BEHAVE WG
Internet-Draft
Intended status: Standards Track
Expires: September 8, 2009

M. Bagnulo
UC3M
A. Sullivan
Shinkuro
P. Matthews
Alcatel-Lucent
I. van Beijnum
IMDEA Networks
M. Endo
Yokogawa Electric Corporation

March 7, 2009

DNS64: DNS extensions for Network Address Translation from IPv6 Clients to IPv4 Servers

draft-bagnulo-behave-dns64-02

Status of this Memo

This Internet-Draft will expire on September 8, 2009.

Copyright Notice

Abstract

DNS64 is a mechanism for synthesizing AAAA records from A records. DNS64 is used with NAT64, 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 NAT64.

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 NAT64, allows an IPv6-only client to initiate communications by name to an IPv4-only server.

DNS64 is a mechanism for synthesizing AAAA resource records (RR) from A RRs. The synthesis is done by adding a /96 prefix to the IPv4 address to create an IPv6 address, where the /96 prefix is assigned to a NAT64 device.

NAT64 as defined in a companion document [I-D.bagnulo-behave-nat64] is a mechanism for translating IPv6 packets to IPv4 packets. The translation is done by translating the packet headers according to SIIT [RFC2765], translating the IPv4 server address by adding or removing a /96 prefix, and translating the IPv6 client address by installing mappings in the normal NAT manner.

Together, 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's likely that in the future, a lot of IPv6-only clients will want to connect to IPv4-only servers. These include hosts running IPv6-only applications, IPv6-only hosts as well as the cases where only IPv6-only connectivity is available between the client and the NAT64. In the general case, the approach only requires the deployment of NAT64-enabled devices that connect an IPv6-only network to the IPv4-only Internet, along with the deployment of one or more DNS64-enabled name servers in the IPv6-only network. However, some advanced features require performing the DNS64 function directly by the end-hosts themselves.

1.1. Overview

This section provides a non-normative introduction to the DNS64 mechanism.

We assume that we have a NAT64 box connecting the IPv4 network and the IPv6 network. The NAT64 device provides translation services between the two networks enabling communication between IPv4-only hosts and IPv6-only hosts. NAT64, however, is not symmetric. In order to be able to perform IPv6 - IPv4 translation NAT64 requires state, binding an IPv6 address and port (hereafter called an IPv6 transport address) to an IPv4 address and port (hereafter called an IPv4 transport address). Such binding state 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 either direction that are part of that particular flow are translated. The result is that NAT64 only supports communications initiated by the IPv6-only node towards an IPv4-only node.

To allow an IPv6 initiator to do the standard DNS lookup to learn the address of the responder, DNS64 is used to synthesize an AAAA record from the A record containing the real IPv4 address of the responder, whenever the DNS64 service cannot retrieve a AAAA record for the requested host name. DNS64 appears as a regular recursive resolver for the IPv6 initiator. The DNS64 node receives a AAAA DNS query generated by the IPv6 initiator. It first attempts a recursive resolution of the requested AAAA record. If there is no AAAA record available for the target node (which is the normal case when the target node is an IPv4-only node), DNS64 performs a query for the A record. If an A record is discovered, DNS64 creates a synthetic AAAA RR by adding the Pref64::/96 of a NAT64 to the responder's IPv4 address (i.e. if the IPv4 node has IPv4 address X, then the synthetic AAAA RR will contain the IPv6 address formed as Pref64:X). The synthetic AAAA RR is passed back to the IPv6 initiator, which will initiate an IPv6 communication with the 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 in [I-D.bagnulo-behave-nat64].

The only shared state between the DNS64 and the NAT64 is the Pref64::/96 that must be configured to be the same on both; there is no communication between the DNS64 and NAT64 functions.

There are two main different setups where DNS64+NAT64 approach is expected to be used (other setups are possible as well, but these two are the main ones identified at the time of this writing).

One possible setup that is expected to be common is the case of an end site or an ISP that is providing IPv6-only connectivity or connectivity to IPv6-only hosts that wants to allow the communication from these IPv6-only connected hosts to the IPv4 Internet. (This case is called An-IPv6-network-to-IPv4-Internet). In this case, the NAT64 is used to connect the end site or the ISP to the IPv4 Internet and the DNS64 function is provided by the end site or the ISP.

The other possible setup that is expected is an IPv4 site that wants that its IPv4 servers to be reachable from the IPv6 Internet. (This case is called IPv6-Internet-to-an-IPv4-network). It should be noted that the IPv4 addresses used in the IPv4 site can be either public or private. In this case, the NAT64 is used

to connect the IPv4 end site to the IPv6 Internet and the DNS64 function is provided by the end site itself.

The DNS64 function can be performed in two places.

One option is to locate the DNS64 function in recursive name servers serving end hosts. In this case, when an IPv6 device queries the name server for a AAAA RR for an IPv4 only host, the name server can perform the synthesis to the AAAA RR and pass it 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, called DNS64 in DNS server mode, is expected to be used in both An-IPv6-network-to-IPv4-Internet setup and IPv6-Internet-to-an-IPv4-network setup.

The other option is to place the DNS64 function in the end hosts themselves, 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 the AAAA RR for internal usage. This mode is compatible with some advanced 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, called DNS64 in stub-resolver mode, is expected to be used only in the An-IPv6-network-to-IPv4-Internet setup case.

1.2. Walkthrough

In this section we illustrate how the DNS64 behaves in the different scenarios that are expected to be common. We consider then 3 possible scenarios, namely:

1. An-IPv6-network-to-IPv4-Internet setup with DNS64 in DNS server mode

2. An-IPv6-network-to-IPv4-Internet setup with DNS64 in stub-resolver mode

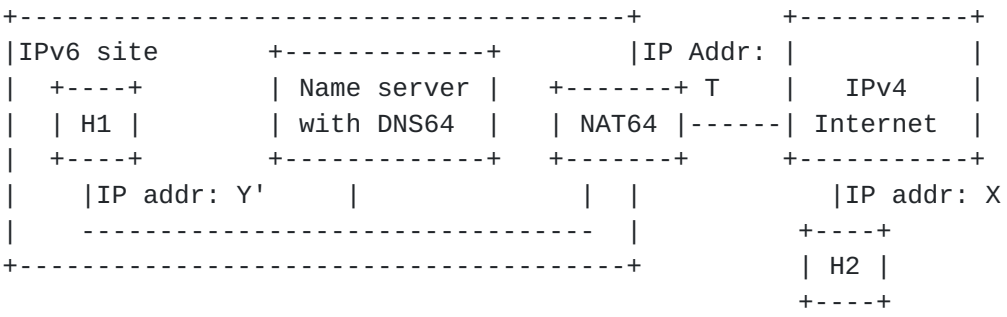
3. IPv6-Internet-to-an-IPv4-network setup DNS64 in DNS server mode

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; prefixes are indicated by "P::X", which is an IPv6 address built from an IPv4 address X by adding the prefix P, mappings are indicated as "(X,x) <--> (Y',y)".

1.2.1. 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 a IPv4 node located in the IPv4 Internet.

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



The figure shows an IPv6 node H1 which has an IPv6 address Y' and an IPv4 node H2 with IPv4 address X.

A NAT64 connects the IPv6 network to the IPv4 Internet. This NAT64 has a /96 prefix (called Pref64::/96) associated to its IPv6 interface and an IPv4 address T assigned to its IPv4 interface.

The other element involved is the local name server. The name server is a dual-stack node, so that H1 can contact it via IPv6, while it can contact IPv4-only name servers via IPv4.

The local name server needs to know the /96 prefix assigned to the local NAT64 (Pref64::/96). For the purpose of this example, we assume it learns this through manual configuration.

For this example, assume the typical DNS situation where IPv6 hosts have only stub resolvers, and always query a name server 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 FQDN(H2). H1 does this by sending a DNS query for an 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.

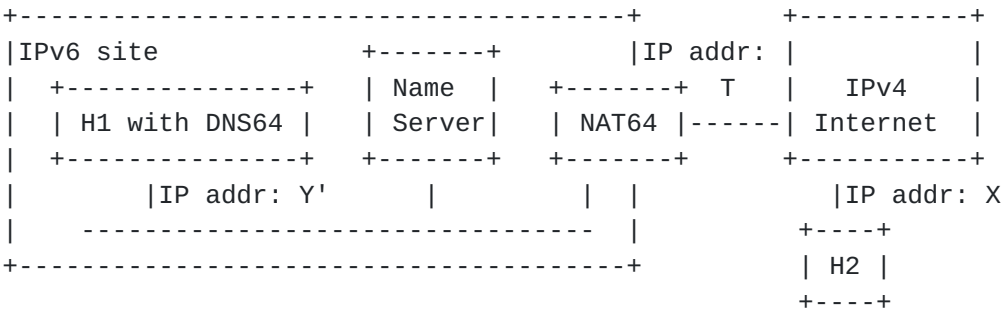
3. The recursive name server queries for an A record for H2 and gets back an A record containing the IPv4 address X. The name server then synthesizes an AAAA record. The IPv6 address in the AAAA record contains the prefix assigned to the NAT64 in the upper 96 bits and the IPv4 address X in the lower 32 bits.

4. H1 receives the synthetic AAAA record and sends a packet towards H2. The packet is sent from a source transport address of (Y',y) to a destination transport address of (Pref64:X,x), where y and x are ports chosen by H2.

5. The packet is routed to the IPv6 interface of the NAT64 and the subsequent communication flows by means of the NAT64 mechanisms as described in the NAT64 specification[I-D.bagnulo-behave-nat64].

1.2.2. An-IPv6-network-to-IPv4-Internet setup with DNS64 in stub-resolver mode

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



The figure shows an IPv6 node H1 which has an IPv6 address Y' and an IPv4 node H2 with IPv4 address X. Node H1 is implementing the DNS64 function.

A NAT64 connects the IPv6 network to the IPv4 Internet. This NAT64 has a /96 prefix (called Pref64::/96) associated to its IPv6 interface and an IPv4 address T assigned to its IPv4 interface.

H1 needs to know the /96 prefix assigned to the local NAT64 (Pref64::/96). For the purpose of this example, we assume it learns this through manual configuration but we will discuss different options for doing this in the analysis section of this document.

Also shown is a name server. For the purpose of this example, we assume that the name server is a dual-stack node, so that H1 can contact it via IPv6, while it can contact IPv4-only name servers via

IPv4.

For this example, assume the typical situation where IPv6 hosts have only stub resolvers and always query a name server that provides recursive lookups (henceforth called "the recursive name server"). 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 FQDN(H2). 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 X. The DNS64 function within H1 then synthesizes a AAAA record. The IPv6 address in the AAAA record contains the prefix assigned to the NAT64 in the upper 96 bits and the IPv4 address X in the lower 32 bits.

4. H1 sends a packet towards H2. The packet is sent from a source transport address of (Y',y) to a destination transport address of (Pref64:X,x), where y and x are ports chosen by H2.

5. The packet is routed to the IPv6 interface of the NAT64 and the subsequent communication flows using the NAT64 mechanisms as described in the NAT64 specification[I-D.bagnulo-behave-nat64].

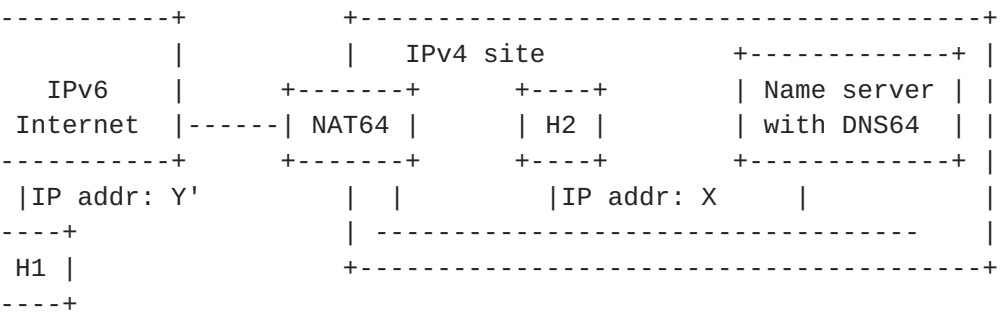
1.2.3. IPv6-Internet-to-an-IPv4-network setup DNS64 in DNS server mode

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

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 servers. Such an IPv6 address would be constructed as the Pref64::/96 concatenated with the IPv4 address of the IPv4 server. Note that the IPv4 address can be a public or a private address; the latter does not present any additional difficulty. Once these IPv6 addresses have been assigned to represent the IPv4 servers 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.

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 servers, there are two options: One option is to modify the 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 the authoritative server to generate synthetic AAAA records for a zone, possibly based on additional constraints, upon the reception of the DNS query for the AAAA RR. The first option, the synthesis upon the DynDNS update, recommended over the second option, i.e. the synthesis over the reception of the AAAA RR DNS query. The DNS64 behavior that we describe in this section covers the last case i.e. AAAA RR being synthesized when the DNS query arrives. Note that we don't recommend this approach over the previous one where AAAA RR are generated upon the dynamic registration. However, this is specified in this document because this is a part that is related to the DNS64 function.

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



The figure shows an IPv6 node H1 which has an IPv6 address Y' and an IPv4 node H2 with IPv4 address X.

A NAT64 connects the IPv4 network to the IPv6 Internet. This NAT64 has a /96 prefix (called Pref64::/96) associated to its IPv6 interface.

Also shown is the authoritative name server for the local domain with DNS64 functionality. For the purpose of this example, we assume that the name server is a dual-stack node, so that H1 can contact it via IPv6, while it can be contacted by IPv4-only nodes to receive dynamic DNS updates via IPv4.

The local name server needs to know the /96 prefix assigned to the local NAT64 (Pref64::/96). For the purpose of this example, we assume it learns this through manual configuration.

The steps by which H1 establishes communication with H2 are:

1. H1 does a DNS lookup for FQDN(H2). H1 does this by sending a DNS query for an AAAA record for H2. The query is eventually forwarded to the server in the IPv4 site. Assume the local name server is implementing DNS64 functionality.

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 FQDN(H2) 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 an AAAA record from the A RR and the relevant Pref64::/96. The IPv6 address in the AAAA record contains the prefix assigned to the NAT64 in the first 96 bits and the IPv4 address X in the lower 32 bits.

4. H1 receives the synthetic AAAA record and sends a packet towards H2. The packet is sent from a source transport address of (Y',y) to a destination transport address of (Pref64:X,x), where y and x are ports chosen by H2.

5. The packet is routed through the IPv6 Internet to the IPv6 interface of the NAT64 and the communication flows using the NAT64 mechanisms as described in the NAT64 specification[I-D.bagnulo-behave-nat64].

2. Terminology

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

Authoritative server A DNS server that can answer authoritatively for a given DNS question.

DNS64: A logical function that synthesizes AAAA records (containing IPv6 addresses) from A records (containing IPv4 addresses).

DNS64 recursor: A recursive resolver that provides the DNS64 functionality as part of its operation.

Recursive resolver: A DNS server that accepts requests from one resolver, and asks another resolver for the answer on behalf of the first resolver. In the context of this document, "the recursive resolver" means a recursive resolver immediately next in the DNS resolution chain from an end point. The end point usually has only a stub resolver available.

Synthetic RR: A DNS resource record (RR) that is not contained in any zone data file, but has been synthesized from other RRs. An example is a synthetic AAAA record created from an A record.

Stub resolver: A resolver with minimum functionality, typically for use in end points that depend on a recursive resolver. Most end points on the Internet as of this writing use stub resolvers.

NAT64: A device that translates IPv6 packets to IPv4 packets and vice-versa, with the provision that the communication must be initiated from the IPv6 side. The translation involves not only the IP header, but also the transport header (TCP or UDP).

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

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).

Mapping: A mapping between an IPv6 transport address and a IPv4 transport address. Used to translate the addresses and ports of packets flowing between the IPv6 host and the IPv4 host. In NAT64, the IPv4 transport address is always a transport address assigned to the NAT64 itself, while the IPv6 transport address belongs to some IPv6 host.

For a detailed understanding of this document, the reader should also be familiar with DNS terminology [RFC1035] and current NAT terminology [RFC4787]. Some parts of this document assume familiarity with the terminology of the DNS Security extensions outlined in [RFC4035]

3. Normative Specification

3.1. DNS64

A 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.

The only configuration parameter required by the DNS64 is the IPv6 prefix assigned to a NAT64. This prefix is used to map IPv4 addresses into IPv6 addresses, and is denoted Pref64. The DNS64 learns this prefix through some means not specified here.

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 recursive query or, in the case of an authoritative server, by examining its own results. If this query results in one or more AAAA records or in an error condition, this result is returned to the requesting client as per normal DNS semantics. If the query results in no error but an empty answer section in the response, the DNS64 resolver attempts to retrieve an A record for the name in question. If this query results in an empty result or in an error, this result is returned to the client. If the query results in one or more A RRs, the DNS64 synthesizes AAAA RRs based on the A RRs and the Pref64 prefix of the translator. The DNS64 resolver then returns the synthesized AAAA records to the client.

DNS64 MAY perform the query for the AAAA RR and for the A RR in parallel, in order to minimize the delay. However, this would result in performing unnecessary A RR queries in the case that the AAAA RR exists. A possible trade-off would be to make 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 to 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.)

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

- The NAME field is set to the NAME field from the A record
- The TYPE field is set to 28 (AAAA)
- The CLASS field is set to 1 (IN)

- The TTL field is set to the TTL of the original A RR
- The RDLENGTH field is set to 16
- The RDATA field is set to the IPv6 address whose upper 96 bits are Pref64::/96 and whose lower 32 bits are the IPv4 address from the RDATA field of the A record.

3.2. Handling PTR RRs

If the DNS64 receives a PTR query for the IP6.ARPA domain, the DNS64 searches for the queried prefix on its own list of prefixes (i.e. one or more Pref64 available). If the prefix contained in the query is not included in its own list of prefixes, the DNS64 just forwards the query to the (internal or external) resolver. If the prefix is included in its own prefix list, then the DNS64 translates the QNAME field to the IN-ADDR.ARPA domain by removing the Pref64:/96, and extracting the IPv4 address in reversed order. The DNS64 then sends the translated query to the resolver. When the resulting query is resolved, the DNS64 restores the QNAME field to the IP6.ARPA domain, and sends the DNS response to the original client.

4. Solution space analysis

So far the document describes the basic functionality that is needed to perform the DNS64 function. However, there are several open issues that require further discussion. This section present the issues and several approaches to deal with them.

Having DNS synthesize AAAA records creates a number of issues, as described in [RFC4966]:

- The synthesized AAAA records may leak outside their intended scope;
- Dual-stack hosts may communicate with IPv4-only servers using IPv6 which is then translated to IPv4, rather than using their IPv4 connectivity;
- Interaction with DNSSEC;
- The DNS64 box needs to learn the Pref64::/96 used by the NAT64 box;
- Supporting the case of multiple NAT64 boxes with different associated prefixes.

4.1. Tagging synthetic RR

As a general architecture consideration, it seems a good approach to preserve the transparency when the semantics of an existent protocol is changed. In this case, it seems architecturally sound to tag the synthetic RR, so they can be identified as synthetic and act accordingly. There are several ways we can achieve that, but all of them impose some trade-offs between architectural cleanness and deployability.

Tagging the synthetic RRs is relevant in the An-IPv6-network-to-IPv4-Internet setup, where the synthesis is not made by the authoritative name server and the following discussion applies. This is not the case when the synthesis is performed by the authoritative DNS server, such as in the case of the setup presented in IPv6-Internet-to-An-IPv4-network.

Tagging is mostly useful for troubleshooting, and for translation-aware end points.

One option to tag the synthetic RR would be to use a different RR type i.e. not to synthesize AAAA RR but to create a new RR type e.g. AAAASYNT that would be used in this cases. This seems architecturally clean, but the problem is that the host needs to explicitly ask for this new RR type and this is simply incompatible with existing IPv6 hosts. In order to support this, we would need to upgrade the hosts and if we are going to do that, we may as well simply use the DNS64 stub resolver mode. However, it is an explicit goal of DNS64/NAT64 to support unmodified IPv6 hosts, so this could be considered as an optimization but we would still need to synthesize AAAA RR and we still need to mark those. Therefore, this option is rejected.

Another option is to create a new RR that would be included in the additional information part of the DNS response, basically saying that one or more of the RRs contained in the DNS response message are synthetic. So, in this case, we could create a new AAAASYNT RR type and queries could be accepted directly for this RR and when a AAAA RR is synthesized for the correspondent FQDN, the AAAASYNT would be included in the additional information part of the DNS response that contains the synthetic AAAA RR. Of course, in order to benefit from this mechanism, the receiving host needs to be upgraded to understand the new AAAASYNT RR, but this is backward compatible, in the sense that if the host does not understand the AAAASYNT RR it would still use the AAAA RR and it would be able to communicate. In addition, a host can query explicitly for the AAAASYNT RR and verify if a given AAAA RR is synthetic or not. This would result in a sort of public repository of synthetic AAAA RRs, which is useful for transparency.

One downside with this is that the tag is not directly associated with the synthetic AAAA RR but is some additional information contained in the DNS response. In this sense we are tagging the DNS response message rather than tagging the synthetic RR. Such additional information could be lost in caching servers or other means of relying DNS information, losing the tag.

A similar option as the previous one would be to use an EDNS0 option [RFC2671] to tag the DNS responses that contain one or more synthetic AAAA RRs. There are however some additional issues with this. The EDNS0 option can only be included if the DNS query contained the EDNS0 option. It would also be possible to find out if a given AAAA RR is synthetic, since the querying party could ask for the AAAA RR and include the EDNS0 option.

Another option would be to use a well known prefix as the Pref64::/96. In this case, we could assume that any AAAA RR containing the well know Pref64::/96 is synthetic. This would achieve tagging the RR itself, since this information can not be lost in caching servers. Additional discussion about the advantages and disadvantages of using a Well-Known prefix can be found [I-D.miyata-behave-prefix64].

4.2. Dual stack nodes

When dual stack nodes are involved in the communication, the potential issue is that they end up using translated connectivity even though the native connectivity is available. There are multiple ways to try to deal with this issue, here we consider those related to DNS64.

There are two different cases involving dual-stack nodes. Communication initiated from an IPv6-only node towards a dual stack node and communication initiated from a dual stack node towards an IPv4-only node. We will next consider each one of these cases.

4.2.1. Communication initiated from an IPv6-only node towards a dual stack node

In this case, the IPv6 only node will query for the FQDN of the dual stack node. The DNS64 function will try first to get the AAAA RR. Since there is one available, it will return it and no AAAA RR will be synthesized from the A RR of the dual stack node. However, it should be noted that the DNS64 must first try to get the real AAAA RR before starting the synthesis, if not, it may result in the aforementioned problem.

4.2.2. Communication initiated from a dual stack node toward an IPv4 only node

We consider now the case of a dual stack node is initiating a communication with a IPv4-only node that has a public IPv4 address published in an A RR. Dual stack nodes that have both IPv6 and IPv4 connectivity and are configured with an address for a DNS64 as their resolving nameserver may receive responses containing synthetic AAAA resource records. If the node prefers IPv6 over IPv4, using the addresses in the synthetic AAAA RRs means that the node will attempt to communicate through the NAT64 mechanism first, and only fall back to native IPv4 connectivity if connecting through NAT64 fails (if the application tries the full set of destination addresses). We have multiple options to avoid this.

One option would be to configure the dual stack nodes not to use the DNS64 mechanism. This would mean that the server they are using should not be performing this function (at least not for them). The drawback of this option is that the translated connectivity would not be usable for backup purposes if the native connectivity is down.

The other option is that the dual stack nodes perform the DNS64 in stub resolver mode. In this case, they know which RRs are synthetic and so they know when the connectivity is translated and can be avoided. The problem with this option is that it only works for upgraded dual stack nodes and not with currently available nodes.

Another option is that dual stack nodes identify synthetic AAAA RR from their tagging (whatever this is) and avoid using the translated connectivity associated with the synthetic RR. However, again, this option only works for upgraded nodes.

Another option not specific to DNS64 includes using the RFC3484 policy table e.g. configuring the Pref64::/96 as low priority preference in the table. This option requires some means to properly configure the policy table, which is not currently available (only manual configuration is currently defined) (see [I-D.ietf-6man-addr-select-sol] for more on this topic).

4.3. IPv6 nodes implementing DNSSEC

DNSSEC 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. This section outlines the various cases and discusses possible ways to address the problem.

4.3.1. Recursive resolvers and an-IPv6-network-to-IPv4-Internet

A recursive resolver can be security-aware or security-oblivious. Moreover, a security-aware recursive name server can be validating or non-validating, according to operator policy. For the purposes of notation, we call these Rso (recursing, security oblivious), Rsav (recursing, security aware and validating), and Rsan (recursing, security aware and non-validating). In the cases below, the recursive server 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 signalling bits that offer some indicators of what the query originator understands.

If a query arrives at a vDNS64 with the DO bit set, the query originator is signalling 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 with the 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. Rso recursor receives a query without the DO bit clear. In this case, DNSSEC is not a concern, because the querying agent does not understand DNSSEC responses.

2. Rso receives a query with the DO bit set, and the CD bit clear. 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. Rsan receives a query with the DO bit set and the CD bit clear. An Rsan 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. Rsan receives a query with DO bit set and the CD bit set. In this case, the Rsan is supposed to pass on all the data it gets to the query initiator (this is in section 3.2.2 of 4035). This is a case will be problematic for vNAT64. If it 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. Rsav receives a query with the DO bit clear and CD clear. In this case, the Rsav validates the data. If it fails, it returns RCODE 2 (SERVFAIL); otherwise, it returns the answer. This is the ideal case for vDNS64. The Rsav validates the data, and then synthesizes the new record and passes that to the client.

6. Rsav receives a query with the DO bit set and CD clear. In principle, this ought to work like the previous case, except that the Rsav should also set the AD bit on the response. There is a potential difficulty with the AD bit; it is addressed below.

7. Rsav receives a query with DO bit set and CD set. This is effectively the same as the case where an Rsan receives a similar query, and the same thing will happen: the downstream validator will mark the data as invalid.

4.3.1.1. Strategy 1: Response-specific DNSSEC information

The vDNS64 can use 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 any translation it wants. The DNS64 MAY translate the A record to AAAA.

2. If CD is not set and DO is set, then vDNS64 SHOULD perform validation. If the data validates, the server MAY perform translation, but it MUST NOT set the AD bit. This is acceptable, because whereas the original data validated, the answer that is actually returned to the originating client is not the validated data (and therefore would not itself validate). Similarly, if the data does not validate, the vDNS64 MUST respond with RCODE=2 (server failure).
   A security-aware end point may want the security data, and may want to pass it up to an application, and this strategy will make that data unavailable. 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.

3. If the CD is set and DO is set, then vDNS64 MUST NOT perform validation, and MUST NOT perform translation. It MUST hand the data back to the query initiator, just like a regular recursing server, and depend on the client to do the validation and the translation itself.
   The disadvantage to this approach is that an end point that is

translation-oblivious but security-aware and validating simply won't 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.

4.3.1.2. Strategy 2: Treat synthesized AAAA similarly to synthesized CNAMEs

An alternative to the strategy in the previous section is to emulate the approach used by DNAME to synthesize CNAMEs [RFC2672]. In this approach, the vDNS64 synthesizes the AAAA record from the A record. When replying to the query for the AAAA, the server includes the A record in one section of the response. The AAAA record is not signed (the DNS64 server does not have the capability to do so anyway, since it does not have the necessary key). The original A record would be returned in one of two sections:

1. In the answer section: a server preparing an answer satisfied by the DNAME substitution includes a synthesized CNAME in the answer section, so by way of analogy a DNS64 server could add the synthesized AAAA to the original A record, and return all of this in the answer section.
   There are some significant differences, however, between this case and CNAME/DNAME. First, this will mean that clients will receive an A record in response to a AAAA query. Stub resolvers have always expected CNAMEs in response to A record queries; it might be surprising to receive an A record in response to a query for AAAA. In addition, it is likely that some validators will treat the unsigned AAAA record as bogus.

2. In the additional section: a DNS64 server could replace the A record with the synthetic AAAA, and put the original A record in the additional section. A validating resolver could then find the A record in the additional, detect that the IP address it contains forms part of the synthetic address, and perform an additional validation step on that A record if so desired.
   The difficulty here is that the additional data is most likely to be truncated whenever it is necessary to avoid the protocol limits, and in a DNSSEC context that is more likely to happen.

4.3.2. IPv6-Internet-to-An-IPv4-network

In this scenario, DNSSEC is naturally supported if the IPv4 network simply publishes the AAAA RR containing the IPv6 representations of the IPv4 address of its internal hosts. In this case, these AAAA RR are regular R, and the correspondent RRSIG and NSEC RRs can be

created as usual. This is the preferred approach.

If we consider dynamic environments, where AAAA RR are created dynamically, the situation is more complex. In the case of a validating security-aware stub resolver, the main issue is how to sign the new synthetic AAAA RRs that are created. If the AAAA RRs are created when the query is received, this would imply that the AAAA RRs need to be signed on-the-fly right after the AAAA RR has been synthesized. This requires that the signing keys be online in the DNS64 server, and that the signing process be very fast, and is one of the reasons that it is better to configure AAAA records in a standard way, as though the authority server were answering AAAA queries for hosts using native IPv6. In the case of a non-validating security-aware stub resolvers contact it , there's no reason to sign synthetic records and the problem is no longer relevant.

Probably we may want to recommend that if DNSSEC is used, the AAAA RRs for this case need to be generated manually or when the Dyn DNS update is performed. Question: how does Dyn DNS works with DNSSEC?

4.4. Learning the Pref64::/96 prefix

The only piece of information that needs to be shared between the devices performing the NAT64 function and the devices performing the DNS64 function is the prefix Pref64::/96. Note that the Pref64::/96 must be distributed to all the hosts that are performing the DNS64 function in stub-resolver mode and to all the name servers that are performing the DNS64 function.

One option is to configure the Pref64::/96 manually in all these devices. While this may work for servers, it doesn't seem the best approach for stub-resolvers.

Another option is to define a DHCP option to carry this information. The main issue here is the security, especially when this information is used in conjunction with DNSSEC.

Another option is to store this information in a new RR under a well known name within each domain. This information can then be signed using DNSSEC so its distribution would be secured. One possibility is to use a well known name, such as pref64.example.com, or even in example.com. Another possibility is to put it in the reverse zone. So the DNS64-aware system, as part of its initiation step, asks for the reverse lookup of the configured-interface address (i.e. $reverseaddress.ip6.arpa) but with the new RRTYPE (call it 64PREFIX). This way, the data can be part of the signed reverse zone, it can get dynamically determined as part of the protocol establishing the address of the end point, and we don't have to reserve a new special

well-known name.

For more extensive discussion on this topic, the reader is referred to [I-D.wing-behave-learn-prefix]

4.5. Supporting multiple NAT64 boxes with different associated prefixes

This discussion applies to the An-IPv6-network-to-IPv4-Internet setup.

Consider the case where we have a site with multiple NAT64 boxes. Each of these boxes has a different prefix associated, namely Pref64_1::/96, Pref64_2::/96, ..., Pref64_n::/96. suppose that the site is using one or more servers using providing the DNS64 function. The question that we consider in this section is how these prefixes are managed by the DNS64 function.

One option would be to configure only one prefix to each DNS64 device. In this case, we would achieve some form of load balance and traffic engineering features, since the hosts configured to use a given DNS64 server will use a given prefix and this means that their traffic will flow through a given NAT64 box. The problem is what happens if the NAT64 box fails. At that point, the DNS64 server should detect the failure and start using an alternative prefix. (Note that it is the NAT64 the one that have failed, but the DNS64 server is still working, so the host would not try an alternative DNS64 in this failure mode). The failure could be detected by the DNS64 device pinging itself from its IPv6 address towards its IPv4 address through the NAT64 in question.

The other option would be to configure multiple prefixes in each DNS64 server. The next question is how these are managed? We can envision several ways of managing the prefixes in the DNS64 server:

- One option is that the DNS64 synthesizes a single AAAA RR using a randomly chosen prefix. This would result in load sharing across the multiple NAT64 boxes. However, this would mean that a given IPv6 host can use different IPv4 transport addresses in the IPv4 Internet. This is because the different synthesized AAAA RR contain different prefixes and this means that the communication is established through a different NAT64 box, hence using a different IPv4 address. Moreover, it is also possible that when an IPv6 hosts initiates two different communications using the same IPv6 transport source address, these are routed through different NAT64 boxes and they are presented to the IPv4 Internet as coming from different IPv4 transport source address. While the endpoint independence requirement doesn't cover the case of multiple NATs, it does seems that this option is against the

endpoint independent behavior and should be avoided.

- Another option is to track the requesting hosts and always use the same prefix for a given host. In case of failure, the DNS64 function should detect the NAT64 is down and start using a different prefix (associated to a working NAT64 box). The downside of this option is that the DNS64 function needs to keep track of the hosts and prefixes and working NAT64 boxes. Rather than actually tracking per-client state, the same result could be achieved by performing a hash over the client's address and return AAAA records synthesized using the same Pref64 for all addresses that hash to the same value.

- Another option is for the DNS64 to return a list of synthesized AAAA RR, one per available prefix. Besides, the DNS64 function should keep track of the hosts, so the same prefix order is used in all the replies to the same host. In this case, the host will normally use the first one if it is working, so it will always use the same NAT64 box and if something fails, it should retry with an alternative address, effectively using a different NAT64 box. This would provide the fault tolerance capabilities required without need for the DNS64 to keep track of the state of the NAT64 boxes.

5. Security Considerations

See the discussion on the usage of DNSSEC and DNS64 described in the analysis section.

6. IANA Considerations

7. Changes from Previous Draft Versions

Note to RFC Editor: Please remove this section prior to publication of this document as an RFC.

[[This section lists the changes between the various versions of this draft.]]

8. Contributors

Dave Thaler

Microsoft

dthaler@windows.microsoft.com

9. Acknowledgements

This draft has benefited from the review from Dave Thaler.

This draft contains the result of discussions involving many people, including: Dan Wing, Jari Arkko, Mark Townsley, Fred Baker, Xing Li, Hiroshi Miyata, Brian Carpenter, Ed Jankiewicz, Magnus Westerlund, Ed Lewis, Rob Austein, Matthijs Mekking.

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

10. References

10.1. Normative References

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

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

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

[RFC2672] Crawford, M., "Non-Terminal DNS Name Redirection", RFC 2672, August 1999.

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

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

[I-D.ietf-behave-tcp] Guha, S., Biswas, K., Ford, B., Sivakumar, S., and P. Srisuresh, "NAT Behavioral Requirements for TCP", draft-ietf-behave-tcp-08 (work in progress), September 2008.

[I-D.ietf-behave-nat-icmp]
Srisuresh, P., Ford, B., Sivakumar, S., and S. Guha, "NAT Behavioral Requirements for ICMP protocol", draft-ietf-behave-nat-icmp-12 (work in progress), January 2009.

[I-D.bagnulo-behave-nat64]
Bagnulo, M., Matthews, P., and I. Beijnum, "NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers", draft-bagnulo-behave-nat64-02 (work in progress), November 2008.

10.2. Informative References

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

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

[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.

[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.

[RFC4966] Aoun, C. and E. Davies, "Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status", RFC 4966, July 2007.

[I-D.ietf-mmusic-ice]
Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", draft-ietf-mmusic-ice-19 (work in progress), October 2007.

[I-D.ietf-6man-addr-select-sol]
Matsumoto, A., Fujisaki, T., Hiromi, R., and K. Kanayama, "Solution approaches for address-selection problems", draft-ietf-6man-addr-select-sol-01 (work in progress), June 2008.

[RFC3498] Kuhfeld, J., Johnson, J., and M. Thatcher, "Definitions of Managed Objects for Synchronous Optical Network (SONET) Linear Automatic Protection Switching (APS) Architectures", RFC 3498, March 2003.

[I-D.wing-behave-learn-prefix]
Wing, D., "Learning the Address Family Translator's IPv6 Prefix", draft-wing-behave-learn-prefix-00 (work in progress), October 2008.

[I-D.miyata-behave-prefix64]
Miyata, H., "PREFIX64 Comparison", draft-miyata-behave-prefix64-00 (work in progress), October 2008.

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

Masahito Endo
Yokogawa Electric Corporation

Email: masahito.endou@jp.yokogawa.com