1v4) = (S2,s2v4) for all values of D2,d2

    Implementations MAY also provide support for "Address-Dependent
    Mapping" as also defined in this document and following the
    guidelines defined in [RFC4787].

    The security properties however are determined by which packets the
    NAT64 filter allows in and which it does not. The security

properties are determined by the filtering behavior and filtering
configuration in the filtering portions of the NAT64, not by the
address mapping behavior. For example,

    Without filtering - When "Endpoint-Independent Mapping" is used in
    NAT64, once a binding is created in the IPv6 ---> IPv4 direction,
    packets from any node on the IPv4 side destined to the IPv6
    transport address will traverse the NAT64 gateway and be forwarded
    to the IPv6 transport address that created the binding. However,

    With filtering - When "Endpoint-Independent Mapping" is used in
    NAT64, once a binding is created in the IPv6 ---> IPv4 direction,
    packets from any node on the IPv4 side destined to the IPv6
    transport address will first be processed against the filtering
    rules. If the source IPv4 address is permitted, the packets will
    be forwarded to the IPv6 transport address. If the source IPv4
    address is explicitly denied -- or the default policy is to deny
    all addresses not explicitly permitted -- then the packet will be
    discarded. A dynamic filter may be employed where by the filter
    will only allow packets from the IPv4 address to which the
    original packet that created the binding was sent. This means
    that only the IPv4 addresses to which the IPv6 host has initiated
    connections will be able to reach the IPv6 transport address, and
    no others. This essentially narrows the effective operation of
    the NAT64 device to an "Address-Dependent Mapping" behavior,
    though not by its mapping behavior, but instead by its filtering
    behavior.

   As currently specified, the NAT64 only requires filtering traffic
   based on the 5-tuple. In some cases (e.g., statically configured
   mappings), this may make it easy for an attacker to guess. An
   attacker need not be able to guess other fields, e.g. the TCP
   sequence number, to get a packet through the NAT64. While such
   traffic might be dropped by the final destination, it does not
   provide additional mitigations against bandwidth/CPU attacks
   targeting the internal network. To avoid this type of abuse, a NAT64
   MAY keep track of the sequence number of TCP packets in order to
   verify that proper sequencing of exchanged segments, in particular,
   those of the SYNs and the FINs.

5.3. Attacks on NAT64

   The NAT64 device itself is a potential victim of different types of
   attacks. In particular, the NAT64 can be a victim of DoS attacks.
   The NAT64 device has a limited number of resources that can be
   consumed by attackers creating a DoS attack. The NAT64 has a limited
   number of IPv4 addresses that it uses to create the bindings. Even
   though the NAT64 performs address and port translation, it is

possible for an attacker to consume all the IPv4 transport addresses
by sending IPv6 packets with different source IPv6 transport
addresses. This attack can only be launched from the IPv6 side,
since IPv4 packets are not used to create binding state. DoS attacks
can also affect other limited resources available in the NAT64 such
as memory or link capacity. For instance, it is possible for an
attacker to launch a DoS attack on the memory of the NAT64 device by
sending fragments that the NAT64 will store for a given period. If
the number of fragments is high enough, the memory of the NAT64 could
be exhausted. Similarly, A DoS attack against the NAT64 can be
crafted by sending either V4 or V6 SYN packets that consume memory in
the form of session and/or binding table entries. In the case of
IPv4 SYNs the situation is aggravated by thee requirement to also
store the data packets for a given amount of time, requiring more
memory from the NAT64 device. NAT64 devices MUST implement proper
protection against such attacks, for instance allocating a limited
amount of memory for fragmented packet storage as specified in
Section 3.4.

Another consideration related to NAT64 resource depletion refers to
the preservation of binding state. Attackers may try to keep a
binding state alive forever by sending periodic packets that refresh
the state. In order to allow the NAT64 to defend against such
attacks, the NAT64 MAY choose not to extend the session entry
lifetime for a specific entry upon the reception of packets for that
entry through the external interface. As described in the Framework
document [I-D.ietf-behave-v6v4-framework], the NAT64 can be deployed
in multiple scenarios, some of which the Internet side is the IPv6
one and some of which the Internet side is the IPv4 one. It is then
important to properly set which is the Internet side of the NAT64 in
each specific configuration.

5.4. Avoiding hairpinning loops

If an IPv6-only client can guess the IPv4 binding address that will
be created, it can use the IPv6 representation of it as source
address for creating this binding. Then any packet sent to the
binding's IPv4 address could loop in the NAT64. This is prevented in
the current specification by filtering incoming packets containing
Pref64::/n in the source address as described next.

Consider the following example:

Suppose that the IPv4 pool is 192.0.2.0/24

Then the IPv6-only client sends this to NAT64:

Source: [Pref64::192.0.2.1]:500

Destination: whatever

The NAT64 allocates 192.0.2.1:500 as IPv4 binding address. Now anything sent to 192.0.2.1:500, be it a hairpinned IPv6 packet or an IPv4 packet, could loop.

It is not hard to guess the IPv4 address that will be allocated. First the attacker creates a binding and use (for example) STUN to learn its external IPv4 address. New bindings will always have this address. Then it uses a source port in the range 1-1023. This will increase the chances to 1/512 (since range and parity are preserved by NAT64 in UDP).

In order to address this vulnerability, the NAT64 MUST drop IPv6 packets whose source address is in Pref64::/n as defined in Section 3.5.

6. IANA Considerations

This document contains no actions for IANA.

7. Contributors

George Tsirtsis

Qualcomm

tsirtsis@googlemail.com

Greg Lebovitz

Juniper

gregory.ietf@gmail.com

Simon Parreault

Viagenie

simon.perreault@viagenie.ca

8. Acknowledgements

   Dave Thaler, Dan Wing, Alberto Garcia-Martinez, Reinaldo Penno,
   Ranjana Rao, Lars Eggert, Senthil Sivakumar, Zhen Cao, Xiangsong Cui,
   Mohamed Boucadair, Dong Zhang, Bryan Ford, Kentaro Ebisawa, Charles
   Perkins, Magnus Westerlund, Ed Jankiewicz, David Harrington, Peter
   McCann, Julien Laganier, Pekka Savola and Joao Damas reviewed the
   document and provided useful comments to improve it.

   The content of the draft was improved thanks to discussions with
   Christian Huitema, Fred Baker and Jari Arkko.

   Marcelo Bagnulo and Iljitsch van Beijnum are partly funded by
   Trilogy, a research project supported by the European Commission
   under its Seventh Framework Program.

9. References

9.1. Normative References

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC4787]   Audet, F. and C. Jennings, "Network Address Translation
               (NAT) Behavioral Requirements for Unicast UDP", BCP 127,
               RFC 4787, January 2007.

   [RFC5382]   Guha, S., Biswas, K., Ford, B., Sivakumar, S., and P.
               Srisuresh, "NAT Behavioral Requirements for TCP", BCP 142,
               RFC 5382, October 2008.

   [RFC5508]   Srisuresh, P., Ford, B., Sivakumar, S., and S. Guha, "NAT
               Behavioral Requirements for ICMP", BCP 148, RFC 5508,
               April 2009.

   [I-D.ietf-behave-v6v4-xlate]
               Li, X., Bao, C., and F. Baker, "IP/ICMP Translation
               Algorithm", draft-ietf-behave-v6v4-xlate-20 (work in
               progress), May 2010.

   [I-D.ietf-behave-address-format]
               Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X.
               Li, "IPv6 Addressing of IPv4/IPv6 Translators",
               draft-ietf-behave-address-format-09 (work in progress),
               July 2010.

   [RFC4443]   Conta, A., Deering, S., and M. Gupta, "Internet Control

Message Protocol (ICMPv6) for the Internet Protocol
Version 6 (IPv6) Specification", RFC 4443, March 2006.

9.2. Informative References

[I-D.ietf-behave-dns64]
          Bagnulo, M., Sullivan, A., Matthews, P., and I. Beijnum,
          "DNS64: DNS extensions for Network Address Translation
          from IPv6 Clients to IPv4 Servers",
          draft-ietf-behave-dns64-10 (work in progress), July 2010.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7,
          RFC 793, September 1981.

[RFC1858] Ziemba, G., Reed, D., and P. Traina, "Security
          Considerations for IP Fragment Filtering", RFC 1858,
          October 1995.

[RFC3128] Miller, I., "Protection Against a Variant of the Tiny
          Fragment Attack (RFC 1858)", RFC 3128, June 2001.

[RFC3022] Srisuresh, P. and K. Egevang, "Traditional IP Network
          Address Translator (Traditional NAT)", RFC 3022,
          January 2001.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment
          (ICE): A Protocol for Network Address Translator (NAT)
          Traversal for Offer/Answer Protocols", RFC 5245,
          April 2010.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly
          Errors at High Data Rates", RFC 4963, July 2007.

[I-D.ietf-behave-v6v4-framework]
          Baker, F., Li, X., Bao, C., and K. Yin, "Framework for
          IPv4/IPv6 Translation",
          draft-ietf-behave-v6v4-framework-09 (work in progress),
          May 2010.

[RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M.
          Stenberg, "UDP Encapsulation of IPsec ESP Packets",
          RFC 3948, January 2005.

Authors' Addresses

   Marcelo Bagnulo
   UC3M
   Av. Universidad 30
   Leganes, Madrid   28911
   Spain

   Phone: +34-91-6249500
   Fax:
   Email: marcelo@it.uc3m.es
   URI: http://www.it.uc3m.es/marcelo

   Philip Matthews
   Alcatel-Lucent
   600 March Road
   Ottawa, Ontario
   Canada

   Phone: +1 613-592-4343 x224
   Fax:
   Email: philip_matthews@magma.ca
   URI:

   Iljitsch van Beijnum
   IMDEA Networks
   Avda. del Mar Mediterraneo, 22
   Leganes, Madrid   28918
   Spain

   Email: iljitsch@muada.com

Internet Engineering Task Force   I. Yamagata
Internet-Draft   Y. Shirasaki
Intended status: Informational   NTT Communications
Expires: January 5, 2013   A. Nakagawa
                                            Japan Internet Exchange (JPIX)
                                                         J. Yamaguchi
                                                       Fiber 26 Network
                                                            H. Ashida
                                                     IS Consulting G.K.
                                                         July 4, 2012

                                 NAT444
                       draft-shirasaki-nat444-06

Abstract

   This document describes one of the network models that are designed
   for smooth transition to IPv6. It is called NAT444 model. NAT444
   model is composed of IPv6, and IPv4 with Carrier Grade (CGN).

   NAT444 is the only scheme not to require replacing Customer Premises
   Equipment (CPE) even if IPv4 address exhausted. But it must be noted
   that NAT444 has serious restrictions i.e. it limits the number of
   sessions per CPE so that rich applications such as AJAX and RSS feed
   cannot work well.

   Therefore, IPv6 which is free from such a difficulty has to be
   introduced into the network at the same time. In other words, NAT444
   is just a tool to make IPv6 transition easy to be swallowed. It is
   designed for the days IPv4 and IPv6 co-existence.

Copyright Notice

Table of Contents

1. Introduction

   The only permanent solution of the IPv4 address exhaustion is to
   deploy IPv6. Now, just before the exhaustion, it's time to make a
   transition to IPv6.

   After the exhaustion, unless ISP takes any action, end users will not
   be able to get IPv4 address.

   The servers that have only IPv4 address will continue to exist on the
   Internet after the IPv4 address exhaustion. In this situation, IPv6
   only hosts cannot reach IPv4 only hosts.

   This document explains NAT444 model that bridges the gap between the
   coming IPv6 Internet and the present IPv4 Internet.

2. Definition of NAT444 Model

   NAT444 Model is a network model that uses two Network Address and
   Port Translators (NAPTs) with three types of IPv4 address blocks.

   The first NAPT is in CPE, and the second NAPT is in Carrier Grade NAT
   (CGN) [I-D.ietf-behave-lsn-requirements]. CGN is supposed to be
   installed in the ISP's network.

```
      (The IPv4 Internet)     (The IPv6 Internet)
               |                    |
             +---------+            |
IPv4 Global Address    |            |
             +--------+--------+    |
             |      CGN        |    |
             +--------+--------+    |
                 IPv4 |            | IPv6
                      +------------+
             Dual Stack |
        +---------------+---------------+
        |  IPv4 NAT/IPv6 Dual Stack CPE  |
        +---------------+---------------+
IPv4 Private Address / |
IPv6 Dual Stack        |
             +----------+------------+
             |IPv4/IPv6 Dual Stack host|
             +----------------------+
```

   The first IPv4 address block is Private Address [RFC1918] inside CPE.
   The second one is an IPv4 Address block between CPEs and CGN. The
   third one is IPv4 Global Addresses that is outside CGN. The ISPs

using NAT444 provide IPv6 connectivity by dual stack model.

3. Behavior of NAT444 Model

   The IPv6 packets from the host reach the IPv6 Internet without using
   NAT functionality.

   The following figure shows the behavior of the IPv4 packet from the
   host to the IPv4 server via two NATs. The first NAT in CPE
   overwrites the Source IP Address and Source Port from 10.0.0.2:tt to
   w.w.w.w:uu. Then the second NAT in CGN overwrites them from
   w.w.w.w:uu to y.y.y.y:vv. Destination IP Address and Port are not
   overwritten.

```
                   +-------------+
   (Port=80)       | IPv4 Server |   ^
       x.x.x.x-> +------+------+    :
                        |          :
   IPv4 Global Address  |          :
                        |          :
             (The IPv4 Internet):(Dst=x.x.x.x:80/Src=y.y.y.y:vv)
                        |          :
   IPv4 Global Address  |          :
                        |          :
       y.y.y.y->  +----+----+      :
         (Port=vv) |   CGN   |     ^
       z.z.z.z->  +----+----+      :
                        |          :
          IPv4 Address  |          :(Dst=x.x.x.x:80/Src=w.w.w.w:uu)
                        |          :
     w.w.w.w-> +-------+-------+   :
      (Port=uu) | IPv4 NAT CPE  |  ^
      10.0.0.1-> +-------+-------+  :
                        |          :
   IPv4 Private Address |          :
                        |          :
      10.0.0.2->  +----+----+      :(Dst=x.x.x.x:80/Src=10.0.0.2:tt)
        (Port=tt) |IPv4 Host|
                  +---------+
```

   The following figure explains the behavior of returning IPv4 packet
   via two NATs. The first NAT in CGN overwrites the Destination IP
   Address and Port Number from y.y.y.y:vv to w.w.w.w:uu. Then the
   second NAT in CPE overwrites them from w.w.w.w:u to 10.0.0.2:tt.

```
                    +-------------+
         (Port=80) | IPv4 Server |  :
         x.x.x.x-> +------+------+  :
                          |         :
    IPv4 Global Address   |         :
                          |         :
              (The IPv4 Internet):(Dst=y.y.y.y:vv/Src=x.x.x.x:80)
                          |         :
    IPv4 Global Address   |         :
                          |         :
          y.y.y.y-> +----+----+     :
          (Port=vv) |   CGN   |     v
          z.z.z.z-> +----+----+     :
                          |         :
          IPv4 Address    |         :(Dst=w.w.w.w:uu/Src=x.x.x.x:80)
                          |         :
      w.w.w.w-> +-------+-------+   :
      (Port=uu) | IPv4 NAT CPE  |   v
      10.0.0.1-> +-------+-------+  :
                          |         :
    IPv4 Private Address  |         :(Dst=10.0.0.2:tt/Src=x.x.x.x:80)
                          |         :
         10.0.0.2-> +----+----+     :
         (Port=tt) |IPv4 Host|      v
                   +---------+
```

4. Pros and Cons of NAT444 Model

4.1. Pros of NAT444 Model

   This network model has following advantages.

   - This is the only network model that doesn't require replacing CPEs
   those are owned by customers.
   - This network model is composed of the present technology.
   - This network model doesn't require address family translation.
   - This network model doesn't require DNS rewriting.
   - This network model doesn't require additional fragment for the
   packets because it doesn't use tunneling technology.

4.2. Cons of NAT444 Model

   This network model has some technical restrictions.

   - Some application such as SIP requires special treatment, because IP
   address is written in the payload of the packet. Special treatment
   means application itself aware double NAPT or both of two NAPTs

support inspecting and rewriting the packets.
- Because both IPv4 route and IPv6 route exist, it doubles the number
of IGP route inside the CGN.
- UPnP doesn't work with double NAPTs.

5. Acknowledgements

   Thanks for the input and review by Shin Miyakawa, Shirou Niinobe,
   Takeshi Tomochika, Tomohiro Fujisaki, Dai Nishino, JP address
   community members, AP address community members and JPNIC members.

6. IANA Considerations

   There are no IANA considerations.

7. Security Considerations

   Each customer inside a CGN looks using the same Global Address from
   outside an ISP. In case of incidents, the ISP must have the function
   to trace back the record of each customer's access without using only
   IP address.

   If a Global Address of the CGN is listed on the blacklist, other
   customers who share the same address could be affected.

8. References

8.1. Normative References

   [RFC1918] Rekhter, Y., Moskowitz, R., Karrenberg, D., Groot, G., and
              E. Lear, "Address Allocation for Private Internets",
              BCP 5, RFC 1918, February 1996.

   [RFC4925] Li, X., Dawkins, S., Ward, D., and A. Durand, "Softwire
              Problem Statement", RFC 4925, July 2007.

   [I-D.ietf-behave-lsn-requirements]
              Perreault, S., Yamagata, I., Miyakawa, S., Nakagawa, A.,
              and H. Ashida, "Common requirements for Carrier Grade NATs
              (CGNs)", draft-ietf-behave-lsn-requirements-07 (work in
              progress), June 2012.

8.2. Informative References

   [I-D.shirasaki-isp-shared-addr]
              Yamagata, I., Miyakawa, S., Nakagawa, A., Yamaguchi, J.,
              and H. Ashida, "ISP Shared Address",
              draft-shirasaki-isp-shared-addr-07 (work in progress),
              January 2012.
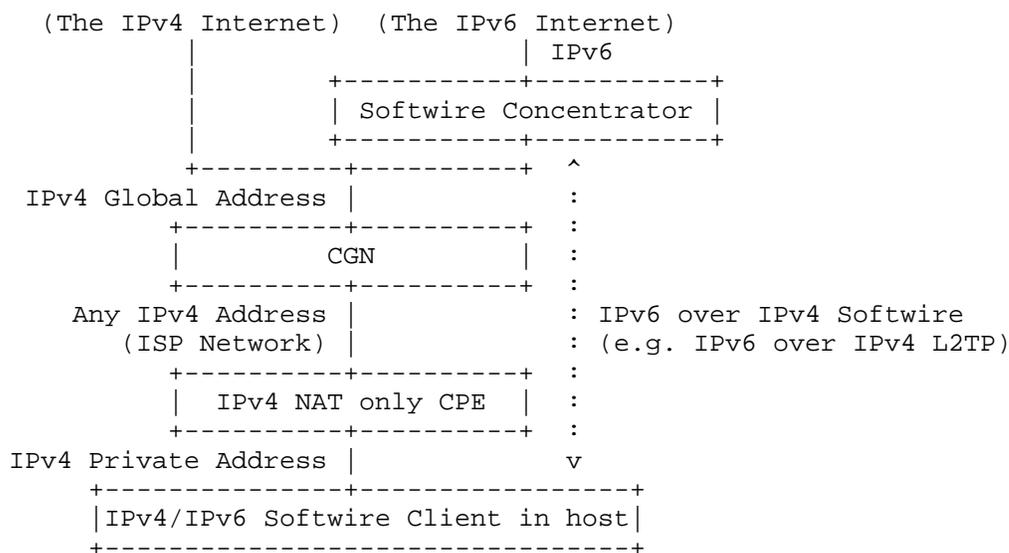
   [I-D.shirasaki-nat444-isp-shared-addr]
              Yamaguchi, J., Shirasaki, Y., Miyakawa, S., Nakagawa, A.,
              and H. Ashida, "NAT444 addressing models",
              draft-shirasaki-nat444-isp-shared-addr-07 (work in
              progress), January 2012.

Appendix A. Example IPv6 Transition Scenario

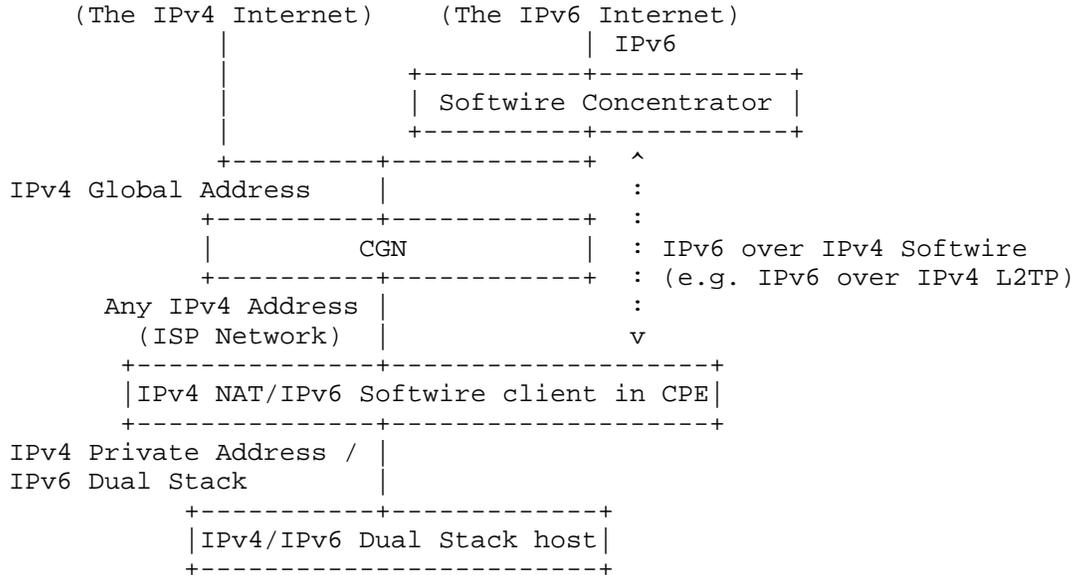   The steps of IPv6 transition are as follows.

   Step 1: Enabling softwire client in host

   ISP provides IPv6 connectivity to customers with softwire [RFC4925].
   ISP installs CGN and softwire concentrator in its network. A
   softwire client in host connects to the IPv6 internet via ISP's
   concentrator. ISP can use existing IPv4 equipments. Customers can
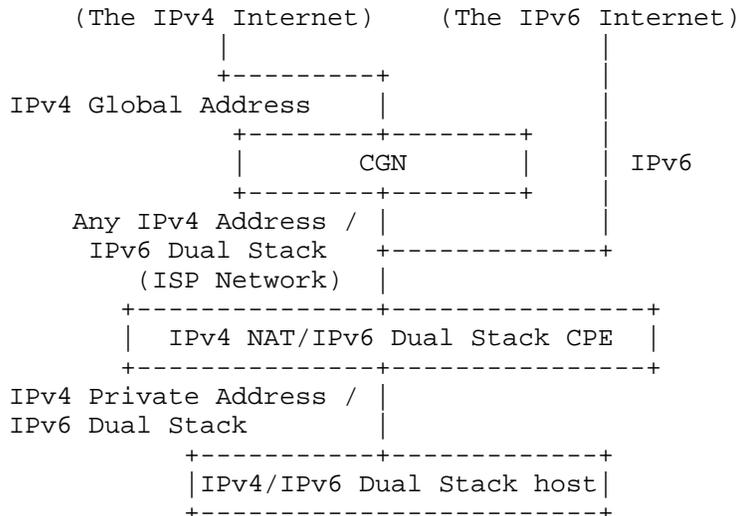   just use existing CPE.

```
   (The IPv4 Internet)  (The IPv6 Internet)
            |                        | IPv6
            |          +-----------+-----------+
            |          | Softwire Concentrator |
            |          +-----------+-----------+
         +---------+---------+  ^
 IPv4 Global Address |          :
         +---------+---------+  :
            |       CGN       | :
         +---------+---------+  :
     Any IPv4 Address |         : IPv6 over IPv4 Softwire
        (ISP Network) |         : (e.g. IPv6 over IPv4 L2TP)
         +---------+---------+  :
            | IPv4 NAT only CPE | :
         +---------+---------+  :
 IPv4 Private Address |         v
       +--------------+----------------+
       |IPv4/IPv6 Softwire Client in host|
       +-------------------------------+
```

   Step 2: Enabling softwire client in CPE

A customer enables softwire client in CPE. A softwire client in CPE
connects to the IPv6 internet via ISP's concentrator. A Customer's
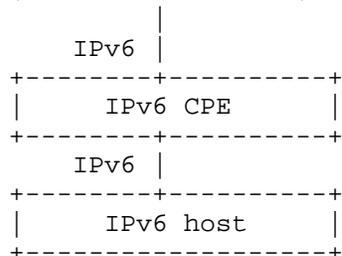network is now dual stack.

```
     (The IPv4 Internet)    (The IPv6 Internet)
              |                      | IPv6
              |           +----------+-----------+
              |           | Softwire Concentrator |
              |           +----------+-----------+
         +---------+-----------+  ^
IPv4 Global Address      |       :
         +---------+-----------+  :
              |       CGN       | : IPv6 over IPv4 Softwire
         +---------+-----------+  : (e.g. IPv6 over IPv4 L2TP)
     Any IPv4 Address |           :
       (ISP Network)  |           v
       +--------------+------------------+
       |IPv4 NAT/IPv6 Softwire client in CPE|
       +--------------+------------------+
IPv4 Private Address / |
IPv6 Dual Stack        |
          +----------+------------+
          |IPv4/IPv6 Dual Stack host|
          +-----------------------+
```

Step 3: Moving on to dual stack

ISP provides dual stack access to CPE. A CPE uplink is now dual
stack.

```
     (The IPv4 Internet)    (The IPv6 Internet)
              |                      |
          +---------+                |
IPv4 Global Address      |           |
         +--------+--------+          |
              |       CGN    |        | IPv6
         +--------+--------+          |
     Any IPv4 Address / |            |
      IPv6 Dual Stack  +-------------+
        (ISP Network)  |
       +--------------+---------------+
       |  IPv4 NAT/IPv6 Dual Stack CPE  |
       +--------------+---------------+
IPv4 Private Address / |
IPv6 Dual Stack        |
          +----------+------------+
          |IPv4/IPv6 Dual Stack host|
          +-----------------------+
```

    Step 4: Moving on to pure IPv6

    IPv6 transition completes.

    (The IPv6 Internet)
              |
        IPv6 |
    +--------+----------+
    | IPv6 CPE |
    +--------+----------+
        IPv6 |
    +--------+----------+
    | IPv6 host |
    +------------------+

Authors' Addresses

    Ikuhei Yamagata
    NTT Communications Corporation
    Granpark Tower 17F, 3-4-1 Shibaura, Minato-ku
    Tokyo 108-8118
    Japan

    Phone: +81 3 6733 8671
    Email: ikuhei@nttv6.jp

    Yasuhiro Shirasaki
    NTT Communications Corporation
    NTT Hibiya Bldg. 7F, 1-1-6 Uchisaiwai-cho, Chiyoda-ku
    Tokyo 100-8019
    Japan

    Phone: +81 3 6700 8530
    Email: yasuhiro@nttv6.jp

    Akira Nakagawa
    Japan Internet Exchange Co., Ltd. (JPIX)
    Otemachi Building 21F, 1-8-1 Otemachi, Chiyoda-ku
    Tokyo 100-0004
    Japan

    Phone: +81 90 9242 2717
    Email: a-nakagawa@jpix.ad.jp

Jiro Yamaguchi
Fiber 26 Network Inc.
Haraguchi bldg., 5F, 3-11-4 Kanda Jinbo-cho, Chiyoda-ku
Tokyo 101-0051
Japan

Phone: +81 50 3463 6109
Email: jiro-y@f26n.jp

Hiroyuki Ashida
IS Consulting G.K.
12-17 Odenma-cho, Nihonbashi, Chuo-ku
Tokyo 103-0011
Japan

Email: assie@hir.jp

Network Working Group   S. Venaas
Internet-Draft   cisco Systems
Intended status: Informational   X. Li
Expires: December 23, 2011   C. Bao
                                                  CERNET Center/Tsinghua
                                                             University
                                                          June 21, 2011

                 Framework for IPv4/IPv6 Multicast Translation
                  draft-venaas-behave-v4v6mc-framework-03.txt

Abstract

   This draft describes how IPv4/IPv6 multicast translation may be used
   in various scenarios and attempts to be a framework for possible
   solutions. This can be seen as a companion document to the document
   "Framework for IPv4/IPv6 Translation" by Baker et al. When
   considering scenarios and solutions for unicast translation, one
   should also see how they may be extended to provide multicast
   translation.

Status of this Memo

Copyright Notice

carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction

   There will be a long period of time where IPv4 and IPv6 systems and
   networks need to coexist. There are various solutions for how this
   can be done for unicast, some of which are based on translation. The
   document [RFC6144] discusses the needs and provides a framework for
   unicast translation for various scenarios. Here we discuss the need
   for multicast translation for those scenarios.

   For unicast the problem is basically how two hosts can communicate
   when they are not able to use the same IP protocol. For multicast we
   can restrict ourselves to looking at how a single source can
   efficiently send to multiple receivers. When using a single IP
   protocol one builds a multicast distribution tree from the source to
   the receivers, and independent of the number of receivers, one sends
   each data packet only once on each link. We wish to maintain the
   same characteristics when there are different IP protocols used.
   That is, when the nodes of the tree (source, receivers and routers)
   cannot all use the same IP protocol. In general there may be
   multiple sources sending to a multicast group, but that can be
   thought of as separate trees, one per source. We will focus on the
   case where the source and the receivers cannot all use the same IP
   protocol. If the issue is the network in between, encapsulation may
   be a better alternative. Note that if the source supports both IPv4
   and IPv6, then one alternative could be for the source to send two
   streams. This need not be the same host. There could be two
   different hosts, and in different locations/networks, sending the
   same content.

2. Translation scenarios

   We will consider six different translation scenarios. For each of
   the scenarios we will look at how host in one network can receive
   multicast from a source in another network. For unicast one might
   consider the following six scenarios as described in [RFC6144]:

   Scenario 1: An IPv6 network to the IPv4 Internet

   Scenario 2: The IPv4 Internet to an IPv6 network

   Scenario 3: The IPv6 Internet to an IPv4 network

   Scenario 4: An IPv4 network to the IPv6 Internet

   Scneario 5: An IPv6 network to an IPv4 network

   Scenario 6: An IPv4 network to an IPv6 network

   We have intentionally left out how one might connect the entire IPv4
   Internet with the entire IPv6 Internet. In these scenarios one would
   look at how a host in one network initiates a uni- or bi-directional
   flow to another network. The initiator needs to somehow know which
   address to send the initial packet to, and the initial packet gets
   translated before reaching its destination.

   For unicast translation it is quite natural to talk about networks
   and the Internet. For multicast this is not so clear, since there is
   limited use of multicast on the Internet. Certain parts of the
   Internet, e.g. academic and research networks and the links
   connecting them do carry multicast though. Also, the challenges and
   ideas described here regarding the Internet, also applies in other
   cases where there are multiple connected networks exchanging
   multicast.

   For multicast one generally need a receiver to signal the group (and
   sometimes also the source) it wants to receive from. The signalling
   generally goes hop-by-hop towards the source to build multicast
   forwarding state that later is used to forward multicast in the
   reverse direction. This means that for the receiving host to receive
   multicast, it must first somehow know which group (and possibly
   source) it should signal that it wants to receive. These signals
   would then probably go hop-by-hop to a translator, and then the
   translated signalling would go hop-by-hop from the translator to the
   source. Note that this description is correct for SSM (source-
   specific multicast), but is in reality more complex for ASM (any-
   source multicast). An anology to unicast might perhaps be TCP
   streaming where a SYN is sent from the host that wants to receive the

stream to the source of the stream. Then the application data flows
in the reverse direction of the initial signal. Hence we argue that
the above unicast scenarios correspond to the following multicast
scenarios, respectively:

Scenario 1: An IPv6 network receiving multicast from the IPv4
Internet

Scenario 2: The IPv4 Internet receiving multicast from an IPv6
network

Scneario 3: The IPv6 Internet receiving multicast from an IPv4
network

Scenario 4: An IPv4 network receiving multicast from the IPv6
Internet

Scenario 5: An IPv6 network receiving multicast from an IPv4 network

Scenario 6: An IPv4 network receiving multicast from an IPv6 network

2.1. Scenario 1: An IPv6 network receiving multicast from the IPv4
      Internet

Here we have a network, say ISP or enterprise, that for some reason
is IPv6-only, but the hosts in the IPv6-only network should be able
to receive multicast from sources in the IPv4 internet. The unicast
equivalent is "IPv6 network to the IPv4 Internet".

This is simple because the global IPv4 address space can be embedded
into IPv6 [RFC6052]. Unicast addresses according to the unicast
translation in use. For multicast one may embed all IPv4 multicast
addresses inside a single IPv6 multicast prefix. Or one may have
multiple embeddings to allow for appropriate mapping of scopes and
ASM versus SSM. Using this embedding, the IPv6 host (or an
application running on the host) can send IPv6 MLD reports for IPv6
groups (and if SSM, also sources) that specify which IPv4 source and
groups that it wants to receive. The usual IPv6 state (including MLD
and possibly PIM) needs to be created. If PIM is involved we need to
use RPF to set up the tree and accept data, so the source addresses
must be routed towards some translation device. This is likely to be
the same device that would do the unicast translation. The
translation device can in this case be completely stateless. There
is some multicast state, but that is similar to the state in a
multicast router when translation is not performed. Basically if the
translator receives MLD or PIM messages asking for a specific group
(or source and group), it uses these mappings to find out which IPv4
group (or source and group) it needs to send IGMP or PIM messages

for. This is no different than multicast in general, except for the
translation. Whenever the translator receives data from the IPv4
source, it checks if it has anyone interested in the respective IPv6
group (or source and group), and if so, translates and forwards the
data packets.

IPv6 applications need to somehow learn which IPv6 group (or source
and group) to join. This is further discussed in Section 3.4.

2.2. Scenario 2: The IPv4 Internet receiving multicast from an IPv6
      network

Here we will consider an IPv6 network connected to the IPv4 internet,
and how any IPv4 host may receive multicast from a source in the IPv6
network. The unicast equivalent is "the IPv4 Internet to an IPv6
network".

This is difficult since the IPv6 multicast address space cannot be
embedded into IPv4. Indeed this case has many similarities with how
IPv4 networks can receive from the IPv6 Internet. See scenario (4),
Section 2.4. However, in this case, all IPv4 hosts on the Internet
should use the same mapping, and it might make sense to have
additional requirements on the IPv6 network, rather than to add
requirements for the IPv4 Internet.

One solution here might be for the IPv6 source application to somehow
register with the translator to set up a mapping and receive an IPv4
address. The application could then possibly send SDP that includes
both its IPv6 source and group, and the IPv4 source and group it got
from the translator. Of course the signalling could also be done by
manually adding a static mapping to the translator and specifying
that address to the application. If instead we were to do signalling
on the IPv4 side, then an IPv4 receiver would probably need a
mechanism for finding an IPv4 address of the translator for a given
IPv6 group. The IPv4 address could perhaps be embedded in the IPv6
group address? Or with say SDP there could be a way of specifying
the IPv4 translator address. The IPv4 host could then communicate
with the translator to establish a mapping (unless one exists) and
learn which IPv4 group to join.

The best alternative might be to restrict the IPv6 multicast groups
that should be accessible on the IPv4 internet to a certain IPv6
prefix. This may allow stateless translation. This could also be
used in the reverse direction, for an IPv6 host to receive from an
IPv4 source. Or in other words, the same mapping can be used in both
directions. This has similarities with IVI [I-D.xli-behave-ivi],
[RFC6145], [RFC6052] and also [I-D.venaas-behave-mcast46]. By using
IVI source addresses (IPv4-translatable addresses) and a similar

technique for multicast addresses, the correct IPv4 source and group addresses can be derived from those. This method has many benefits, the main issue is that it cannot work for arbitrary IPv6 multicast addresses.

2.3. Scenario 3: The IPv6 Internet receiving multicast from an IPv4 network

We here consider the case where the Internet is IPv6, but there is some network of perhaps legacy IPv4 hosts that is IPv4-only. We want any IPv6 host on the Internet to be able to receive multicast from an IPv4 source. The unicast equivalent is "the IPv6 Internet to an IPv4 network".

This scenario can be solved using the same techniques as in Scenario 1, Section 2.1. There may however be differences regarding exactly which mappings are used and how applications may become aware of them. To obtain full benefit of multicast, all IPv6 hosts need to use the same mappings.

2.4. Scenario 4: An IPv4 network receiving multicast from the IPv6 Internet

Here we consider how an IPv4-only host in an IPv4 network may receive from an IPv6 multicast sender on the Internet. The unicast equivalent is "an IPv4 network to the IPv6 Internet".

For dual-stack hosts in an IPv4 network one should consider tunneling. This is difficult since we cannot embed the entire IPv6 space into IPv4. One might consider some of the techniques from scenario (2), Section 2.2. That scenario is however much easier since one may restrict which IPv6 groups are used and there is a limited number of sources.

For unicast one might use a DNS-ALG for this, where the ALG would instantiate translator mappings as needed. This is the technique used in NAT-PT [RFC2766], which was deprecated by [RFC4966].

However, for multicast one generally does not use DNS. One could consider doing the same with an ALG for some other protocol. E.g. translate addresses in SDP files when they pass the translator, or in any other protocol that might transfer multicast addresses. This would be very complicated and not recommended.

Rather than using an ALG that translates addresses in application protocol payload, one could consider new signalling mechanisms for more explicit signalling. The additional signalling could be either on the IPv6 or the IPv4 side. It may however not be a good idea to

require additional behavior by host and applications on the IPv6
Internet to accomodate legacy IPv4 networks. Also, since one may not
be able to provide unique IPv4 multicast addresses for all the IPv6
multicast groups that are in use, it makes more sense that the
mappings are done locally in each of the IPv4 networks, where IPv4
multicast addresses might be assigned on-demand. An IPv4 receiver
might somehow request an IPv4 mapping for an IPv6 group (and possibly
source). This creates a mapping in the translator so that when the
IPv4 receiver joins the IPv4 group, the translator knows which IPv6
group (and possibly source) to translate it into. Of course the
signalling could also be done manually by adding a static mapping to
the translator and somehow specifying the right IPv4 address to the
application.

2.5. Scenario 5: An IPv6 network receiving multicast from an IPv4
      network

   In this scenario we consider IPv4 and IPv6 networks belonging to the
   same organization. The unicast equivalent is "an IPv6 network to an
   IPv4 network".

   We would like any IPv6 host to receive from any IPv4 sources. Here
   one can use the same techniques as for an IPv6 network receiving from
   the IPv4 internet. It is really a special case of scenario (1),
   Section 2.1.

   The fact that the number of hosts are limited and that there is
   common management might simplify things. Due to the limited scale,
   one could perhaps just manually configure all the static mappings
   needed in the translator and manually create the necessary
   announcements or in some cases have the applications create the
   necessary announcements. But it might be better to use a stateless
   approach where IPv4 unicast and multicast addresses are embedded into
   IPv6. Like IVI [I-D.xli-behave-ivi], or [I-D.venaas-behave-mcast46].

2.6. Scenario 6: An IPv4 network receiving multicast from an IPv6
      network

   In this scenario we consider IPv4 and IPv6 networks belonging to the
   same organization. The unicast equialent is "an IPv4 network to an
   IPv6 network".

   We would like any IPv4 host to receive from any IPv6 source. This
   can be seen as special cases of either scenario (2), Section 2.2 or
   scenario (4), Section 2.4, where any of those techniques might apply.
   However, as discussed in scenario (5) Section 2.5 where we looked at
   how to do multicast in the reverse direction; the limited number of
   hosts and common managment might allow us to just use static mappings

or a stateless approach by restricting which IPv6 addresses are used.
By using these techniques one may be able to create mappings that can
be used for multicast in both directions, combining this scenario
with scenario (5).

3. Framework

   Having considered some possible scenarios for where and how we may
   use multicast translation, we will now consider some general issues
   and the different components of such solutions.

3.1. Addressing

   When doing stateless translation, one need to somehow encode IPv4
   addresses inside IPv6 addresses so that there is a well defined way
   for the translator to transform an IPv6 address into IPv4. This can
   be done with techniques like IVI [I-D.xli-behave-ivi] and
   [I-D.venaas-behave-mcast46].

   There are two types of addressing schemes related to the IPv4/IPv6
   multicast translation. The source addressing and the group
   addressing.

3.1.1. Source addressing

   Source addressing issues is the same as in the unicast IPv4/IPv6
   translation defined in [RFC6052]. The IPv4-mapped address is used
   for representing IPv4 in IPv6 and the IPv4-translatable address is
   used for representing IPv6 in IPv4 when the stateless translator is
   used. The multicast RPF relies on the source address to build the
   distribution tree. Therefore, depending on the operation mode of the
   IPv4/IPv6 translator and receiving directions, the IPv4-mapped or the
   IPv4-translatable addresses will be used.

3.1.2. Group addressing

   Group addressing issue is unique to the IPv4/IPv6 multicast
   translation. The entire IPv4 group addresses can be uniquely
   represented by the IPv6 group addresses, while the entire IPv6 group
   addresses cannot be uniquely represented by the IPv6 group addresses.
   Therefore, special group address mapping rule between IPv4 group
   addresses and IPv6 group addresses should be defined for the IPv4/
   IPv6 multicast translation.

3.2. Routing

   The actual translation of multicast packets may not be very
   complicated, in particular if it can be stateless. For the multicast
   to actually go through the translator we need to have routes for the
   multicast source addresses involved, so that multicast packets both
   on their way to and from the translator satisfy RPF checks. These
   routes are also needed for protocols like PIM-SM to establish a
   multicast tree, since RPF is used to determine where to send join

messages. To go into more detail we need to look at different
scenarios like SSM (Source-Specific Multicast) and ASM (Any-Source
Multicast), and PIM versus IGMP/MLD.

3.2.1. Translation with PIM and SSM

When doing SSM, a receiver specifies both source and group addresses.
If the receiver is to receive translated packets, it must do an IGMP/
MLD join for the source and group address that the data packets will
have after translation. We will later look at how it may learn those
addresses. For the source address it joins, the unicast routing (or
it may be an alternate topology specific to multicast), must point
towards the translator. With this in place, PIM should build a tree
hop-by-hop from the last-hop router to the translator. The
translator then maps the source and group addresses in the PIM join
to the source and group the data packets have before translation.
The translator then does a PIM join for that source and group.
Provided the routing is correct, this will then build a tree all the
way to the source. Finally when these joins reach the source, any
data sent by the source will follow this path to the translator, get
translated, and then continue to the receiver.

3.2.2. Translation with PIM and ASM

Let us first consider PIM Sparse Mode. In this case a receiver just
joins a group. If this group is to be received via the translator we
need to send joins towards the translator, but initially PIM will
send joins towards the RP (Rendezvous-Point) for the group. The most
efficient solution is probably to make sure that the translator is
configured as an RP for all groups that one may receive through it.
That is, for the groups it translates to. E.g. if IPv4 groups are
embedded into an IPv6 multicast prefix, then the translator could be
an RP for that specific prefix. The translator may then translate
the group and join towards the group address that is used before
translation. Note that if the translator also is an RP for the
addresses used before translation, it should know which sources exist
and join each of these. If it is not an RP, it needs to join towards
the RP. If the translator did not know the sources, it may join each
of the sources as soon as it receives from them (that is, switching
to Shortest Path Trees). When the translator receives data, it
translates it and then sends the translated data. This then follows
the joins for the translated groups to the receivers. When the last-
hop routers start receiving, they will probably (this is usually the
default behavior) switch to SPTs (Shortest Path Trees). These trees
also need to go to the translator and would probably follow the same
path as the previously built shared tree. One might argue here that
switching to the SPT has no benefit if it is the same path anyway.
Also with shared trees, RPF is not an issue, so the translated source

addresses don't need to be routed towards the translator.

At the end of the previous paragraph we pointed out that there is no benefit in switching to shortest path trees if they have to go via the translator anyway. A possibility here could be to use Bidirectional PIM where there is no source specific state and data always go through the RP. It is possible to use Bidir just for those groups that are translated, and then make the translator the RP.

3.2.3. Translation with IGMP/MLD

For translation taking place close to the edge, e.g. a home gateway, one may consider just using IGMP and MLD, and no PIM. In that case the translator should for any received MLD reports for IPv6 groups that correspond to translated IPv4 groups, map those into IGMP reports that it sends out on the IPv4 side. And vice versa for data in the other direction. Note that a translator implementation could also choose to do this in just one direction. For SSM it would also need to translate the source addresses.

3.3. Translation in operation

Currently, the proposed solutions for IPv6/IPv4 translation are classified into stateless translation and stateful translation.

3.3.1. Stateless Translation

For stateless translation, the translation information is carried in the address itself, permitting both IPv4->IPv6 and IPv6-<IPv4 sessions establishment. The stateless translation supports end-to-end address transparency and has better scalability compared with the stateful translation. See [RFC6145] and [I-D.xli-behave-ivi].

Stateless translation can be used for Scenarios 1, 2, 5 and 6, i.e. it supports "An IPv6 network receiving multicast from the IPv4 Internet", "the IPv4 Internet receiving multicast from an IPv6 network", "An IPv6 network receiving multicast from an IPv4 network" and "An IPv4 network receiving multicast from an IPv6 network".

In the stateless translation, an IPv6 network uses the IPv4-translatable addresses, while the IPv4 Internet or an IPv4 network can be represented by IPv4-mapped addresses.
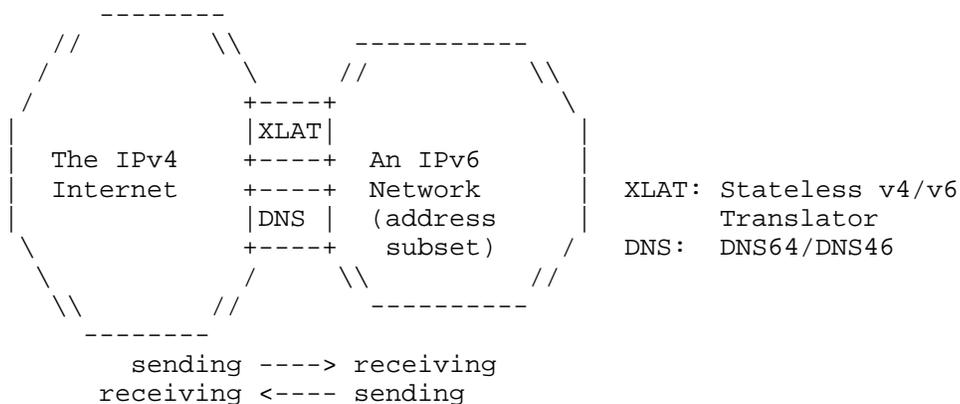
```
                 --------
             //          \\     ----------
            /              \   //          \\
           /              +----+             \
          |               |XLAT|              |
          |   The IPv4    +----+  An IPv6     |     XLAT: Stateless v4/v6
          |   Internet    +----+  Network     |           Translator
          |               |DNS |  (address    |     DNS:  DNS64/DNS46
           \              +----+   subset)    /
            \              /   \\           //
             \\          //     ---------
              --------
                sending ----> receiving
               receiving <---- sending
```

Figure 1: Stateless translation for Scenarios 1 and 2

```
            --------        ---------
        //          \\    //          \\
       /              \  //             \
      /              +----+              \
     |               |XLAT|               |
     |   An IPv4     +----+  An IPv6      |     XLAT: v4/v6
     |   Network     +----+  Network      |           Translator
     |               |DNS |               |     DNS:  DNS64/DNS46
      \              +----+              /
       \\          //    \\           //
        --------        ---------
           sending ----> receiving
          receiving <---- sending
```
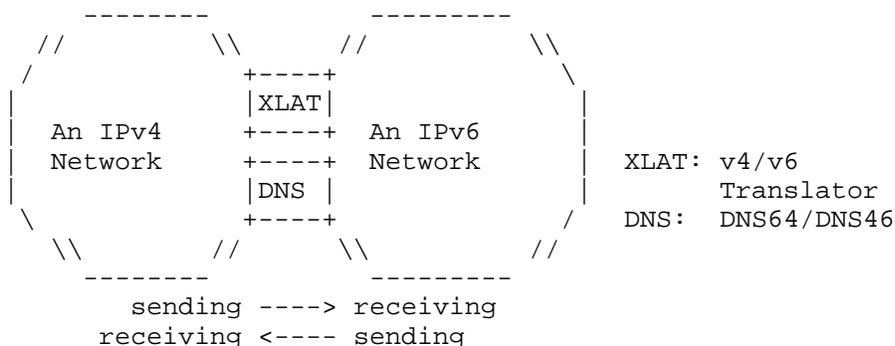
Figure 2: Stateless translator for Scenarios 5 and 6

3.3.2. Stateful Translation

   For stateful translation, the translation state is maintained between
   IPv4 address/port pairs and IPv6 address/port pairs, enabling IPv6
   systems to open sessions with IPv4 systems. See [RFC6145] and
   [RFC6146].

   Stateful translator can be used for Scenarios 1, 3 and 5, i.e. it
   supports "An IPv6 network receiving multicast from the IPv4
   Internet", "The IPv6 Internet receiving multicast from an IPv4
   network" and "An IPv6 network receiving multicast from an IPv4
   network".

In the stateful translation, an IPv6 network or the IPv6 Internet use
any IPv6 addresses, while the IPv4 Internet or an IPv4 network can be
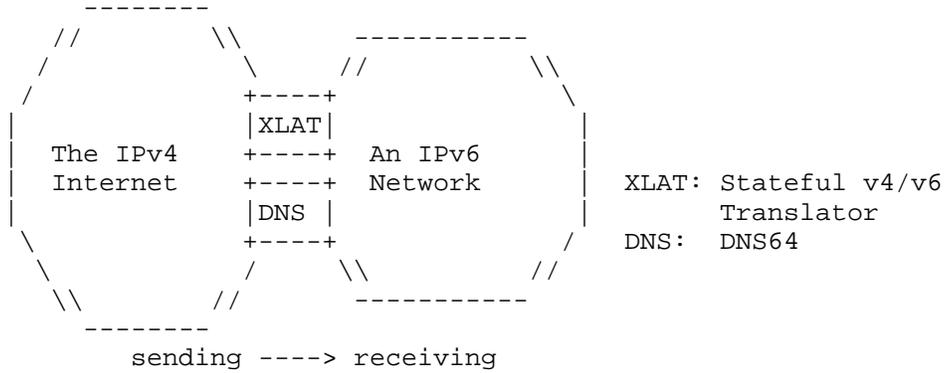represented by IPv4-mapped addresses.

```
              --------
           //          \\    -----------
          /             \   //          \\
         /          +----+              \
         |          |XLAT|              |
         |  The IPv4 +----+  An IPv6     |
         |  Internet  +----+  Network     |   XLAT: Stateful v4/v6
         |          |DNS |              |          Translator
          \         +----+            /   DNS:  DNS64
           \          /    \\       //
            \\       //      -----------
              --------
              sending ----> receiving
```

                 Figure 3: Stateful translator for Scenario 1

```
                           -----------
            ----------    //          \\
          //          \\  /             \
         //            \ /               \
        /          +----+               \
        |          |XLAT|               |
        |  An IPv4  +----+  The IPv6      |
        |  Network   +----+  Internet      |   XLAT: v4/v6
        |          |DNS |               |          Translator
         \         +----+             /   DNS:  DNS64
          \\       //     \          /
           ---------       \\      //
                             -----------
              sending ----> receiving
```
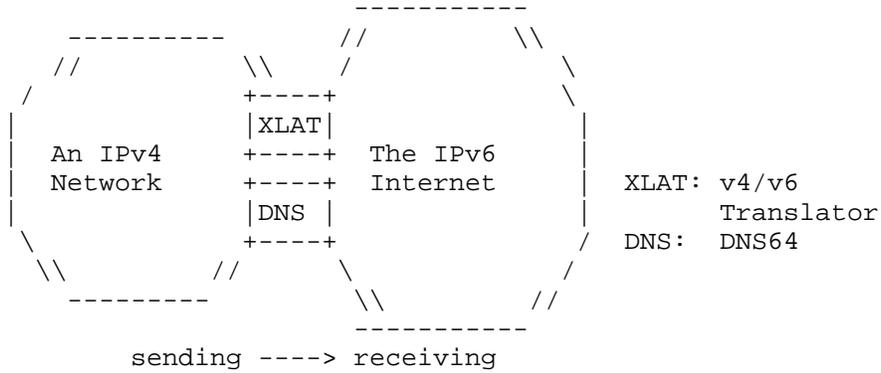
                 Figure 4: Stateful translator for Scenario 3

```
                --------         ---------
              //         \\    //           \\
             /             +----+             \
             |             |XLAT|             |
             |   An IPv4   +----+  An IPv6    |
             |   Network   +----+  Network    |   XLAT: v4/v6
             |             |DNS |             |         Translator
             \             +----+             /   DNS:  DNS64
              \\         //    \\           //
                --------         ---------
                 sending ----> receiving
```
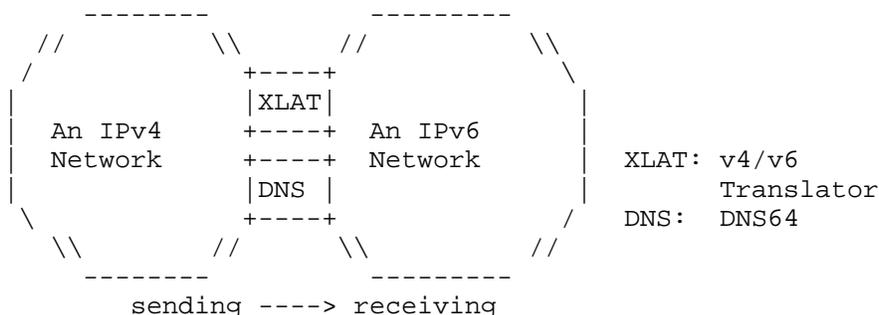
                Figure 5: Stateful translator for Scenario 5

3.4. Application layer issues

   The main application layer issue is perhaps how the applications
   learn what groups (or sources and groups) to join. For unicast,
   applications may often obtain addresses via DNS and a DNS-ALG. For
   multicast, DNS is usually not used, and there are a wide range of
   different ways applications learn addresses. It can be through
   configuration or user input, it can be URLs on a web page, it can be
   SDP files (via SAP or from web page or mail etc), or also via
   protocols like RTSP/SIP. It is no easy task to handle all of these
   possible methods using ALGs.

   SDP is maybe the most common way for applications to learn which
   multicast addresses (and other parameters) to use in order to receive
   a multicast session. Inside the SDP files it is common to use
   literal IP addresses, but it is also possible to specify domain
   names. Applications would then query the DNS for the addresses, and
   a DNS-ALG could perform the necessary translation. There is however
   a problem with this.

Here is a typical SDP taken from RFC 2327:

```
v=0
o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e=mjh@isi.edu (Mark Handley)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 31
m=application 32416 udp wb
a=orient:portrait
```

The line of interest here is "c=IN IP4 224.2.17.12/127". It is legal
to use a domain name, this line would then become e.g. "c=IN IP4
mcast.example.com/127". The problem here is that the application is
told to use IPv4. It will expect the name to resolve to an IPv4
address, and may ignore any IPv6 replies. One could argue that it
would be incorrect to use IPv6, since IPv4 is specified. For DNS to
solve our problem, we would need a new IP neutral SDP syntax, and
applications would need to be updated to support it.

An alternative to rewriting addresses in the network is to make the
applications aware of the translation and mappings in use. One
approach could be for the source to create say SDP that includes both
the original and the translated addresses. This may require use of
techniques like CCAP [I-D.boucadair-mmusic-ccap] for specifying both
IPv4 and IPv6 multicast addresses, allowing the receiver to choose
which one to use. The other alternative would be for the receiving
application to be aware of the translation and the mapping in use.
This means that the receiving application can receive the original
SDP, but then apply the mapping to those addresses.

As we just discussed, it may be useful for applications to perform
the mappings. The next question is how they may learn those
mappings. The easiest would be if there was a standard way used for
all mappings, e.g. a well-known IPv6 prefix for embedding IPv4
addresses. But that does not work in all scenarios. There could be
a way for applications to learn which prefix to use, see
[I-D.wing-behave-learn-prefix]. But note that there may be different
multicast prefixes depending on whether we are doing SSM or ASM and
scope. In addition we need the unicast prefix for the multicast
source addresses. Alternatively one could imagine applications
requesting mappings for specific addresses on demand from the
translator. The translator could have static mappings, or install

mappings as requested by applications.

An alternative to making applications aware of the translation and
rewriting addresses as needed, could be to do translation in the API
or stack, so that e.g. an application joins an IPv4 group, the API or
stack rewrites that into IPv6 and sends the necessary MLD reports.
When IPv6 packets arrive, the API/stack can rewrite those packets
back to IPv4. This could allow legacy IPv4 applications to run on a
dual-stack node (or IPv6-only with translation in the API) to receive
IPv4 packets through an IPv6-only network. But in this case it might
be better to just use tunneling.

3.5. Further Work

There are some special cases and scenarios that should be added to
this document. One is addressing. Are there certain types of IPv6
multicast addresses that could make translation easier? What happens
if there are multiple translators? And also more details on
translation in the host, e.g. bump-in-the-stack or bump-in-the-API.

The document layout of the IPv4/IPv6 multicast translation should be
presented in this document.

4. IANA Considerations

   This document requires no IANA assignments.

5. Security Considerations

   This requires more thought, but the author is not aware of any
   obvious security issues specific to multicast translation.

6. Acknowledgements

   Dan Wing provided early feedback that helped shape this document.
   Dave Thaler also provided good feedback that unfortunately still has
   not been addressed in this document. See Section 3.5.

7. References

7.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC2766] Tsirtsis, G. and P. Srisuresh, "Network Address
              Translation - Protocol Translation (NAT-PT)", RFC 2766,
              February 2000.

   [RFC4966] Aoun, C. and E. Davies, "Reasons to Move the Network
              Address Translator - Protocol Translator (NAT-PT) to
              Historic Status", RFC 4966, July 2007.

   [RFC5245] Rosenberg, J., "Interactive Connectivity Establishment
              (ICE): A Protocol for Network Address Translator (NAT)
              Traversal for Offer/Answer Protocols", RFC 5245,
              April 2010.

   [RFC6052] Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X.
              Li, "IPv6 Addressing of IPv4/IPv6 Translators", RFC 6052,
              October 2010.

   [RFC6144] Baker, F., Li, X., Bao, C., and K. Yin, "Framework for
              IPv4/IPv6 Translation", RFC 6144, April 2011.

   [RFC6145] Li, X., Bao, C., and F. Baker, "IP/ICMP Translation
              Algorithm", RFC 6145, April 2011.

   [RFC6146] Bagnulo, M., Matthews, P., and I. van Beijnum, "Stateful
              NAT64: Network Address and Protocol Translation from IPv6
              Clients to IPv4 Servers", RFC 6146, April 2011.

7.2. Informative References

   [I-D.xli-behave-ivi]
              Li, X., Bao, C., Chen, M., Zhang, H., and J. Wu, "The
              CERNET IVI Translation Design and Deployment for the IPv4/
              IPv6 Coexistence and Transition", draft-xli-behave-ivi-07
              (work in progress), January 2010.

   [I-D.venaas-behave-mcast46]
              Venaas, S., Asaeda, H., SUZUKI, S., and T. Fujisaki, "An
              IPv4 - IPv6 multicast translator",
              draft-venaas-behave-mcast46-02 (work in progress),
              December 2010.

   [I-D.wing-behave-learn-prefix]
             Wing, D., "Learning the IPv6 Prefix of a Network's IPv6/
             IPv4 Translator", draft-wing-behave-learn-prefix-04 (work
             in progress), October 2009.

   [I-D.boucadair-mmusic-ccap]
             Boucadair, M. and H. Kaplan, "Session Description Protocol
             (SDP) Connectivity Capability (CCAP) Attribute",
             draft-boucadair-mmusic-ccap-00 (work in progress),
             July 2009.

Authors' Addresses

    Stig Venaas
    cisco Systems
    Tasman Drive
    San Jose, CA 95134
    USA

    Email: stig@cisco.com

    Xing Li
    CERNET Center/Tsinghua University
    Room 225, Main Building, Tsinghua University
    Beijing 100084
    CN

    Phone: +86 10-62785983
    Email: xing@cernet.edu.cn

    Congxiao Bao
    CERNET Center/Tsinghua University
    Room 225, Main Building, Tsinghua University
    Beijing 100084
    CN

    Phone: +86 10-62785983
    Email: congxiao@cernet.edu.cn

                DNS46 for the IPv4/IPv6 Stateless Translator
                  draft-xli-behave-dns46-for-stateless-07

Abstract

   The stateless translator can support IPv6-initiated communications as
   well as IPv4-initiated communications for IPv6 hosts using IPv4-
   translatable addresses. DNS support for the IPv6-initiated
   communication is defined in the DNS64 specification. This document
   defines the DNS46 function for the stateless translator.

Status of this Memo

   This Internet-Draft is submitted to IETF in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at http://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on April 20, 2018.

to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction

   DNS mechanism is one of the functions model for the IPv4/IPv6
   transition [RFC6144]. DNS64 allows IPv6 hosts to resolve names of
   IPv4 hosts whereas DNS46 allows IPv4 hosts to resolve names of IPv6
   hosts.

   General DNS46 is considered harmful, as NAT-PT was deprecated by IETF
   [RFC4966]. However, stateless translators can support IPv6-initiated
   communications (scenario 1 and 3) which requires the DNS64 support
   [RFC6146] [RFC6145], as well as IPv4-initiated communications
   (scenario 2 and 4) which requires the DNS46 support [RFC6145].

   DNS64 is used for both stateful and stateless translators and it is
   defined in [RFC6147]. DNS46 is only used for the stateless
   translators and it is defines in this document.

2. Notational Conventions

   The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD,
   SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this
   document, are to be interpreted as described in [RFC2119].

3. DNS46 for the IPv4/IPv6 Stateless Translator

   The DNS46 for stateless IPv4/IPv6 translation is for the following
   scenarios:

   o Scenario 2: The IPv4 Internet to an IPv6 network.

   o Scenario 6: An IPv4 network to an IPv6 network.

3.1. Authoritative DNS server

   Since the destination is "an IPv6 network", DNS46 can be deployed as
   an authoritative DNS server [RFC1035] in an IPv6 network.

3.1.1. Static AAAA record

   This is very similar to the authoritative DNS configuration of dual-
   stack hosts. The only difference is that in the dual-stack case, the
   A record and AAAA record are independent, while in stateless
   translation case, the hosts are typically IPv6 single stack (or for
   some reason incapable of using IPv4 on a particular network) using
   IPv4-translatable addresses and the A record MUST be derived from the
   AAAA record based on the algorithm and the PREFIX information

[RFC6052].

3.1.2. Varying AAAA record

If an IPv6 host has a varying AAAA record (that is, it could change
due to the IPv6-only host changing its IPv6 address and registering
its new address via, for example, DNS Dynamic Updates [RFC2316]),
then the dynamic DNS46 function is required. However, it is still
the authoritative DNS. When the authoritative DNS receives a dynamic
update containing AAAA record, it MUST synthesize corresponding A
record before signing the zone, which can be derived based on the
algorithm and the PREFIX information [RFC6052].

3.2. DNS resolver

For scenario 2 (the IPv4 Internet to an IPv6 network) the
implementation of DNS46 resolver is almost impossible, since the IPv4
hosts are in the IPv4 Internet.

3.2.1. DNS resolver for scenario 6

However, for scenario 6 (an IPv4 network to IPv6 network) [RFC6144],
it is possible to use DNS resolver in an IPv4 network to synthesize A
records from the AAAA records based on the algorithm and the PREFIX
information [RFC6052].

4. Security Considerations

When DNS46 function is provided by authoritative DNS server, DNS46
can support DNSSEC without problem.

When DNS46 function is provided by authoritative DNS server, the
reverse DNS is also under the same network operator's control which
may provide additional validation function for the IPv4-translatable
IPv6 addresses.

5. IANA Considerations

This memo adds no new IANA considerations.

Note to RFC Editor: This section will have served its purpose if it
correctly tells IANA that no new assignments or registries are
required, or if those assignments or registries are created during
the RFC publication process. From the author's perspective, it may
therefore be removed upon publication as an RFC at the RFC Editor's
discretion.

6. Acknowledgments

   The authors would like to acknowledge the following contributors of
   this document: Branimir Rajtar, Dan Wing and Kevin Yin.

7. Normative References

   [RFC1035] Mockapetris, P., "Domain names - implementation and
              specification", STD 13, RFC 1035, DOI 10.17487/RFC1035,
              November 1987, <https://www.rfc-editor.org/info/rfc1035>.

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/
              RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

   [RFC2316] Bellovin, S., "Report of the IAB Security Architecture
              Workshop", RFC 2316, DOI 10.17487/RFC2316, April 1998,
              <https://www.rfc-editor.org/info/rfc2316>.

   [RFC4966] Aoun, C. and E. Davies, "Reasons to Move the Network
              Address Translator - Protocol Translator (NAT-PT) to
              Historic Status", RFC 4966, DOI 10.17487/RFC4966,
              July 2007, <https://www.rfc-editor.org/info/rfc4966>.

   [RFC6052] Bao, C., Huitema, C., Bagnulo, M., Boucadair, M., and X.
              Li, "IPv6 Addressing of IPv4/IPv6 Translators", RFC 6052,
              DOI 10.17487/RFC6052, October 2010,
              <https://www.rfc-editor.org/info/rfc6052>.

   [RFC6144] Baker, F., Li, X., Bao, C., and K. Yin, "Framework for
              IPv4/IPv6 Translation", RFC 6144, DOI 10.17487/RFC6144,
              April 2011, <https://www.rfc-editor.org/info/rfc6144>.

   [RFC6145] Li, X., Bao, C., and F. Baker, "IP/ICMP Translation
              Algorithm", RFC 6145, DOI 10.17487/RFC6145, April 2011,
              <https://www.rfc-editor.org/info/rfc6145>.

   [RFC6146] Bagnulo, M., Matthews, P., and I. van Beijnum, "Stateful
              NAT64: Network Address and Protocol Translation from IPv6
              Clients to IPv4 Servers", RFC 6146, DOI 10.17487/RFC6146,
              April 2011, <https://www.rfc-editor.org/info/rfc6146>.

   [RFC6147] Bagnulo, M., Sullivan, A., Matthews, P., and I. van
              Beijnum, "DNS64: DNS Extensions for Network Address
              Translation from IPv6 Clients to IPv4 Servers", RFC 6147,
              DOI 10.17487/RFC6147, April 2011,

<https://www.rfc-editor.org/info/rfc6147>.

Authors' Addresses

   Congxiao Bao
   CERNET Center/Tsinghua University
   Room 225, Main Building, Tsinghua University
   Beijing 100084
   CN

   Phone: +86 10-62785983
   Email: congxiao@cernet.edu.cn

   Xing Li
   CERNET Center/Tsinghua University
   Room 225, Main Building, Tsinghua University
   Beijing 100084
   CN

   Phone: +86 10-62785983
   Email: xing@cernet.edu.cn

   Yan Ma
   Beijing University of Post and Telecommunication
   Mailbox 121, Beijing University of Post and Telecommunication
   Beijing 100876
   CN

   Phone: +86 10-62283044
   Email: mayan@bupt.edu.cn