The NAT64/DNS64 tool suite for IPv6 transition

Marcelo Bagnulo
Universidad Carlos III de Madrid
marcelo@it.uc3m.es

Alberto García-Martínez
Universidad Carlos III de Madrid
alberto@it.uc3m.es

Iljitsch van Beijnum
Institute IMDEA Networks and
Universidad Carlos III de Madrid
iljitsch.vanbeijnum@imdea.org

***Abstract*—It is clear that there is not enough time to upgrade existing Internet hosts to dual stack before the IPv4 address pool depletes. This implies that the IPv6 transition and co-existence must support interaction between IPv4 nodes and IPv6 nodes. In this paper we describe NAT64 and DNS64, a tool suite that provides a way forward in the IPv4-to-IPv6 transition by allowing communication among unmodified IPv6 and IPv4 nodes.**

***Index Terms*—IPv6 transition, NAT64, DNS64.**

I. INTRODUCTION

In the early 1990s it became apparent that the 32-bit address size of the IPv4 protocol would be too small in the long run, so the IETF started to produce an updated version of IP with larger addresses, IPv6. Unfortunately, the resulting protocol is not backward compatible with IPv4, so there is a need for tools to allow the transition and coexistence of the two protocols. There are three main transition mechanisms between the existing IPv4 and the new IPv6: tunneling, dual stack and translation. Tunneling entails encapsulating an IPv6 packet inside an IPv4 packet to communicate IPv6 nodes across IPv4-only paths. Dual stack means running both IPv4 and IPv6 at the same time in endpoints (i.e. having dual stack operating systems and applications in upgraded nodes) and in the network (i.e. having dual stack network equipment, and configuring both types of addresses for new networks). Dual stack was conceived to introduce IPv6 without resigning from IPv4 until all hosts transitioned and IPv4 could be removed without disruption. However, this transition strategy is no longer viable, as the IANA IPv4 address pool was depleted on February 3rd of 2011[1] and currently less than 10% of the Internet hosts are IPv6 enabled[2]. The result is that there is no more time to migrate the current IPv4-only host base to dual stack before IPv6-only hosts are deployed. This implies the need in the short to medium term for the translation of packets between IPv4-only hosts and IPv6-only hosts to support communication between IPv4-only hosts and IPv6-only hosts.

Stateless IP and ICMP translation (SIIT) [1] and Network Address Translation – Protocol Translation (NAT-PT) [2] specified how to translate between IPv4 and IPv6. SIIT specifies *stateless* translation, which happens on a per-packet basis without the need of any state. The limitation of stateless translation is the need of a one-to-one mapping between IPv4 and IPv6 addresses. Therefore, the number of IPv6 hosts that can be served by a translator is limited to the number of IPv4 addresses available to the translation service.

NAT-PT provided a *stateful* mechanism for address translation in which a single IPv4 address could be used to map multiple IPv6 hosts. In order to map the IPv4 destination address back to the correct IPv6 destination address in the IPv4-to-IPv6 direction, the NAT-PT translator maintained temporary state in a translation table. This procedure is similar to the Network Address Translation (NAT) that is popular in IPv4 networks. Due to the larger address space of IPv6 it is easy to initiate communications in this realm to the IPv4 side by embodying the IPv4 destination address into an IPv6 prefix. However, to enable communications initiated in the IPv4 side, a complex setup involving synchronization between a DNS Application Level Gateway (DNS-ALG) and a Bi-Directional-NAT-PT was required. In 2007 the NAT-PT specification was moved to "Historic" status within the IETF after identifying problematic interactions with other aspects of IPv6 operation that resulted in reduced functionality and reliability of the network [3].

NAT64/DNS64 [4] [5] is a tool suite aimed to replace NAT-PT in a manner that addresses most of the concerns that led to its deprecation. NAT64 translates IPv4 packets into IPv6 packets and vice versa in a stateful manner and DNS64 synthesizes `AAAA` resource records for IPv4 hosts that only have `A` resource records available. A key design decision for NAT64/DNS64 is to explicitly manage only communications initiated from the IPv6 side, while relying in existent NAT-traversal techniques, such as STUN [6], to support communications initiated by the IPv4 side. To do so, NAT64 is designed to conform to the requirements and recommendations for translators defined by the IETF BEHAVE working group, thus resulting in an homogeneous behavior of NAT64 implementations. DNS64 provides similar functionality as the NAT-PT DNS-ALG, but implemented as a new architectural block instead of being performed as a transparent ALG. A consequence of this design is the ability of DNS64 to maintain compatibility with most modes of DNSSEC. Like NAT-PT, NAT64 and DNS64 are compatible with, and largely transparent to, unmodified IPv6 hosts.

This paper is organized as follows: In section II, we discuss the requirements upon which NAT64 and DNS64 are

[1] http://www.nro.net/news/ipv4-free-pool-depleted
[2] http://www.ipv6matrix.org/

based. Section III and IV describes NAT64 and DNS64 operation respectively. Section V presents a walkthrough that illustrates the behavior of NAT64 and DNS64. Finally, section VI presents our conclusions.

II. Requirements for IPv6-IPv4 translation

The Network Address Translation (NAT) technology for IPv4 was initially defined by the IETF in RFC 1631 [7]. This document described the overall functioning of a NAT, but lacked of the detailed specification that would guarantee an homogeneous behavior of NAT devices produced by different vendors. The IETF was reluctant to specify the NAT behavior in greater detail, since NAT was deemed an inferior technology that would negatively affect the Internet architecture [8]. This approach resulted in a myriad of different NAT implementations that behaved in different ways with respect to state creation and management [9]. The actual properties of a communication through a NAT box, such as how the address/port pool was managed or the lifetime of the address translation state, were hard to predict for the applications. In particular, it became cumbersome for the applications running in the private realm to set and preserve the appropriate state in the NAT to enable communications initiated from the outside. To mitigate this phenomenon, the BEHAVE WG of the IETF defined a set of behavioral requirements for IPv4 NATs covering its interaction with TCP [10], UDP [11] and ICMP [12].

Similarly to the initial NAT RFC, the NAT-PT specification failed to define the behavior of the IPv4-IPv6 stateful translator in detail. To guarantee homogenous behavior of IPv4-IPv6 translators, a set of requirements for IPv6-IPv4 translation, analogous to the existing IPv4-IPv4 requirements, should be stated. In order to do that, we take as a starting point the requirements defined for IPv4 NATs. It is straightforward to transpose some of these requirements to the NAT64 context, such as minimum binding lifetime, port assignment strategies, handling of fragmented packets, etc. However, some other requirements are essential for the NAT architecture and deserve a more careful analysis for its application to the NAT64 case, which is presented next.

To understand NAT operation it is relevant to distinguish between the *mapping* behavior and the *filtering* behavior. In general terms, an IPv4 NAT is a device connecting two realms of IPv4 addressing, one that uses private addresses and another realm which usually is the public Internet. Upon the reception of a packet coming from the private realm, the NAT creates a mapping between the source address and source port pair of the received packet and a public address and port pair available in its own pool. We will refer to an address/port pair as a *transport address*. The NAT then substitutes the source transport address of the packet with the one assigned in the binding, and forwards the packet to the public realm. The mapping behavior defines how the aforementioned binding is created. Three types of mapping behavior are defined:

- Endpoint independent mapping (Fig. 1): The mapping is solely determined by the transport address of the internal host. Packets containing the same private transport address are translated to the same transport address of the NAT's pool irrespectively of their public address and/or port.
- Address dependent mapping: The mapping is determined by the transport address of the internal host and the address of the external host. Packets containing the same private transport address and the same public address are translated to the same transport address of the NAT's pool irrespectively of the port used by the external host.
- Address and port dependent mapping: The mapping is determined by the transport address of the internal host and the transport address of the external host.

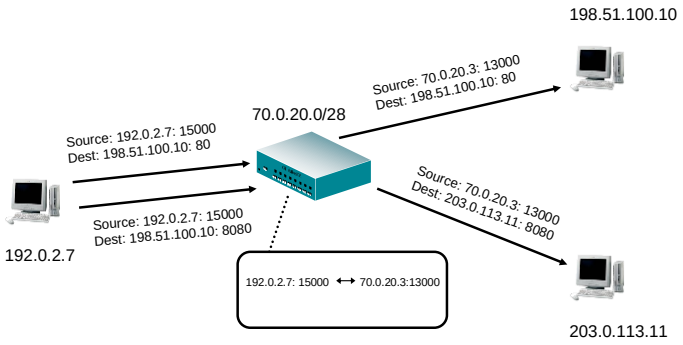

Figure 1. Endpoint independent mapping.

The different types of mappings determine how internal hosts are perceived by external hosts. Different connections initiated by the same process running in a host behind a NAT that uses endpoint independent mappings are presented with the same transport address to external hosts. This is needed by optimized NAT traversal techniques such as STUN [6] and TURN [13]. Both TCP and UDP requirements for NAT operation mandate the use of endpoint independent mappings.

However, this does not imply that a NAT box is required to forward all packets, as filtering rules can apply to comply with security policies. When a NAT box receives a packet through any of its interfaces, it applies the filtering rules to determine whether to forward the packet or to discard it, based on the address and/or port information. The following filtering behaviors are defined:

- Endpoint independent filtering: The filtering rules only depend on the transport address of the internal host. This means that packets are forwarded or dropped solely based on the transport address of the

internal host (either the private one or the one from the NAT's pool assigned through the mapping)

- Address dependent filtering: The filtering rules depend on the transport address of the internal host and also on the IP address of the external host.
- Address and port dependent filtering: the filtering rules depend on the transport address of the internal host as well as on the transport address of the external host.

The recommendation is for NATs to implement endpoint independent filtering, and if more security is needed, to allow the use of address dependent filtering. While the former type of filtering is compatible with most NAT traversal techniques (including both STUN and TURN), the latter type of filtering is compatible with a reduced set of techniques (supporting TURN but not STUN).

To extend these requirements to the NAT64 case, we need to map the roles of IPv4 and IPv6 address to the roles of private and public addresses in IPv4 NATs. Since a mapping can only be created when the translator receives a packet from the IPv6 realm, it is straightforward to map the IPv6 realm in NAT64 to the private realm in IPv4 NATs, and the IPv4 realm in NAT64 to the public realm in IPv4 NATs. The close similarities among the IPv4 NAT and the NAT64 setups allow deriving immediately the mapping and filtering requirements for NAT64, by stating that endpoint independent mappings, and both endpoint independent and address dependent filtering, must be supported.

III. NAT64

NAT64 translates IPv6 packets into IPv4 packets and vice-versa. It has essentially two components, the address translation mechanism and the protocol translation mechanism. The latter, which translates IP headers fields other than the addresses, operates in a stateless manner trying to preserve as much as possible the semantics of the original field whenever possible. This was originally defined in [1] and has been updated in [14].

Address translation maps IPv6 transport addresses to IPv4 transport addresses and vice-versa. In order to create these mappings, the NAT64 box has two pools of IP 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).

NAT64 creates the mappings by using an IPv6 prefix (denoted as `Pref64::/n`) as the IPv6 address pool. Each IPv4 address is mapped into a different IPv6 address by concatenating the `Pref64::/n` with the IPv4 address being mapped and, if `n` is less than 96, a suffix with all its bits set to 0 [15]. `Pref64::/n` can be either the *Well-Known prefix* defined for this purpose (`64:ff9b::/96`) [15] or a local prefix manually assigned from the global unicast IPv6 address block of the site for this particular use. In both cases the mapping is stable over time since there is no need to re-use the IPv6 addresses, as the IPv6 address pool is large enough. By using the Well-Known prefix, the resulting IPv6 representations of IPv4 addresses are globally meaningful. This allows for any party in the Internet receiving such an address to recognize it as an IPv6 representation of an IPv4 address, and even reach the IPv4 destination if a local NAT64 service is available. The use of the Well-Know prefix is recommended in the absence of a manually configured prefix. Fig. 2 shows an example of the representation of an IPv4 address with different IPv6 prefix types.

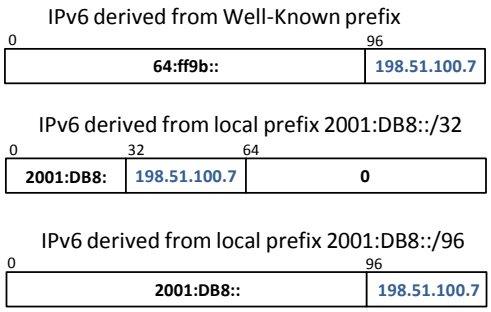

Figure 2. IPv6 address representation for 198.51.100.7.

The IPv4 address pool is normally a small prefix assigned to the NAT64's external (IPv4) interface. Because of the size of the IPv4 address space, the IPv4 address pool is not sufficient to establish permanent one-to-one mappings with IPv6 addresses. So, mappings using the IPv4 address pool are created and released dynamically.

An IPv6 initiator learns the IPv6 address representing the IPv4 target either through the DNS64, as described in the next section or by other means. Packets to that address sent by the IPv6 host are intercepted by the NAT64 device. The NAT64 associates an IPv4 transport address of its pool to the IPv6 transport address of the initiator, creating a 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.

In order to implement endpoint-independent mapping and support both endpoint-independent filtering and address-dependent filtering, NAT64 relies in two data structures to store mapping information, namely the Binding Information Base (BIB) and the Session table.

The BIB stores only mapping information. Each entry of the BIB corresponds to one transport address of an IPv6 node and the associated IPv4 transport address from the NAT64's IPv4 address pool. When an IPv6 node initiates a new communication using a source transport address that it is not in the BIB, a new entry is created. If the IPv6 node initiates a new communication with an IPv6 transport address for which

there is a BIB entry, this entry is reused for this new communication, irrespectively whether the destination IPv6 address or destination port are different from the one used in the previous communications. The result is that multiple communications involving the same IPv6 transport address are translated by the NAT64 to the same IPv4 transport address, resulting in endpoint-independent mapping.

The information contained in the BIB is enough to perform the address translation of any packet and to provide endpoint-independent filtering. However, the information contained in the BIB is not enough to perform address-dependent filtering. If this is required, the NAT64 needs to keep information about the IPv4 address of the IPv4 node involved in the communication. To support this flavor of filtering, the NAT64 relies in an additional data structure, the Session table, which contains the source and destination IPv6 transport address as well as the source and destination IPv4 transport address. This allows the NAT64 to verify if an IPv4 packet is addressed to an IPv4 transport address in use from the pool, but also that it comes from an IPv4 address already involved in a communication.

In order to comply with the requirements imposed in the minimum lifetime of the bindings [10], [11], each Session table entry has a lifetime, which in the case of UDP is set to 2 minutes and in the case of TCP is set to 2 hours.

It is apparent that bindings associated to TCP communications are "expensive" in the sense that they consume an IPv4 transport address from the reduced pool for a long time. It then seems wise to ascertain that there is a real TCP communication ongoing before creating the binding. NAT64 does so by keeping track of the TCP three-way handshake to identify TCP connection establishment before actually creating the binding. In addition, it also keeps track of the FIN exchange of the TCP connections to remove the binding even if the lifetime has not expired.

IV. DNS64

DNS64 synthesizes `AAAA` resource records (`AAAA` RRs) from `A` resource records (`A` RRs). DNS64 allows IPv6-only hosts to use the Fully-Qualified-Domain-Name of an IPv4-only node to initiate a communication.

When an IPv6-only node starts a communication, it naturally queries for a `AAAA` RR and it expects to obtain the IPv6 address of the target node. To allow an IPv6 initiator to learn the address of the responder, DNS64 is used to synthesize a `AAAA` record from the `A` record (containing the real IPv4 address of the responder). DNS64 is designed as an additional function of a DNS recursive resolver. As such, when a DNS64 enabled resolver receives a `AAAA` RR query generated by the IPv6 initiator, it searches for a `AAAA` RR. If no `AAAA` record is 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::/n` of a NAT64 to the responder's IPv4 address and if `n` is less than 96, a suffix. The synthetic `AAAA` RR is passed back to the IPv6 initiator, which starts an IPv6 communication with the IPv6 address associated to the IPv4 receiver.

The packet is routed to the NAT64 device, which creates the IPv6-to-IPv4 address mapping as described before. It is important to highlight that the DNS64 and the NAT64 do not share any state. In particular, when the DNS64 generates a synthetic response, no state is created in the NAT64. The only information shared by the NAT64 and the DNS64 is the `Pref64::/n`, which must be the same for a given domain. By default, both NAT64 and DNS64 use the Well-Known prefix, imposing no manual configuration to none of them.

One of the major challenges for DNS64 is the compatibility with DNSSEC. DNSSEC defines extensions to provide origin authentication, authenticated denial of existence, and data integrity of the DNS data. As such, it is in fundamental conflict with DNS64, since DNS64 synthesizes RRs and presents them as RRs coming from another origin. As opposed to the obsolete NAT-PT DNS-ALG, which intercepted and modified DNS packets, DNS64 is a full-fledged architectural component. Because of that, it is possible to place the DNS64 functionality within the resolution chain of the DNS to be compatible with some modes of DNSSEC by assuring that the synthesis always occurs *after* validation.

There are different configurations for a recursive resolver involving DNSSEC. A recursive resolver can be *DNSSEC-capable* or not. Moreover, a DNSSEC-capable resolver can be *validating*, i.e. performing DNSSEC data validation or not, i.e. simply passing the DNSSEC data.

Let's next consider how a DNS64 recursive resolver handles different types of DNSSEC queries:

- Queries arriving from a non DNSSEC-capable originator (Fig. 3a). A DNSSEC-capable and validating DNS64 recursive resolver can validate the data of the `A` RR before creating the synthetic `AAAA` RR.
- Queries arriving from a DNSSEC-capable but not validating originator (Fig. 3b). This is the ideal case for DNS64. If the DNS64 resolver is implementing DNSSEC validation, it validates the DNSSEC data, it creates the synthetic `AAAA` RR and signals back to the querying party that the data included is authentic. This is a fairly common case in DNSSEC deployments, where the client is not actually performing validation but it expects the local DNS server to do it on its behalf. Typically there is a secure channel between the client and its server (e.g. IPsec protection).
- Queries from a DNSSEC-capable and validating originator (Fig. 3c). In this case, the originator asks for the DNSSEC data to perform the validation itself. Because of that, the

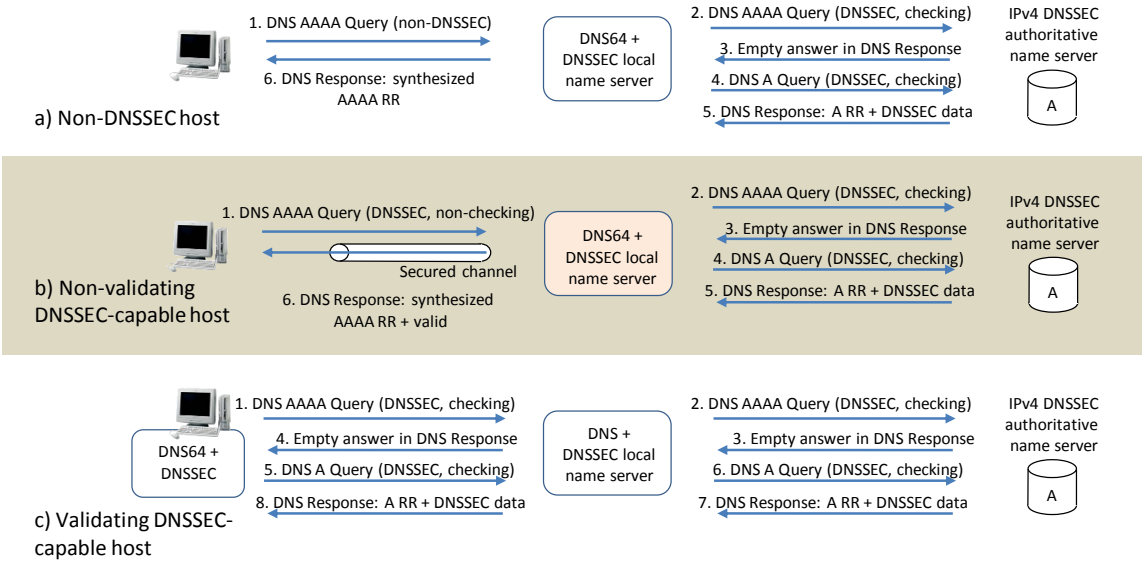

Figure 3. Different modes of DNS64 operation (detail of iterative DNS messages to the higher levels of the DNS hierarchy is not included)

DNS64 recursive resolver cannot send any synthetic AAAA RR in the response, or it would cause the validation to fail. This case can be handled by placing the DNS64 functionality in the client itself, after the validation module.

V. WALKTHROUGH

For this walkthrough we consider the topology described in Fig. 4. The NAT64 uses the Well-Known prefix 64:ff9b::/96 to map IPv4 addresses into IPv6, and has an IPv4 address T assigned to its IPv4 interface. The local name server implements the DNS64 function and uses the Well-Known prefix for its synthesis. IPv6 hosts only have stub resolvers, so they request recursive lookups to the local name server. No DNSSEC is considered.

We now describe a typical scenario in which H1 initiates a communication with H2:

1. H1 performs a DNS lookup for the IPv6 address of H2 by sending a DNS query for a AAAA record to the local DNS/DNS64 server.

2. The local DNS/DNS64 server resolves the query, discovering that there are no AAAA records for H2.

3. The DNS/DNS64 server queries for a A record for H2, obtaining the IPv4 address X.

4. The DNS/DNS64 server synthesizes a AAAA record by appending the IPv4 address X to 64:ff9b::/96, and includes this address in the response to H1.

5. After receiving the synthetic AAAA record, H1 sends a packet towards H2 from a source transport address (Y', y)[4] to a destination transport address (64:ff9b:X, x), where y and x are ports chosen by H1.

6. The packet is routed to the IPv6 interface of the NAT64 (since 64:ff9b::/96 has been associated to this interface), and the NAT64 performs the following actions:

- It selects an unused port t on its IPv4 address T and creates the BIB entry (Y', y) ↔ (T, t) and a session table entry (Y', y, 64:ff9b:X, x) ↔ (T, t, X, x)
- It translates the IPv6 header into an IPv4 header using stateless translation.
- It includes in the packet (T, t) as source transport address and (X, x) as destination transport address.

The NAT64 sends the translated packet through the IPv4 network.

7. H2 node receives the packet and responds by sending a packet with destination transport address (T, t) and source transport address (X, x).

8. The packet is routed to the NAT64 box, which looks for a Session table entry containing (T, t). When the entry is found,

- the NAT64 translates the IPv4 header into an IPv6 header using stateless translation.
- the NAT64 includes in the packet (Y', y) as destination transport address and (Pref64:X, x) as source transport address.

The translated packet is finally sent out to H1.

[4] We use a prime (') to highlight that the address is IPv6.

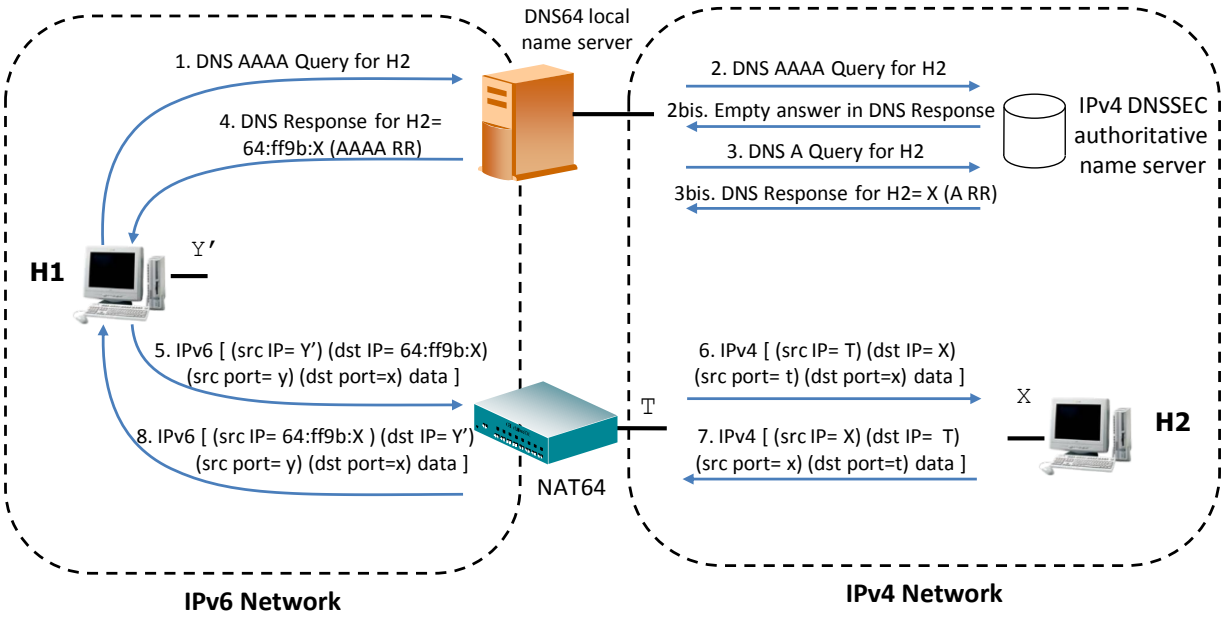

Figure 4. Walkthrough scenario (detail of iterative DNS messages to the higher levels of the DNS hierarchy is not included).

VI. Conclusions

In this paper we present the NAT64/DNS64 tool suite for IPv6 transition. NAT64 is a network address translator and protocol translator that allows communication between IPv6 and IPv4 nodes. DNS64 synthesizes `AAAA` RRs from the information available in `A` RRs for a given Fully Qualified Domain Name. This tool suite is expected to play a critical role in the IPv6 transition in the near future. There are already several open source and commercial implementations of the tool suite.

The suite is designed to support several deployment models although we expect two of them to be preeminent: The first one is to allow the internal nodes of an IPv6-only stub network to reach the public IPv4 Internet. In this case the NAT64/DNS64 functions can be provided either by the IPv6 stub network itself or by its direct provider i.e. in a Carrier Grade NAT. In the second scenario, an IPv4-only stub site decides to give access to its IPv4-only servers to clients in the IPv6 Internet. For this, NAT64 can be provisioned by the IPv4 stub network. Because the DNS server of the IPv4 site is authoritative for the local data, the DNS64 function is replaced by a DNS server with `AAAA` RRs that contain the IPv6 representation of the IP addresses assigned to the IPv4-only servers.

As DNS64/NAT64 is a replacement for the deprecated NAT-PT, to conclude the paper we compare NAT64/DNS64 and NAT-PT. In order to deal with the main limitations of NAT-PT, DNS64/NAT64 design makes a few key architectural decisions: First, the DNS64/NAT64 manages explicitly only communications initiated from the IPv6 side. Communications initiated from the IPv4 are supported through standard NAT-traversal techniques, such as STUN and TURN, since NAT64 is designed to be NAT-traversal compatible. Second, DNS64 is a full-fledged architectural component that is part of a DNS resolver. As such, it does not need to transparently intercept DNS queries. The result of these two design decisions is a more robust design, as DNS queries and data packets do not need to flow through the same path, significantly improving the reliability of the resulting network. Finally, DNS64/NAT64 use by default the Well-Known prefix that allows having a globally valid IPv6 representation of an IPv4 address. This implies that even if the IPv6 representation of an IPv4 address or the synthetic `AAAA` RR leak outside the realm of the NAT64, the receiving node can identify the address as being an IPv6 representation of the original IPv4 address.

References

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

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

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

[4] M. Bagnulo, P. Matthews, I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers", RFC-to-be 6146, July 2010.
[5] M. Bagnulo, A. Sullivan, P. Matthews, I. van Beijnum, "DNS64: DNS extensions for Network Address Translation from IPv6 Clients to IPv4 Servers", RFC-to-be 6147, July 2010.
[6] J. Rosenberg, R. Mahy, P. Matthews, D. Wing. "Session Traversal Utilities for NAT (STUN)". RFC5389, 2008.
[7] K. Egevang, P. Francis. "The IP Network Address Translator (NAT)". RFC1631, 1994.
[8] T. Hain, “Architectural Implications of NAT”, RFC2993, 2000.
[9] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti and M. Kojo, “An Experimental Study of Home Gateway Characteristics”, ACM SIGCOMM Internet Measurement Conference (IMC), Melbourne, Australia, November 1-3, 2010.
[10] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, P. Srisuresh. "NAT Behavioral Requirements for TCP". RFC5382, 2008.
[11] F. Audet, Ed., C. Jennings. "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP". RFC4787, 2007.
[12] P. Srisuresh, B. Ford, S. Sivakumar, S. Guha. "NAT Behavioral Requirements for ICMP". RFC5508, 2009.
[13] R. Mahy, P. Matthews, J. Rosenberg, “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)”, RFC 5766, 2010.
[14] X. Li, C. Bao, F. Baker, "IP/ICMP Translation Algorithm", RFC-to-be 6145, 2010.
[15] C. Bao, C. Huitema, M. Bagnulo, M. Boucadair, X. Li, "IPv6 Addressing of IPv4/IPv6 Translators", RFC6052, 2010.