DNSOP Working Group
Internet-Draft
Updates: RFC 7766 (if approved)
Intended status: Standards Track
Expires: September 14, 2017

R. Bellis
ISC
S. Cheshire
Apple Inc.
J. Dickinson
S. Dickinson
Sinodun
A. Mankin
Salesforce
T. Pusateri
Unaffiliated

March 13, 2017

DNS Session Signaling
draft-ietf-dnsop-session-signal-02

Abstract

The EDNS(0) Extension Mechanism for DNS is explicitly defined to only have "per-message" semantics. This document defines a new Session Signaling Opcode used to communicate persistent "per-session" operations, expressed using type-length-value (TLV) syntax, and defines an initial set of TLVs used to manage session timeouts and termination.

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 September 14, 2017.

Copyright Notice

Table of Contents

1. Introduction 2
2. Terminology 4
3. Protocol Details 5
3.1. Session Lifecycle and Timers 6
3.1.1. Client-Initiated Termination 9
3.1.2. Server-Initiated Termination 9
3.2. Connection Sharing 11
3.3. Message Format 13
3.4. Message Handling 15
3.5. TLV Format 17
4. Keepalive TLV 18
5. Retry Delay TLV 21
6. IANA Considerations 22
6.1. DNS Session Signaling Opcode Registration 22
6.2. DNS Session Signaling RCODE Registration 22
6.3. DNS Session Signaling Type Codes Registry 22
7. Security Considerations 23
8. Acknowledgements 23
9. References 23
9.1. Normative References 23
9.2. Informative References 24
Authors' Addresses 24

1. Introduction

The use of transports for DNS other than UDP is being increasingly specified, for example, DNS over TCP [RFC1035][RFC7766] and DNS over TLS [RFC7858]. Such transports can offer persistent, long-lived sessions and therefore when using them for transporting DNS messages it is of benefit to have a mechanism that can establish parameters associated with those sessions, such as timeouts. In such situations it is also advantageous to support server initiated messages.

The existing EDNS(0) Extension Mechanism for DNS [RFC6891] is explicitly defined to only have "per-message" semantics. Whilst EDNS(0) has been used to signal at least one session related

parameter (the EDNS(0) TCP KeepAlive option [RFC7828]) the result is less than optimal due to the restrictions imposed by the EDNS(0) semantics and the lack of server-initiated signalling. This document defines a new Session Signaling Opcode used to carry persistent "per-session" operations, expressed using type-length-value (TLV) syntax, and defines an initial set of TLVs used to manage session timeouts and termination.

With EDNS(0), multiple options may be packed into a single OPT pseudo-RR, and there is no generalized mechanism for a client to be able to tell whether a server has processed or otherwise acted upon each individual option within the combined OPT RR. The specifications for each individual option need to define how each different option is to be acknowledged, if necessary.

With Session Signaling, in contrast, there is no compelling motivation to pack multiple operations into a single message for efficiency reasons. Each Session Signaling operation is communicated in its own separate DNS message, and the transport protocol can take care of packing separate DNS messages into a single IP packet if appropriate. For example, TCP can pack multiple small DNS messages into a single TCP segment. The RCODE in each response message indicates the success or failure of the operation in question.

It should be noted that the message format for Session Signaling operations (see Section 3.3) differs from the traditional DNS packet format used for standard queries and responses. The standard twelve-octet header is used, but the four count fields (QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT) are set to zero and the corresponding sections are empty. The actual data pertaining to Session Signaling operations is appended to the end of the DNS message, following the four (empty) data sections. When displayed using today’s packet analyser tools that have not been updated to recognize the DNS Session Signaling format, this will result in the Session Signaling data being displayed as unknown additional data after the end of the DNS message. It is likely that future updates to these tools will add the ability to recognize, decode, and display the Session Signaling data.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119].

The term "connection" means a bidirectional stream of reliable, in-order messages, such as provided by using DNS over TCP [RFC1035][RFC7766] or DNS over TLS [RFC7858].

The term "session" in the context of this document means the exchange of DNS messages over a connection where:

- The connection between client and server is persistent and relatively long-lived (i.e., minutes or hours, rather than seconds).
- Either end of the connection may initiate messages to the other.

The term "server" means the software with a listening socket, awaiting incoming connection requests.

The term "client" means the software which initiates a connection to the server’s listening socket.

The terms "initiator" and "responder" correspond respectively to the initial sender and subsequent receiver of a Session Signaling request message, regardless of which was the "client" and "server" in the usual DNS sense.

The term "sender" may apply to either an initiator (when sending a Session Signaling request message) or a responder (when sending a Session Signaling response message).

Likewise, the term "receiver" may apply to either a responder (when receiving a Session Signaling request message) or an initiator (when receiving a Session Signaling response message).

Session Signaling operations are expressed using type-length-value (TLV) syntax.

A "Session Signaling Operation TLV" specifies the operation to be performed. A Session Signaling request message MUST contain exactly one Operation TLV. Depending on the operation, the corresponding Session Signaling response message MAY contain no Operation TLV, or it may contain a single corresponding response Operation TLV, with the same SSOP-TYPE as in the request message.

A "Session Signaling Modifier TLV" specifies additional parameters relating to the operation. Immediately following the Operation TLV, if present, a Session Signaling message MAY contain one or more Modifier TLVs.

The first TLV in a Session Signaling request message (and its counterpart in the corresponding Session Signaling response message, if present) is the Operation TLV. Any subsequent TLVs after this initial Operation TLV (if present) are Modifier TLVs.

If a Session Signaling request is received containing an unrecognized Operation TLV then an error response with RCODE SSOPNOTIMP (tentatively 11) is returned.

If a Session Signaling message (request or response) is received containing one or more unrecognized Modifier TLVs, the unrecognized Modifier TLVs are silently ignored.

3. Protocol Details

Session Signaling messages MUST only be carried in protocols and in environments where a session may be established according to the definition above. Standard DNS over TCP [RFC1035][RFC7766], and DNS over TLS [RFC7858] are suitable protocols. DNS over plain UDP is not appropriate since it fails on the requirement for in-order message delivery, and, in the presence of NAT gateways and firewalls with short UDP timeouts, it fails to provide a persistent bi-directional communication channel unless an excessive amount of keepalive traffic is used.

Session Signaling messages relate only to the specific session in which they are being carried. Where an application-layer middle box (e.g., a DNS proxy, forwarder, or session multiplexer) is in the path the middle box MUST NOT blindly forward the message in either direction. This does not preclude the use of these messages in the presence of an IP-layer middle box such as a NAT that rewrites IP-layer and/or transport-layer headers, but otherwise preserves the effect of a single session.

A client MAY attempt to initiate Session Signaling messages at any time on a connection; receiving a NOTIMP response in reply indicates that the server does not implement Session Signaling, and the client SHOULD NOT issue further Session Signaling messages on that connection.

A server SHOULD NOT initiate Session Signaling messages until a client-initiated Session Signaling message is received first, unless in an environment where it is known in advance by other means that

the client supports Session Signaling. This requirement is to ensure that the clients that do not support Session Signaling do not receive unsolicited inbound Session Signaling messages that they would not know how to handle.

3.1. Session Lifecycle and Timers

A session begins when a client makes a new connection to a server.

The client may perform as many DNS operations as it wishes using the newly created session. Operations SHOULD be pipelined (i.e., the client doesn't need wait for a response before sending the next message). The server MUST act on messages in the order they are received, but responses to those messages MAY be sent out of order, if appropriate.

Two timer values are associated with a session: the idle timeout, and the keepalive interval. On a new session, before any explicit Session Signaling Keepalive message exchange, the default value for both timers is 15 seconds.

At both servers and clients, the generation or reception of any complete DNS message, including DNS requests, responses, updates, or Session Signaling messages, resets both timers for that session [RFC7766], with the exception that a Session Signaling Keepalive message resets only the keepalive interval timer, not the idle timeout timer.

In addition, for as long as the client has an outstanding operation in progress, the idle timeout timer remains fixed at zero, and an idle timeout cannot occur.

For short-lived DNS operations like traditional queries and updates, an operation is considered in progress for the time between request and response, typically a period of a few hundred milliseconds at most. At the client, the idle timeout timer is set to zero upon transmission of a request and remains at zero until reception of the corresponding response. At the server, the idle timeout timer is set to zero upon reception of a request and remains at zero until transmission of the corresponding response.

For long-lived DNS operations like Push Notification subscriptions [I-D.ietf-dnssd-push], an operation is considered in progress for as long as the subscription is active, until it is cancelled. This means that a session can exist, with a Push Notification subscription active, with no messages flowing in either direction, for far longer than the idle timeout, and this is not an error. This is why there are two separate timers: the idle timeout, and the keepalive

interval. Just because a session has no traffic for an extended period of time does not automatically make that session "idle", if it has an active Push Notification subscription that is awaiting notification events.

The first timer value, the idle timeout, is the maximum time for which a client may speculatively keep a session open in the expectation that it may have future requests to send to that server.

The purpose of the idle timeout is for the server to balance its trade off between the costs of setting up new sessions and the costs of maintaining idle sessions. A server with abundant session capacity can offer a high idle timeout, to permit clients to keep a speculative session open for a long time, to save the cost of establishing a new session for future communications with that server. A server with scarce memory resources can offer a low idle timeout, to cause clients to promptly close sessions whenever they have no outstanding operations with that server, and then create a new session later when needed.

The second timer value, the keepalive interval, is the maximum permitted interval between client messages to the server if the client wishes to keep the session alive.

The purpose of the keepalive interval is to manage the generation of sufficient messages to maintain state in middleboxes (such at NAT gateways or firewalls) and for the client and server to periodically verify that they still have connectivity to each other. This allows them to clean up state when connectivity is lost, and attempt re-connection if appropriate.

For both timers, lower values of the timer result in higher network traffic and higher CPU load on the server.

For the idle timeout value, lower values result in more frequent session teardown and re-establishment. Higher values result in lower traffic and CPU load on the server, but a larger memory burden to maintain state for idle sessions.

For the keepalive interval value, lower values result in higher volume keepalive traffic. Higher values of the keepalive interval reduce traffic and CPU load, but have minimal effect on the memory burden at the server, because clients keep a session open for the same length of time (determined by the idle timeout) regardless of the level of keepalive traffic required.

The two timer values are independent. The idle timeout may be lower, the same, or higher than the keepalive interval, though in most cases

the idle timeout is expected to be shorter than the keepalive interval.

A shorter idle timeout with a longer keepalive interval signals to the client that it should not speculatively keep idle sessions open for very long for no reason, but when it does have an active reason to keep a session open, it doesn't need to be sending an agressive level of keepalive traffic. Only when the client has a very long-lived low-traffic operation outstanding like a Push Notification subscription, does the keepalive interval timer come into play, to ensure that a sufficient residual amount of traffic is generated to maintain NAT and firewall state.

A longer idle timeout with a shorter keepalive interval signals to the client that it may speculatively keep idle sessions open for a long time, but it should be sending a lot of keepalive traffic on those idle sessions. This configuration is expected to be less common.

If, at any time during the life of the session, the idle timeout value (i.e., 15 seconds by default) elapses without there being any operation active on the session, the client MUST gracefully close the connection with a TCP FIN (or equivalent for other protocols).

If, at any time during the life of the session, twice the idle timeout value (i.e., 30 seconds by default) elapses without there being any operation active on the session, the server SHOULD consider the client delinquent, and forcibly abort the session. For sessions over TCP (or over TLS over TCP), to avoid the burden of having a connection in TIME-WAIT state, instead of closing the connection gracefully with a TCP FIN the server SHOULD abort the connection with a TCP RST (or equivalent for other protocols). (In the BSD Sockets API this is achieved by setting the SO_LINGER option to zero before closing the socket.)

In this context, an operation being active on a session includes a query waiting for a response, an update waiting for a response, or an outstanding Push Notification subscription [I-D.ietf-dnssd-push], but not a Session Signaling Keepalive message exchange itself. A Session Signaling Keepalive message exchange resets only the keepalive interval timer, not the idle timeout timer.

If, at any time during the life of the session, the keepalive interval value (i.e., 15 seconds by default) elapses without any DNS messages being sent or received on a session, the client MUST take action to keep the session alive. To keep the session alive the client MUST send a Session Signaling Keepalive message (see

Section 4). A Session Signaling Keepalive message exchange resets only the keepalive interval timer, not the idle timeout timer.

If a client disconnects from the network abruptly, without cleanly closing its session, leaving long-lived outstanding operations like Push Notification subscriptions uncanceled, the server learns of this after failing to receive the required keepalive traffic from that client. If, at any time during the life of the session, twice the keepalive interval value (i.e., 30 seconds by default) elapses without any DNS messages being sent or received on a session, the server SHOULD consider the client delinquent, and forcibly abort the connection with a TCP RST (or equivalent for other protocols).

If the client wishes to keep an idle session open for longer than the default duration without having to send traffic every 15 seconds, then it uses the Session Signaling Keepalive message to request longer timeout values, as described in Section 4.

3.1.1. Client-Initiated Termination

A client is NOT required to wait until the idle-timeout timer expires before closing a session. A client MAY close a session at any time, at the client's discretion. If a client determines that it has no current or reasonably anticipated future need for an idle session, then the client SHOULD close that connection.

Upon receiving an error response from the server, a client SHOULD NOT automatically close the session. An error relating to one particular operation on a session does not necessarily imply that all other operations on that session have also failed, or that future operations will fail. The client should assume that the server will make its own decision about whether or not to close the session, based on the server's determination of whether the error condition pertains to this particular operation, or would also apply to any subsequent operations. If the server does not close the session then the client SHOULD continue to use that session for subsequent operations.

3.1.2. Server-Initiated Termination

After sending an error response to a client, the server MAY close the session, or may allow the session to remain open. For error conditions that only affect the single operation in question, the server SHOULD return an error response to the client and leave the session open for further operations. For error conditions that are likely to make all operations unsuccessful in the immediate future, the server SHOULD return an error response to the client and then

close the session by sending a Retry Delay request message, as described in Section 5.

There may be rare cases where a server is overloaded and wishes to shed load. If the server handles this by simply closing connections, the likely behaviour of clients is to detect this as a network failure, and reconnect.

To avoid this reconnection implosion, in this situation the server also sends a Retry Delay request message, with an RCODE of SERVFAIL, to inform the client of the overload situation.

After sending a Retry Delay request message, the server MUST NOT send any further messages on that session.

At the moment a server chooses to initiate a Retry Delay request message there may be DNS requests already in flight from client to server on this session, which will arrive at the server after its Retry Delay request message has been sent. The server MUST silently ignore such incoming requests, and MUST NOT generate any response messages for them. When the Retry Delay request message from the server arrives at the client, the client will determine that any DNS requests it previously sent on this session, that have not yet received a response, now will certainly not be receiving any response. Such requests should be considered failed, and should be retried at a later time, as appropriate.

A Retry Delay request message MUST NOT be initiated by a client. If a server receives a Retry Delay request message this is an error and the server MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

Upon receipt of a Retry Delay request from the server, the client MUST make note of the reconnect delay for this server, and then immediately close the connection. This is to place the burden of TCP’s TIME-WAIT state on the client.

After sending the Retry Delay request the server SHOULD allow the client five seconds to close the connection, and if the client has not closed the connection after five seconds then the server SHOULD abort the connection with a TCP RST (or equivalent for other protocols).

In the case where some, but not all, of the existing operations on a session have become invalid (perhaps because the server has been reconfigured and is no longer authoritative for some of the names), but the server is terminating all sessions en masse with a REFUSED (5) RCODE, the RECONNECT DELAY MAY be zero, indicating that the

clients SHOULD immediately attempt to re-establish operations. It is likely that some of the attempts will be successful and some will not.

In the case where a server is terminating a large number of sessions at once (e.g., if the system is restarting) and the server doesn't want to be inundated with a flood of simultaneous retries, it SHOULD send different RECONNECT delay values to each client. These adjustments MAY be selected randomly, pseudorandomly, or deterministically (e.g., incrementing the time value by one tenth of a second for each successive client, yielding a post-restart reconnection rate of ten clients per second).

Apart from the cases described above, a server MUST NOT close a session with a client, except in extraordinary error conditions. Closing the session is the client's responsibility, to be done at the client's discretion, when it so chooses. A server only closes a session under exceptional circumstances, such as when the server application software or underlying operating system is restarting, the server application terminated unexpectedly (perhaps due to a bug that makes it crash), or the server is undergoing maintenance procedures. When possible, a server SHOULD send a Retry Delay message informing the client of the reason for the session being closed, and allow the client five seconds to receive it before the server resorts to forcibly aborting the connection.

After a session is closed by the server, the client SHOULD try to reconnect, to that server, or to another suitable server, if more than one is available. If reconnecting to the same server, the client MUST respect the indicated delay before attempting to reconnect.

If a server is low on resources it MAY simply terminate a client connection with a TCP RST (or equivalent for other protocols). However, the likely behaviour of the client may be simply to reconnect immediately, putting more burden on the server. Therefore, a server SHOULD instead choose to shed client load by sending a Retry Delay message, as described above. Upon reception of the Termination TLV the client is expected to close the session, and if it does not then the server will abort the session five seconds later.

3.2. Connection Sharing

A client that supports Session Signaling SHOULD NOT make multiple connections to the same DNS server.

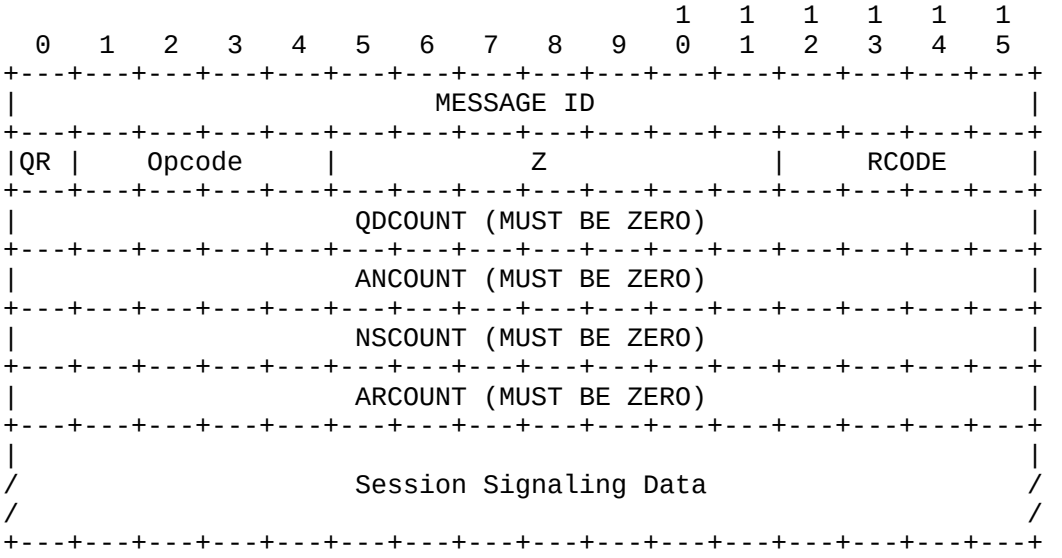
A single server may support multiple services, including DNS Updates [RFC2136], DNS Push Notifications [I-D.ietf-dnssd-push], and other

services, for one or more DNS zones. When a client discovers that the target server for several different operations is the same target hostname and port, the client SHOULD use a single shared session for all those operations. A client SHOULD NOT open multiple connections to the same target host and port just because the names being operated on are different or happen to fall within different zones. This is to reduce unnecessary connection load on the DNS server.

However, server implementers and operators should be aware that connection sharing may not be possible in all cases. A single client device may be home to multiple independent client software instances that don’t coordinate with each other. Similarly, multiple independent client devices behind the same NAT gateway will also typically appear to the DNS server as different source ports on the same client IP address. Because of these constraints, a DNS server MUST be prepared to accept multiple connections from different source ports on the same client IP address.

3.3. Message Format

A Session Signaling message begins with the standard twelve-octet DNS message header [RFC1035] with the Opcode field set to the Session Signaling Opcode (tentatively 6). However, unlike standard DNS messages, the question section, answer section, authority records section and additional records sections are all empty. The corresponding count fields (QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT) MUST be set to zero on transmission. If a Session Signaling message is received where any of the count fields are not zero, then data in the corresponding section MUST be silently skipped by the receiver (unless specified otherwise by a future update to this specification). The skipped data is silently ignored. Any skipped data in a Session Signaling request is discarded, and not copied to the corresponding sections in the Session Signaling response.



In a request the DNS Header QR bit MUST be zero. If the QR bit MUST is not zero the message is not a request.

In a response the DNS Header QR bit MUST be one. If the QR bit is not one the message is not a response.

In a request the MESSAGE ID field MUST be set to a unique value, that the initiator is not using for any other active operation on this connection. For the purposes here, a MESSAGE ID is in use on this connection if the initiator has used it in a request for which it has not yet received a response, or if if the client has used it for a subscription which it has not yet cancelled [I-D.ietf-dnssd-push].

In a response the MESSAGE ID field contain a copy of the value of the MESSAGE ID field in the request being responded to.

The DNS Header Opcode field holds the Session Signaling Opcode value (tentatively 6).

The Z bits are currently unused, and in both requests and responses the Z bits MUST be set to zero (0) on transmission and MUST be silently ignored on reception, unless a future document specifies otherwise.

In a request message (QR=0) the RCODE is generally set to zero on transmission, and silently ignored on reception, except where specified otherwise (for example, the Retry Delay operation, where the RCODE indicates the reason for termination).

The standard twelve-octet DNS message header and the four (usually) empty sections are followed by at most one Session Signaling Operation TLV. The (optional) Operation TLV may be followed by one or more Modifier TLVs, such as the Retry Delay TLV (0), which, in error responses, indicates the time interval during which the client SHOULD NOT re-attempt a failed operation.

Future specifications may define additional Modifier TLVs.

A Session Signaling message MUST contain at most one Operation TLV. In all cases a Session Signaling request message MUST contain exactly one Operation TLV, indicating the operation to be performed. In some cases a Session Signaling response message MAY contain no Operation TLV, because it is simply a response to a previous request message, and the message ID in the header is sufficient to identify the request in question. The specification for each Session Signaling operation type determines whether a response for that operation type is required to carry the Operation TLV.

If a Session Signaling request is received containing an unrecognized Operation TLV, the receiver MUST send a response with matching MESSAGE ID, and RCODE SSOPNOTIMP (tentatively 11). The response MUST NOT contain an Operation TLV.

If a Session Signaling response is received for an operation which requires that the response carry an Operation TLV, and the required Operation TLV is not the first Session Signaling TLV in the response message, then this is a fatal error and the recipient of the defective response message MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

If a Session Signaling message (request or response) is received containing one or more unrecognized Modifier TLVs, the unrecognized Modifier TLVs MUST be silently ignored, and the remainder of the message is interpreted and handled as if the unrecognized parts were not present.

Since the ARCOUNT field MUST be zero, a Session Signaling message MUST NOT contain an EDNS(0) option in the additional records section. If functionality provided by current or future EDNS(0) options is desired for Session Signaling messages, a Session Signaling Operation TLV or Modifier TLV needs to be defined to carry the necessary information.

For example, the EDNS(0) Padding Option used for security purposes [RFC7830] is not permitted in a Session Signaling message, so if message padding is desired for Session Signaling messages, a Session Signaling Modifier TLV needs to be defined to perform this function.

Similarly, a Session Signaling message MUST NOT contain a TSIG record. A TSIG record in a conventional DNS message is added as the last record in the additional records section, and carries a signature computed over the preceding message content. Since Session Signaling data appears after the additional records section, it would not be included in the signature calculation. If use of signatures with Session Signaling messages becomes necessary in the future, an explicit Session Signaling Modifier TLV needs to be defined to perform this function.

Note however that, while Session Signaling _messages_ cannot include EDNS(0) or TSIG records, a Session Signaling _session_ is typically used to carry a whole series of DNS messages of different kinds, including Session Signaling messages, and other DNS message types like Query [RFC1034][RFC1035] and Update [RFC2136], and those messages can carry EDNS(0) and TSIG records.

This specification explicitly prohibits use of the EDNS(0) TCP Keepalive Option [RFC7828] in _any_ messages sent on a Session Signaling session (because it duplicates the functionality provided by the Session Signaling Keepalive operation), but messages may contain other EDNS(0) options as appropriate.

3.4. Message Handling

On a session between a client and server that support Session Signaling, once the client has sent at least one Session Signaling message (or it is known in advance by other means that the client supports Session Signaling) either end may unilaterally send Session Signaling messages at any time, and therefore either client or server

may be the initiator of a message. The initiator MUST set the value of the QR bit in the DNS header to zero (0), and the responder MUST set it to one (1).

Every Session Signaling request message (QR=0) MUST elicit a response (QR=1), which MUST have the same MESSAGE ID in the DNS message header as in the corresponding request. Session Signaling request messages sent by the client elicit a response from the server, and Session Signaling request messages sent by the server elicit a response from the client. With most TCP implementations, the TCP data acknowledgement (generated because data has been received by TCP), the TCP window update (generated because TCP has delivered that data to the receiving software) and the DNS Session Signaling response (generated by the receiving software itself) are all combined into a single packet, so in practice the requirement that every Session Signaling request message MUST elicit a Session Signaling response incurs minimal extra cost on the network. Requiring that every request elicit a corresponding response also avoids performance problems caused by interaction between Nagle's Algorithm and Delayed Ack [NagleDA].

The namespaces of 16-bit MESSAGE IDs are disjoint in each direction. For example, it is _not_ an error for both client and server to send a request message with the same ID. In effect, the 16-bit MESSAGE ID combined with the identity of the initiator (client or server) serves as a 17-bit unique identifier for a particular operation on a session.

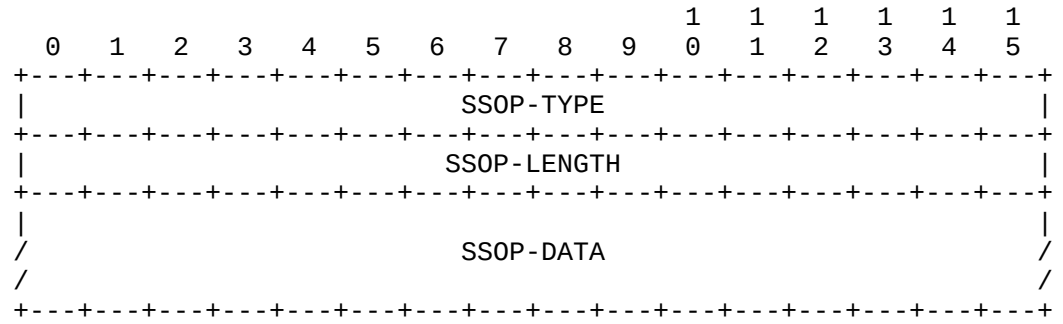
An initiator MUST NOT reuse a MESSAGE ID that is already in use for an outstanding request, unless specified otherwise by the relevant specification for the Session Signaling operation in question. At the very least, this means that a MESSAGE ID MUST NOT be reused in a particular direction on a particular session while the initiator is waiting for a response to a previous request on that session, unless specified otherwise by the relevant specification for the Session Signaling operation in question. (For a long-lived operation, such as a DNS Push Notification subscription [I-D.ietf-dnssd-push] the MESSAGE ID for the operation MUST NOT be reused for a new subscription as long as the existing subscription using that MESSAGE ID remains active.)

If a client or server receives a response (QR=1) where the MESSAGE ID does not match any of its outstanding operations, this is a fatal error and it MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

The RCODE value in a response may be one of the following values:

Code	Mnemonic	Description
0	NOERROR	Operation processed successfully
1	FORMERR	Format error
4	NOTIMP	Session Signaling not supported
5	REFUSED	Operation declined for policy reasons
11	SSOPNOTIMP	Session Signaling operation Type Code not supported

3.5. TLV Format



SSOP-TYPE: A 16 bit field in network order giving the type of the current Session Signaling TLV per the IANA DNS Session Signaling Type Codes Registry.

SSOP-LENGTH: A 16 bit field in network order giving the size in octets of SSOP-DATA.

SSOP-DATA: Type-code specific.

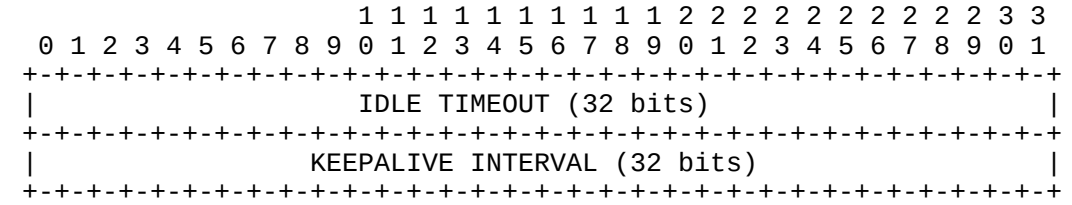
4. Keepalive TLV

The Keepalive TLV (1) performs three functions. When sent by a client, it resets a session's keepalive timer, and at the same time requests what the idle timeout and keepalive interval should be from this point forward in the session.

Once the client has sent at least one Session Signaling message (or it is known in advance by other means that the client supports Session Signaling) the Keepalive TLV also MAY be initiated by a server. When sent by a server, it resets a session's keepalive timer, and unilaterally informs the client of the new idle timeout and keepalive interval to use from this point forward in this session.

It is not required that the Keepalive TLV be used in every session. While many Session Signaling operations (such as DNS Push Notifications [I-D.ietf-dnssd-push]) will be used in conjunction with a long-lived session, not all Session Signaling operations require a long-lived session, and in some cases the default 15-second value for both idle timeout and keepalive interval may be perfectly appropriate.

The SSOP-DATA for the the Keepalive TLV is as follows:



IDLE TIMEOUT: the idle timeout for the current session, specified as a 32 bit word in network (big endian) order in units of milliseconds. This is the timeout at which the client MUST close an idle session. If the client does not gracefully close an idle session then after twice this interval the server will forcibly terminate the connection with a TCP RST (or equivalent for other protocols).

KEEPALIVE INTERVAL: the idle timeout for the current session, specified as a 32-bit word, in network (big endian) order, in units of milliseconds. This is the interval at which a client MUST generate keepalive traffic to maintain connection state. If the client does not generate the necessary keepalive traffic then after twice this interval the server will forcibly terminate the connection with a TCP RST (or equivalent for other protocols).

In a client-initiated Session Signaling Keepalive message, the idle timeout and keepalive interval contain the client's requested values. In a server response to a client-initiated message, the idle timeout and keepalive interval contain the server's chosen values, which the client MUST respect. This is modeled after the DHCP protocol, where the client requests a certain lease lifetime using DHCP option 51 [RFC2132], but the server is the ultimate authority for deciding what lease lifetime is actually granted.

In a server-initiated Session Signaling Keepalive message, the idle timeout and keepalive interval unilaterally inform the client of the new values from this point forward in this session. The client MUST generate a response to the server-initiated Session Signaling Keepalive message. The Message ID in the response message MUST match the ID from the server-initiated Session Signaling Keepalive message, and the response message MUST NOT contain any Operation TLV.

It may be appropriate for clients and servers to select different keepalive interval values depending on the nature of the network they are on.

A corporate DNS server that knows it is serving only clients on the internal network, with no intervening NAT gateways or firewalls, can impose a higher keepalive interval, because frequent keepalive traffic is not required.

A public DNS server that is serving primarily residential consumer clients, where it is likely there will be a NAT gateway on the path, may impose a lower keepalive interval, to generate more frequent keepalive traffic.

A smart client may be adaptive to its environment. A client using a private IPv4 address [RFC1918] to communicate with a DNS server at an address that is not in the same IPv4 private address block, may conclude that there is likely to be a NAT gateway on the path, and accordingly request a lower keepalive interval.

For environments where there is a NAT gateway or firewalls on the path, it is RECOMMENDED that clients request, and servers grant, a keepalive interval of 15 minutes. In other environments it is RECOMMENDED that clients request, and servers grant, a keepalive interval of 60 minutes.

Note that the lower the keepalive interval value, the higher the load on client and server. For example, an keepalive interval value of 100ms would result in a continuous stream of at least ten messages per second, in both directions, to keep the session alive. And, in this extreme example, a single packet loss and retransmission over a

long path could introduce a momentary pause in the stream of messages, long enough to cause the server to overzealously abort the connection.

Because of this concern, the server MUST NOT send a Keepalive message (either a response to a client-initiated request, or a server-initiated message) with an keepalive interval value less than ten seconds. If a client receives an Keepalive message specifying an keepalive interval value less than ten seconds this is an error and the client MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

Similarly, the server MUST NOT send a Keepalive message (either a response to a client-initiated request, or a server-initiated message) with an idle timeout value less than ten seconds. If a client receives an Keepalive message specifying an idle timeout value less than ten seconds this is an error and the client MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

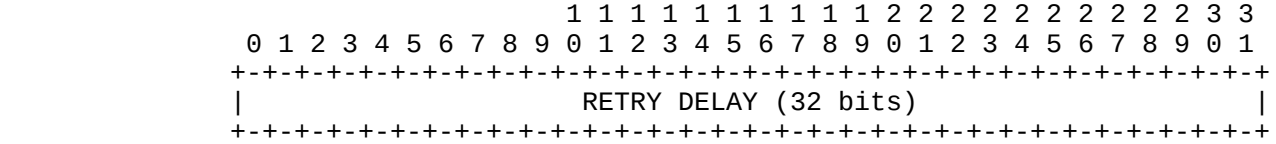
When a client is sending its second and subsequent Keepalive Session Signaling request to the server, the client SHOULD continue to request its preferred values each time. This allows flexibility, so that if conditions change during the lifefime of a session, the server can adapt its responses to better fit the client’s needs.

The Keepalive TLV (1) has similar intent to the EDNS(0) TCP Keepalive Option [RFC7828]. A client/server pair that supports Session Signaling MUST NOT use the EDNS(0) TCP KeepAlive option within any message on a session once bi-directional Session Signaling support has been confirmed. Once bi-directional Session Signaling support has been confirmed, if either client or server receives a DNS message over the session that contains an EDNS(0) TCP KeepAlive option, this is an error and the receiver of the EDNS(0) TCP KeepAlive option MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

5. Retry Delay TLV

The Retry Delay TLV (0) is used by a server to request that a client close the session, and not to reconnect for the indicated time interval. It is also used as a modifier on error responses, to indicate how long the client should wait before retrying that particular operation.

The SSOP-DATA for the the Retry Delay TLV is as follows:



RETRY DELAY: a time value, specified as a 32 bit word in network order in units of milliseconds, within which the client MUST NOT retry this operation, or retry connecting to this server.

The RECOMMENDED value is 10 seconds.

In the case of a client request that returns a nonzero RCODE value, the server MAY append a Retry Delay TLV (0) to the response, indicating the time interval during which the client SHOULD NOT attempt this operation again.

When appended to a Session Signaling response message for some client request, the Retry Delay TLV (0) is considered a Modifier TLV. The indicated time interval during which the client SHOULD NOT retry applies only to the failed operation, not to the session as a whole.

When sent in a Session Signaling request message, from server to client, the Retry Delay TLV (0) is considered an Operation TLV. It applies to the session as a whole, and the client MUST close the session, as described previously. The RCODE MUST indicate the reason for the termination. RCODE NOERROR indicates a routine shutdown. RCODE SERVFAIL indicates that the server is overloaded due to resource exhaustion. RCODE REFUSED indicates that the server has been reconfigured and is no longer able to perform one or more of the functions currently being performed on this session (for example, a DNS Push Notification server could be reconfigured such that is is no longer accepting DNS Push Notification requests for one or more of the currently subscribed names).

This document specifies only these three RCODE values for Retry Delay request. Servers sending Retry Delay requests SHOULD use one of these three values. However, future circumstances may create

situations where other RCODE values are appropriate in Retry Delay requests, so clients MUST be prepared to accept Retry Delay requests with any RCODE value.

6. IANA Considerations

6.1. DNS Session Signaling Opcode Registration

IANA are directed to assign a value (tentatively 6) in the DNS Opcodes Registry for the Session Signaling Opcode.

6.2. DNS Session Signaling RCODE Registration

IANA are directed to assign a value (tentatively 11) in the DNS RCODE Registry for the SSOPNOTIMP error code.

6.3. DNS Session Signaling Type Codes Registry

IANA are directed to create the DNS Session Signaling Type Codes Registry, with initial values as follows:

Type	Name	Status	Reference
0x0000	SSOP-RetryDelay	Standard	RFC-TBD
0x0001	SSOP-KeepAlive	Standard	RFC-TBD
0x0002 - 0x003F	Unassigned, reserved for session management TLVs		
0x0040 - 0xF7FF	Unassigned		
0xF800 - 0xFBFF	Reserved for local / experimental use		
0xFC00 - 65535	Reserved for future expansion		

Registration of additional Session Signaling Type Codes requires publication of an appropriate IETF "Standards Action" or "IESG Approval" document [RFC5226].

7. Security Considerations

If this mechanism is to be used with DNS over TLS, then these messages are subject to the same constraints as any other DNS over TLS messages and MUST NOT be sent in the clear before the TLS session is established.

8. Acknowledgements

Thanks to Tim Chown, Ralph Droms, Jan Komissar, and Manju Shankar Rao for their helpful contributions to this document.

9. References

9.1. Normative References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <http://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

[RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <http://www.rfc-editor.org/info/rfc1918>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, DOI 10.17487/RFC2132, March 1997, <http://www.rfc-editor.org/info/rfc2132>.

[RFC2136] Vixie, P., Ed., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, DOI 10.17487/RFC2136, April 1997, <http://www.rfc-editor.org/info/rfc2136>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC6891] Damas, J., Graff, M., and P. Vixie, "Extension Mechanisms for DNS (EDNS(0))", STD 75, RFC 6891, DOI 10.17487/RFC6891, April 2013, <http://www.rfc-editor.org/info/rfc6891>.

[RFC7766] Dickinson, J., Dickinson, S., Bellis, R., Mankin, A., and D. Wessels, "DNS Transport over TCP - Implementation Requirements", RFC 7766, DOI 10.17487/RFC7766, March 2016, <http://www.rfc-editor.org/info/rfc7766>.

[RFC7828] Wouters, P., Abley, J., Dickinson, S., and R. Bellis, "The edns-tcp-keepalive EDNS0 Option", RFC 7828, DOI 10.17487/RFC7828, April 2016, <http://www.rfc-editor.org/info/rfc7828>.

[RFC7830] Mayrhofer, A., "The EDNS(0) Padding Option", RFC 7830, DOI 10.17487/RFC7830, May 2016, <http://www.rfc-editor.org/info/rfc7830>.

9.2. Informative References

[I-D.ietf-dnssd-push] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-09 (work in progress), October 2016.

[NagleDA] Cheshire, S., "TCP Performance problems caused by interaction between Nagle's Algorithm and Delayed ACK", May 2005, <http://www.stuartcheshire.org/papers/nagledelayedack/>.

[RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <http://www.rfc-editor.org/info/rfc7858>.

Authors' Addresses

Ray Bellis
Internet Systems Consortium, Inc.
950 Charter Street
Redwood City CA 94063
USA

Phone: +1 650 423 1200
Email: ray@isc.org

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino CA 95014
USA

Phone: +1 408 974 3207
Email: cheshire@apple.com

John Dickinson
Sinodun Internet Technologies
Magadalen Centre
Oxford Science Park
Oxford OX4 4GA
United Kingdom

Email: jad@sinodun.com

Sara Dickinson
Sinodun Internet Technologies
Magadalen Centre
Oxford Science Park
Oxford OX4 4GA
United Kingdom

Email: sara@sinodun.com

Allison Mankin
Salesforce

Email: allison.mankin@gmail.com

Tom Pusateri
Unaffiliated

Phone: +1 843 473 7394
Email: pusateri@bangj.com

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

S. Cheshire
Apple Inc.

March 13, 2017

Discovery Proxy for Multicast DNS-Based Service Discovery
draft-ietf-dnssd-hybrid-06

Abstract

This document specifies a mechanism that uses Multicast DNS to automatically populate the wide-area unicast Domain Name System namespace with records describing devices and services found on the local link.

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 September 14, 2017.

Copyright Notice

Table of Contents

1. Introduction 3
2. Operational Analogy 6
3. Conventions and Terminology Used in this Document 7
4. Compatibility Considerations 7
5. Discovery Proxy Operation 8
5.1. Delegated Subdomain for Service Discovery Records 9
5.2. Domain Enumeration 11
5.2.1. Domain Enumeration via Unicast Queries 11
5.2.2. Domain Enumeration via Multicast Queries 13
5.3. Delegated Subdomain for LDH Host Names 14
5.4. Delegated Subdomain for Reverse Mapping 16
5.5. Data Translation 18
5.5.1. DNS TTL limiting 18
5.5.2. Suppressing Unusable Records 19
5.5.3. NSEC and NSEC3 queries 20
5.5.4. No Text Encoding Translation 20
5.5.5. Application-Specific Data Translation 21
5.6. Answer Aggregation 23
6. Administrative DNS Records 26
6.1. DNS SOA (Start of Authority) Record 26
6.2. DNS NS Records 27
6.3. DNS SRV Records 27
7. DNSSEC Considerations 28
7.1. On-line signing only 28
7.2. NSEC and NSEC3 Records 28
8. IPv6 Considerations 29
9. Security Considerations 30
9.1. Authenticity 30
9.2. Privacy 30
9.3. Denial of Service 31
10. Intelectual Property Rights 32
11. IANA Considerations 32
12. Acknowledgments 32
13. References 33
13.1. Normative References 33
13.2. Informative References 34
Appendix A. Implementation Status 36
A.1. Already Implemented and Deployed 36
A.2. Already Implemented 36
A.3. Partially Implemented 36
A.4. Not Yet Implemented 37
Author's Address 37

1. Introduction

Multicast DNS [RFC6762] and its companion technology DNS-based Service Discovery [RFC6763] were created to provide IP networking with the ease-of-use and autoconfiguration for which AppleTalk was well known [RFC6760] [ZC].

For a small home network consisting of just a single link (or a few physical links bridged together to appear as a single logical link from the point of view of IP) Multicast DNS [RFC6762] is sufficient for client devices to look up the ".local" host names of peers on the same home network, and to use Multicast DNS-Based Service Discovery (DNS-SD) [RFC6763] to discover services offered on that home network.

For a larger network consisting of multiple links that are interconnected using IP-layer routing instead of link-layer bridging, link-local Multicast DNS alone is insufficient because link-local Multicast DNS packets, by design, are not propagated onto other links.

Using link-local multicast packets for Multicast DNS was a conscious design choice [RFC6762]. Even when limited to a single link, multicast traffic is still generally considered to be more expensive than unicast, because multicast traffic impacts many devices, instead of just a single recipient. In addition, with some technologies like Wi-Fi [IEEE-11], multicast traffic is inherently less efficient and less reliable than unicast, because Wi-Fi multicast traffic is sent using the lower data rates, and is not acknowledged. Multiplying the amount of expensive multicast traffic by flooding it across multiple links would make the traffic load even worse.

Partitioning the network into many small links curtails the spread of expensive multicast traffic, but limits the discoverability of services. Using a very large local link with thousands of hosts enables better service discovery, but at the cost of larger amounts of multicast traffic.

Performing DNS-Based Service Discovery using purely Unicast DNS is more efficient and doesn’t require excessively large multicast domains, but requires that the relevant data be available in the Unicast DNS namespace. The Unicast DNS namespace in question could fall within a traditionally assigned globally unique domain name, or could use a private local unicast domain name such as ".home" [HOME].)

In the DNS-SD specification [RFC6763], Section 10 ("Populating the DNS with Information") discusses various possible ways that a service’s PTR, SRV, TXT and address records can make their way into

the Unicast DNS namespace, including manual zone file configuration [RFC1034] [RFC1035], DNS Update [RFC2136] [RFC3007] and proxies of various kinds.

Making the relevant data available in the Unicast DNS namespace by manual DNS configuration (as has been done for many years at IETF meetings to advertise the IETF Terminal Room printer) is labor intensive, error prone, and requires a reasonable degree of DNS expertise.

Populating the Unicast DNS namespace via DNS Update by the devices offering the services themselves requires configuration of DNS Update keys on those devices, which has proven onerous and impractical for simple devices like printers and network cameras.

Hence, to facilitate efficient and reliable DNS-Based Service Discovery, a compromise is needed that combines the ease-of-use of Multicast DNS with the efficiency and scalability of Unicast DNS.

This document specifies a type of proxy called a "Multicast Discovery Proxy" (or just "Discovery Proxy") that uses Multicast DNS [RFC6762] to discover Multicast DNS records on its local link, and makes corresponding DNS records visible in the Unicast DNS namespace.

In principle, similar mechanisms could be defined using other local service discovery protocols, to discover local information and then make corresponding DNS records visible in the Unicast DNS namespace. Such mechanisms for other local service discovery protocols could be addressed in future documents.

The design of the Discovery Proxy is guided by the previously published Requirements for Scalable DNS-Based Service [RFC7558].

In simple terms, a descriptive DNS name is chosen for each link in an organization. Using a DNS NS record, responsibility for that DNS name is delegated to a Discovery Proxy physically attached to that link. Now, when a remote client issues a unicast query for a name falling within the delegated subdomain, the normal DNS delegation mechanism results in the unicast query arriving at the Discovery Proxy, since it has been declared authoritative for those names. Now, instead of consulting a textual zone file on disk to discover the answer to the query, as a traditional DNS server would, a Discovery Proxy consults its local link, using Multicast DNS, to find the answer to the question.

For fault tolerance reasons there may be more than one Discovery Proxy serving a given link.

Note that the Discovery Proxy uses a "pull" model. The local link is not queried using Multicast DNS until some remote client has requested that data. In the idle state, in the absence of client requests, the Discovery Proxy sends no packets and imposes no burden on the network. It operates purely "on demand".

An alternative proposal that has been suggested is a proxy that performs DNS updates to a remote DNS server on behalf of the Multicast DNS devices on the local network. The difficulty of this is that the proxy would have to be issuing all possible Multicast DNS queries all the time, to discover all the answers it needed to push up to the remote DNS server using DNS Update. It would thus generate very high load on the network continuously, even when there were no clients with any interest in that data.

Hence, having a model where the query comes to the Discovery Proxy is much more efficient than a model where the Discovery Proxy pushes the answers out to some other remote DNS server.

A client seeking to discover services and other information achieves this by sending traditional DNS queries to the Discovery Proxy, or by sending DNS Push Notification subscription requests [PUSH].

2. Operational Analogy

A Discovery Proxy does not operate as a multicast relay, or multicast forwarder. There is no danger of multicast forwarding loops that result in traffic storms, because no multicast packets are forwarded. A Discovery Proxy operates as a *proxy* for a remote client, performing queries on its behalf and reporting the results back.

A reasonable analogy would be making a telephone call to a colleague at your workplace and saying, "I'm out of the office right now. Would you mind bringing up a printer browser window and telling me the names of the printers you see?" That entails no risk of a forwarding loop causing a traffic storm, because no multicast packets are sent over the telephone call.

A similar analogy, instead of enlisting another human being to initiate the service discovery operation on your behalf, would be to log into your own desktop work computer using screen sharing, and then run the printer browser yourself to see the list of printers. Or log in using ssh and type "dns-sd -B _ipp._tcp" and observe the list of discovered printer names. In neither case is there any risk of a forwarding loop causing a traffic storm, because no multicast packets are being sent over the screen sharing or ssh connection.

The Discovery Proxy provides another way of performing remote queries, just using a different protocol instead of screen sharing or ssh.

When the Discovery Proxy software performs Multicast DNS operations, the exact same Multicast DNS caching mechanisms are applied as when any other client software on that Discovery Proxy device performs Multicast DNS operations, whether that be running a printer browser client locally, or a remote user running the printer browser client via a screen sharing connection, or a remote user logged in via ssh running a command-line tool like "dns-sd".

3. Conventions and Terminology Used in this Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119].

The Discovery Proxy builds on Multicast DNS, which works between hosts on the same link. A set of hosts is considered to be "on the same link" if:

- when any host A from that set sends a packet to any other host B in that set, using unicast, multicast, or broadcast, the entire link-layer packet payload arrives unmodified, and
- a broadcast sent over that link by any host from that set of hosts can be received by every other host in that set

The link-layer *header* may be modified, such as in Token Ring Source Routing [IEEE-5], but not the link-layer *payload*. In particular, if any device forwarding a packet modifies any part of the IP header or IP payload then the packet is no longer considered to be on the same link. This means that the packet may pass through devices such as repeaters, bridges, hubs or switches and still be considered to be on the same link for the purpose of this document, but not through a device such as an IP router that decrements the IP TTL or otherwise modifies the IP header.

4. Compatibility Considerations

No changes to existing devices are required to work with a Discovery Proxy.

Existing devices that advertise services using Multicast DNS work with Discovery Proxy.

Existing clients that support DNS-Based Service Discovery over Unicast DNS work with Discovery Proxy. Service Discovery over Unicast DNS was introduced in Mac OS X 10.4 in April 2005, as is included in Apple products introduced since then, including iPhone and iPad, as well as products from other vendors, such as Microsoft Windows 10.

5. Discovery Proxy Operation

In a typical configuration, a Discovery Proxy is configured to be authoritative [RFC1034] [RFC1035] for four DNS subdomains, and authority for these subdomains is delegated to it via NS records:

A DNS subdomain for service discovery records.
This subdomain name may contain rich text, including spaces and other punctuation. This is because this subdomain name is used only in graphical user interfaces, where rich text is appropriate.

A DNS subdomain for host name records.
This subdomain name SHOULD be limited to letters, digits and hyphens, to facilitate convenient use of host names in command-line interfaces.

A DNS subdomain for IPv6 Reverse Mapping records.
This subdomain name will be a name that ends in "ip6.arpa."

A DNS subdomain for IPv4 Reverse Mapping records.
This subdomain name will be a name that ends in "in-addr.arpa."

In an enterprise network the naming and delegation of these subdomains is typically performed by conscious action of the network administrator. In a home network naming and delegation would typically be performed using some automatic configuration mechanism such as HNCP [RFC7788].

These three varieties of delegated subdomains (service discovery, host names, and reverse mapping) are described below in sections Section 5.1, Section 5.3 and Section 5.4.

How a client discovers where to issue its service discovery queries is described below in section Section 5.2.

5.1. Delegated Subdomain for Service Discovery Records

In its simplest form, each link in an organization is assigned a unique Unicast DNS domain name, such as "Building 1.example.com" or "2nd Floor.Building 3.example.com". Grouping multiple links under a single Unicast DNS domain name is to be specified in a future companion document, but for the purposes of this document, assume that each link has its own unique Unicast DNS domain name. In a graphical user interface these names are not displayed as strings with dots as shown above, but something more akin to a typical file browser graphical user interface (which is harder to illustrate in a text-only document) showing folders, subfolders and files in a file system.

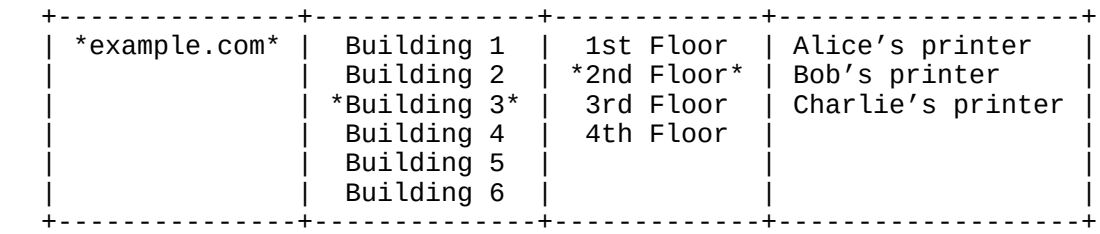


Figure 1: Illustrative GUI

Each named link in an organization has one or more Discovery Proxies which serve it. This Discovery Proxy function for each link could be performed by a device like a router or switch that is physically attached to that link. In the parent domain, NS records are used to delegate ownership of each defined link name (e.g., "Building 1.example.com") to the one or more Discovery Proxies that serve the named link. In other words, the Discovery Proxies are the authoritative name servers for that subdomain.

With appropriate VLAN configuration [IEEE-1Q] a single Discovery Proxy device could have a logical presence on many links, and serve as the Discovery Proxy for all those links. In such a configuration the Discovery Proxy device would have a single physical Ethernet [IEEE-3] port, configured as a VLAN trunk port, which would appear to software on that device as multiple virtual Ethernet interfaces, one connected to each of the VLAN links.

When a DNS-SD client issues a Unicast DNS query to discover services in a particular Unicast DNS subdomain (e.g., "_printer._tcp.Building 1.example.com. PTR ?") the normal DNS delegation mechanism results in that query being forwarded until it reaches the delegated authoritative name server for that subdomain, namely the Discovery Proxy on the link in question. Like a conventional Unicast DNS server, a Discovery Proxy implements the usual Unicast DNS protocol [RFC1034] [RFC1035] over UDP and TCP. However, unlike a conventional Unicast DNS server that generates answers from the data in its manually-configured zone file, a Discovery Proxy generates answers using Multicast DNS. A Discovery Proxy does this by consulting its Multicast DNS cache and/or issuing Multicast DNS queries for the corresponding Multicast DNS name, type and class, (e.g., in this case, "_printer._tcp.local. PTR ?"). Then, from the received Multicast DNS data, the Discovery Proxy synthesizes the appropriate Unicast DNS response. How long the Discovery Proxy should wait to accumulate Multicast DNS responses is described below in section Section 5.6.

Naturally, the existing Multicast DNS caching mechanism is used to minimize unnecessary Multicast DNS queries on the wire. The Discovery Proxy is acting as a client of the underlying Multicast DNS subsystem, and benefits from the same caching and efficiency measures as any other client using that subsystem.

5.2. Domain Enumeration

A DNS-SD client performs Domain Enumeration [RFC6763] via certain PTR queries, using both unicast and multicast. If it receives a Domain Name configuration via DHCP option 15 [RFC2132], then it issues unicast queries using this domain. It issues unicast queries using names derived from its IPv6 prefix(es) and IPv4 subnet address(es). These are described below in Section 5.2.1. It also issues multicast Domain Enumeration queries in the "local" domain [RFC6762]. These are described below in Section 5.2.2. The results of all the Domain Enumeration queries are combined for Service Discovery purposes.

5.2.1. Domain Enumeration via Unicast Queries

The administrator creates Domain Enumeration PTR records [RFC6763] to inform clients of available service discovery domains, e.g.,:

```
b._dns-sd._udp.example.com.    PTR   Building 1.example.com.
                               PTR   Building 2.example.com.
                               PTR   Building 3.example.com.
                               PTR   Building 4.example.com.

db._dns-sd._udp.example.com.   PTR   Building 1.example.com.

lb._dns-sd._udp.example.com.   PTR   Building 1.example.com.
```

The "b" ("browse") records tell the client device the list of browsing domains to display for the user to select from and the "db" ("default browse") record tells the client device which domain in that list should be selected by default. The "lb" ("legacy browse") record tells the client device which domain to automatically browse on behalf of applications that don’t implement UI for multi-domain browsing (which is most of them, as of 2017). The "lb" domain is often the same as the "db" domain, or sometimes the "db" domain plus one or more others that should be included in the list of automatic browsing domains for legacy clients.

DNS responses are limited to a maximum size of 65535 bytes. This limits the maximum number of domains that can be returned for a Domain Enumeration query, as follows:

A DNS response header is 12 bytes. That's typically followed by a single qname (up to 256 bytes) plus qtype (2 bytes) and qclass (2 bytes), leaving 65275 for the Answer Section.

An Answer Section Resource Record consists of:

- Owner name, encoded as a two-byte compression pointer
- Two-byte rrtype (type PTR)
- Two-byte rrclass (class IN)
- Four-byte ttl
- Two-byte rdlength
- rdata (domain name, up to 256 bytes)

This means that each Resource Record in the Answer Section can take up to 268 bytes total, which means that the Answer Section can contain, in the worst case, no more than 243 domains.

In a more typical scenario, where the domain names are not all maximum-sized names, and there is some similarity between names so that reasonable name compression is possible, each Answer Section Resource Record may average 140 bytes, which means that the Answer Section can contain up to 466 domains.

It is anticipated that this should be sufficient for even a large corporate network or university campus.

5.2.2. Domain Enumeration via Multicast Queries

Since a Discovery Proxy exists on many, if not all, the links in an enterprise, it offers an additional way to provide Domain Enumeration data for clients.

A Discovery Proxy can be configured to generate Multicast DNS responses for the following Multicast DNS Domain Enumeration queries issued by clients:

```
b._dns-sd._udp.local.    PTR   ?
db._dns-sd._udp.local.   PTR   ?
lb._dns-sd._udp.local.   PTR   ?
```

This provides the ability for Discovery Proxies to indicate recommended browsing domains to DNS-SD clients on a per-link granularity. In some enterprises it may be preferable to provide this per-link configuration data in the form of Discovery Proxy configuration, rather than populating the Unicast DNS servers with the same data (in the "ip6.arpa" or "in-addr.arpa" domains).

Regardless of how the network operator chooses to provide this configuration data, clients will perform Domain Enumeration via both unicast and multicast queries, and then combine the results of these queries.

5.3. Delegated Subdomain for LDH Host Names

DNS-SD service instance names and domains are allowed to contain arbitrary Net-Unicode text [RFC5198], encoded as precomposed UTF-8 [RFC3629].

Users typically interact with service discovery software by viewing a list of discovered service instance names on a display, and selecting one of them by pointing, touching, or clicking. Similarly, in software that provides a multi-domain DNS-SD user interface, users view a list of offered domains on the display and select one of them by pointing, touching, or clicking. To use a service, users don't have to remember domain or instance names, or type them; users just have to be able to recognize what they see on the display and touch or click on the thing they want.

In contrast, host names are often remembered and typed. Also, host names have historically been used in command-line interfaces where spaces can be inconvenient. For this reason, host names have traditionally been restricted to letters, digits and hyphens (LDH), with no spaces or other punctuation.

While we still want to allow rich text for DNS-SD service instance names and domains, it is advisable, for maximum compatibility with existing usage, to restrict host names to the traditional letter-digit-hyphen rules. This means that while a service name "My Printer._ipp._tcp.Building 1.example.com" is acceptable and desirable (it is displayed in a graphical user interface as an instance called "My Printer" in the domain "Building 1" at "example.com"), a host name "My-Printer.Building 1.example.com" is less desirable (because of the space in "Building 1").

To accomodate this difference in allowable characters, a Discovery Proxy SHOULD support having two separate subdomains delegated to it for each link it serves, one whose name is allowed to contain arbitrary Net-Unicode text [RFC5198], and a second more constrained subdomain whose name is restricted to contain only letters, digits, and hyphens, to be used for host name records (names of 'A' and 'AAAA' address records).

For example, a Discovery Proxy could have the two subdomains "Building 1.example.com" and "bldg1.example.com" delegated to it. The Discovery Proxy would then translate these two Multicast DNS records:

```
My Printer._ipp._tcp.local. SRV 0 0 631 prnt.local.
prnt.local.                 A   203.0.113.2
```

into Unicast DNS records as follows:

```
My Printer._ipp._tcp.Building 1.example.com.
                            SRV 0 0 631 prnt.bldg1.example.com.
prnt.bldg1.example.com.     A   203.0.113.2
```

Note that the SRV record name is translated using the rich-text domain name ("Building 1.example.com") and the address record name is translated using the LDH domain ("bldg1.example.com").

A Discovery Proxy MAY support only a single rich text Net-Unicode domain, and use that domain for all records, including 'A' and 'AAAA' address records, but implementers choosing this option should be aware that this choice may produce host names that are awkward to use in command-line environments. Whether this is an issue depends on whether users in the target environment are expected to be using command-line interfaces.

A Discovery Proxy MUST NOT be restricted to support only a letter-digit-hyphen subdomain, because that results in an unnecessarily poor user experience.

5.4. Delegated Subdomain for Reverse Mapping

A Discovery Proxy can facilitate easier management of reverse mapping domains, particularly for IPv6 addresses where manual management may be more onerous than it is for IPv4 addresses.

To achieve this, in the parent domain, NS records are used to delegate ownership of the appropriate reverse mapping domain to the Discovery Proxy. In other words, the Discovery Proxy becomes the authoritative name server for the reverse mapping domain. For fault tolerance reasons there may be more than one Discovery Proxy serving a given link.

For example, if a given link is using the
IPv6 prefix 2001:0DB8:1234:5678/64,
then the domain "8.7.6.5.4.3.2.1.8.b.d.0.1.0.0.2.ip6.arpa"
is delegated to the Discovery Proxy for that link.

If a given link is using the IPv4 subnet 203.0.113/24,
then the domain "113.0.203.in-addr.arpa"
is delegated to the Discovery Proxy for that link.

When a reverse mapping query arrives at the Discovery Proxy, it issues the identical query on its local link as a Multicast DNS query. The mechanism to force an apparently unicast name to be resolved using link-local Multicast DNS varies depending on the API set being used. For example, in the "/usr/include/dns_sd.h" APIs (available on macOS, iOS, Bonjour for Windows, Linux and Android), using kDNSServiceFlagsForceMulticast indicates that the DNSServiceQueryRecord() call should perform the query using Multicast DNS. Other APIs sets have different ways of forcing multicast queries. When the host owning that IPv6 or IPv4 address responds with a name of the form "something.local", the Discovery Proxy rewrites that to use its configured LDH host name domain instead of "local", and returns the response to the caller.

For example, a Discovery Proxy with the two subdomains "113.0.203.in-addr.arpa" and "bldg1.example.com" delegated to it would translate this Multicast DNS record:

```
2.113.0.203.in-addr.arpa. PTR prnt.local.
```

into this Unicast DNS response:

```
2.113.0.203.in-addr.arpa. PTR prnt.bldg1.example.com.
```

Subsequent queries for the prnt.bldg1.example.com address record, falling as it does within the bldg1.example.com domain, which is delegated to the Discovery Proxy, will arrive at the Discovery Proxy, where they are answered by issuing Multicast DNS queries and using the received Multicast DNS answers to synthesize Unicast DNS responses, as described above.

5.5. Data Translation

Generating the appropriate Multicast DNS queries involves, at the very least, translating from the configured DNS domain (e.g., "Building 1.example.com") on the Unicast DNS side to "local" on the Multicast DNS side.

Generating the appropriate Unicast DNS responses involves translating back from "local" to the appropriate configured DNS Unicast domain.

Other beneficial translation and filtering operations are described below.

5.5.1. DNS TTL limiting

For efficiency, Multicast DNS typically uses moderately high DNS TTL values. For example, the typical TTL on DNS-SD PTR records is 75 minutes. What makes these moderately high TTLs acceptable is the cache coherency mechanisms built in to the Multicast DNS protocol which protect against stale data persisting for too long. When a service shuts down gracefully, it sends goodbye packets to remove its PTR records immediately from neighbouring caches. If a service shuts down abruptly without sending goodbye packets, the Passive Observation Of Failures (POOF) mechanism described in Section 10.5 of the Multicast DNS specification [RFC6762] comes into play to purge the cache of stale data.

A traditional Unicast DNS client on a remote link does not get to participate in these Multicast DNS cache coherency mechanisms on the local link. For traditional Unicast DNS queries (those received without using Long-Lived Query [LLQ] or DNS Push Notification [PUSH]) the DNS TTLs reported in the resulting Unicast DNS response SHOULD be capped to be no more than ten seconds.

Similarly, for negative responses, the negative caching TTL indicated in the SOA record [RFC2308] should also be ten seconds (Section 6.1).

This value of ten seconds is chosen based on user-experience considerations.

For negative caching, suppose a user is attempting to access a remote device (e.g., a printer), and they are unsuccessful because that device is powered off. Suppose they then place a telephone call and ask for the device to be powered on. We want the device to become available to the user within a reasonable time period. It is reasonable to expect it to take on the order of ten seconds for a simple device with a simple embedded operating system to power on. Once the device is powered on and has announced its presence on the

network via Multicast DNS, we would like it to take no more than a further ten seconds for stale negative cache entries to expire from Unicast DNS caches, making the device available to the user desiring to access it.

Similar reasoning applies to capping positive TTLs at ten seconds. In the event of a device moving location, getting a new DHCP address, or other renumbering events, we would like the updated information to be available to remote clients in a relatively timely fashion.

However, network administrators should be aware that many recursive (caching) DNS servers by default are configured to impose a minimum TTL of 30 seconds. If stale data appears to be persisting in the network to the extent that it adversely impacts user experience, network administrators are advised to check the configuration of their recursive DNS servers.

For received Unicast DNS queries that use LLQ or DNS Push Notification, the Multicast DNS record's TTL SHOULD be returned unmodified, because the Push Notification channel exists to inform the remote client as records come and go. For further details about Long-Lived Queries, and its newer replacement, DNS Push Notifications, see Section 5.6.

5.5.2. Suppressing Unusable Records

A Discovery Proxy SHOULD suppress Unicast DNS answers for records that are not useful outside the local link. For example, DNS AAAA and A records for IPv6 link-local addresses [RFC4862] and IPv4 link-local addresses [RFC3927] SHOULD be suppressed. Similarly, for sites that have multiple private address realms [RFC1918], in cases where the Discovery Proxy can determine that the querying client is in a different address realm, private addresses MUST NOT be communicated to that client. IPv6 Unique Local Addresses [RFC4193] SHOULD be suppressed in cases where the Discovery Proxy can determine that the querying client is in a different IPv6 address realm.

By the same logic, DNS SRV records that reference target host names that have no addresses usable by the requester should be suppressed, and likewise, DNS PTR records that point to unusable SRV records should be similarly be suppressed.

5.5.3. NSEC and NSEC3 queries

Since a Discovery Proxy only knows what names exist on the local link by issuing queries for them, and since it would be impractical to issue queries for every possible name just to find out which names exist and which do not, a Discovery Proxy cannot programatically generate the traditional NSEC and NSEC3 records which assert the nonexistence of a large range of names.

When queried for an NSEC or NSEC3 record type, the Discovery Proxy issues a qtype "ANY" query using Multicast DNS on the local link, and then generates an NSEC or NSEC3 response signifying which record types do and do not exist just the specific name queried, and no others.

Multicast DNS NSEC records received on the local link MUST NOT be forwarded unmodified to a unicast querier, because there are slight differences in the NSEC record data. In particular, Multicast DNS NSEC records do not have the NSEC bit set in the Type Bit Map, whereas conventional Unicast DNS NSEC records do have the NSEC bit set.

5.5.4. No Text Encoding Translation

A Discovery Proxy does no translation between text encodings. Specifically, a Discovery Proxy does no translation between Punycode and UTF-8, either in the owner name of DNS records, or anywhere in the RDATA of DNS records (such as the RDATA of PTR records, SRV records, NS records, or other record types like TXT, where it is ambiguous whether the RDATA may contain DNS names). All bytes are treated as-is, with no attempt at text encoding translation. A client implementing DNS-based Service Discovery [RFC6763] will use UTF-8 encoding for its service discovery queries, which the Discovery Proxy passes through without any text encoding translation to the Multicast DNS subsystem. Responses from the Multicast DNS subsystem are similarly returned, without any text encoding translation, back to the requesting client.

5.5.5. Application-Specific Data Translation

There may be cases where Application-Specific Data Translation is appropriate.

For example, AirPrint printers tend to advertise fairly verbose information about their capabilities in their DNS-SD TXT record. TXT record sizes in the range 500-1000 bytes are not uncommon. This information is a legacy from LPR printing, because LPR does not have in-band capability negotiation, so all of this information is conveyed using the DNS-SD TXT record instead. IPP printing does have in-band capability negotiation, but for convenience printers tend to include the same capability information in their IPP DNS-SD TXT records as well. For local mDNS use this extra TXT record information is inefficient, but not fatal. However, when a Discovery Proxy aggregates data from multiple printers on a link, and sends it via unicast (via UDP or TCP) this amount of unnecessary TXT record information can result in large responses. A DNS reply over TCP carrying information about 70 printers with an average of 700 bytes per printer adds up to about 50 kilobytes of data. Therefore, a Discovery Proxy that is aware of the specifics of an application-layer protocol such as AirPrint (which uses IPP) can elide unnecessary key/value pairs from the DNS-SD TXT record for better network efficiency.

Also, the DNS-SD TXT record for many printers contains an "adminurl" key something like "adminurl=http://printername.local/status.html". For this URL to be useful outside the local link, the embedded ".local" hostname needs to be translated to an appropriate name with larger scope. It is easy to translate ".local" names when they appear in well-defined places, either as a record’s name, or in the rdata of record types like PTR and SRV. In the printing case, some application-specific knowledge about the semantics of the "adminurl" key is needed for the Discovery Proxy to know that it contains a name that needs to be translated. This is somewhat analogous to the need for NAT gateways to contain ALGs (Application-Specific Gateways) to facilitate the correct translation of protocols that embed addresses in unexpected places.

As is the case with NAT ALGs, protocol designers are advised to avoid communicating names and addresses in nonstandard locations, because those "hidden" names and addresses are at risk of not being translated when necessary, resulting in operational failures. In the printing case, the operational failure of failing to translate the "adminurl" key correctly is that, when accessed from a different link, printing will still work, but clicking the "Admin" UI button will fail to open the printer’s administration page. Rather than duplicating the host name from the service’s SRV record in its

"adminurl" key, thereby having the same host name appear in two places, a better design might have been to omit the host name from the "adminurl" key, and instead have the client implicitly substitute the target host name from the service's SRV record in place of a missing host name in the "adminurl" key. That way the desired host name only appears once, and it is in a well-defined place where software like the Discovery Proxy is expecting to find it.

Note that this kind of Application-Specific Data Translation is expected to be very rare. It is the exception, rather than the rule. This is an example of a common theme in computing. It is frequently the case that it is wise to start with a clean, layered design, with clear boundaries. Then, in certain special cases, those layer boundaries may be violated, where the performance and efficiency benefits outweigh the inelegance of the layer violation.

These layer violations are optional. They are done primarily for efficiency reasons, and generally should not be required for correct operation. A Discovery Proxy MAY operate solely at the mDNS layer, without any knowledge of semantics at the DNS-SD layer or above.

5.6. Answer Aggregation

In a simple analysis, simply gathering multicast answers and forwarding them in a unicast response seems adequate, but it raises the question of how long the Discovery Proxy should wait to be sure that it has received all the Multicast DNS answers it needs to form a complete Unicast DNS response. If it waits too little time, then it risks its Unicast DNS response being incomplete. If it waits too long, then it creates a poor user experience at the client end. In fact, there may be no time which is both short enough to produce a good user experience and at the same time long enough to reliably produce complete results.

Similarly, the Discovery Proxy -- the authoritative name server for the subdomain in question -- needs to decide what DNS TTL to report for these records. If the TTL is too long then the recursive (caching) name servers issuing queries on behalf of their clients risk caching stale data for too long. If the TTL is too short then the amount of network traffic will be more than necessary. In fact, there may be no TTL which is both short enough to avoid undesirable stale data and at the same time long enough to be efficient on the network.

Both these dilemmas are solved by use of DNS Long-Lived Queries (DNS LLQ) [LLQ] or its newer replacement, DNS Push Notifications [PUSH].

Clients supporting unicast DNS Service Discovery SHOULD implement DNS Push Notifications [PUSH] for improved user experience.

Clients and Discovery Proxies MAY support both DNS LLQ and DNS Push, and when talking to a Discovery Proxy that supports both, the client may use either protocol, as it chooses, though it is expected that only DNS Push will continue to be supported in the long run.

When a Discovery Proxy receives a query using DNS LLQ or DNS Push Notification, it responds immediately using the Multicast DNS records it already has in its cache (if any). This provides a good client user experience by providing a near-instantaneous response. Simultaneously, the Discovery Proxy issues a Multicast DNS query on the local link to discover if there are any additional Multicast DNS records it did not already know about. Should additional Multicast DNS responses be received, these are then delivered to the client using additional DNS LLQ or DNS Push Notification update messages. The timeliness of such update messages is limited only by the timeliness of the device responding to the Multicast DNS query. If the Multicast DNS device responds quickly, then the update message is delivered quickly. If the Multicast DNS device responds slowly, then

the update message is delivered slowly. The benefit of using update messages is that the Discovery Proxy can respond promptly because it doesn't have to delay its unicast response to allow for the expected worst-case delay for receiving all the Multicast DNS responses. Even if a proxy were to try to provide reliability by assuming an excessively pessimistic worst-case time (thereby giving a very poor user experience) there would still be the risk of a slow Multicast DNS device taking even longer than that (e.g., a device that is not even powered on until ten seconds after the initial query is received) resulting in incomplete responses. Using update message solves this dilemma: even very late responses are not lost; they are delivered in subsequent update messages.

There are two factors that determine specifically how responses are generated:

The first factor is whether the query from the client used LLQ or DNS Push Notification (typical with long-lived service browsing PTR queries) or not (typical with one-shot operations like SRV or address record queries). Note that queries using LLQ or DNS Push Notification are received directly from the client. Queries not using LLQ or DNS Push Notification are generally received via the client's configured recursive (caching) name server.

The second factor is whether the Discovery Proxy already has at least one record in its cache that positively answers the question.

- Not using LLQ or Push Notification; no answer in cache:
  Issue an mDNS query, exactly as a local client would issue an mDNS query on the local link for the desired record name, type and class, including retransmissions, as appropriate, according to the established mDNS retransmission schedule [RFC6762]. As soon as any Multicast DNS response packet is received that contains one or more positive answers to that question (with or without the Cache Flush bit [RFC6762] set), or a negative answer (signified via a Multicast DNS NSEC record [RFC6762]), the Discovery Proxy generates a Unicast DNS response packet containing the corresponding (filtered and translated) answers and sends it to the remote client. If after six seconds no Multicast DNS answers have been received, return a negative response to the remote client. Six seconds is enough time to transmit three mDNS queries, and allow some time for responses to arrive.
  DNS TTLs in responses are capped to at most ten seconds.

- Not using LLQ or Push Notification; at least one answer in cache:

Send response right away to minimise delay.
DNS TTLs in responses are capped to at most ten seconds.
No local mDNS queries are performed.
(Reasoning: Given RRSet TTL harmonisation, if the proxy has one Multicast DNS answer in its cache, it can reasonably assume that it has all of them.)

- Using LLQ or Push Notification; no answer in cache:
  As in the case above with no answer in the cache, perform mDNS querying for six seconds, and send a response to the remote client as soon as any relevant mDNS response is received.
  If after six seconds no relevant mDNS response has been received, return negative response to the remote client (for LLQ; not applicable for PUSH).
  (Reasoning: We don’t need to rush to send an empty answer.)
  Whether or not a relevant mDNS response is received within six seconds, the query remains active for as long as the client maintains the LLQ or PUSH state, and if mDNS answers are received later, LLQ or PUSH update messages are sent.
  DNS TTLs in responses are returned unmodified.

- Using LLQ or Push Notification; at least one answer in cache:
  As in the case above with at least one answer in cache, send response right away to minimise delay.
  The query remains active for as long as the client maintains the LLQ or PUSH state, and if additional mDNS answers are received later, LLQ or PUSH update messages are sent.
  (Reasoning: We want UI that is displayed very rapidly, yet continues to remain accurate even as the network environment changes.)
  DNS TTLs in responses are returned unmodified.

Note that the "negative responses" referred to above are "no error no answer" negative responses, not NXDOMAIN. This is because the Discovery Proxy cannot know all the Multicast DNS domain names that may exist on a link at any given time, so any name with no answers may have child names that do exist, making it an "empty nonterminal" name.

6. Administrative DNS Records

6.1. DNS SOA (Start of Authority) Record

The MNAME field SHOULD contain the host name of the Discovery Proxy device (i.e., the same domain name as the rdata of the NS record delegating the relevant zone(s) to this Discovery Proxy device).

The RNAME field SHOULD contain the mailbox of the person responsible for administering this Discovery Proxy device.

The SERIAL field MUST be zero.

Zone transfers are undefined for Discovery Proxy zones, and consequently the REFRESH, RETRY and EXPIRE fields have no useful meaning for Discovery Proxy zones. These fields SHOULD contain reasonable default values. The RECOMMENDED values are: REFRESH 7200, RETRY 3600, EXPIRE 86400.

The MINIMUM field (used to control the lifetime of negative cache entries) SHOULD contain the value 10. The value of ten seconds is chosen based on user-experience considerations (see Section 5.5.1).

In the event that there are multiple Discovery Proxy devices on a link for fault tolerance reasons, this will result in clients receiving inconsistent SOA records (different MNAME, and possibly RNAME) depending on which Discovery Proxy answers their SOA query. However, since clients generally have no reason to use the MNAME or RNAME data, this is unlikely to cause any problems.

6.2. DNS NS Records

In the event that there are multiple Discovery Proxy devices on a link for fault tolerance reasons, the parent zone MUST be configured with glue records giving the names and addresses of all the Discovery Proxy devices on the link.

Each Discovery Proxy device MUST be configured with its own NS record, and with the NS records of its fellow Discovery Proxy devices on the same link, so that it can return the correct answers for NS queries.

6.3. DNS SRV Records

In the event that a Discovery Proxy implements Long-Lived Queries [LLQ] and/or DNS Push Notifications [PUSH] (as most SHOULD) they MUST generate answers for the appropriate corresponding _dns-llq._udp.<zone> and/or _dns-push-tls._tcp.<zone> SRV record queries. These records are conceptually inserted into the namespace of the corresponding zones. They do not exist in the ".local" namespace of the local link.

7. DNSSEC Considerations

7.1. On-line signing only

The Discovery Proxy acts as the authoritative name server for designated subdomains, and if DNSSEC is to be used, the Discovery Proxy needs to possess a copy of the signing keys, in order to generate authoritative signed data from the local Multicast DNS responses it receives. Off-line signing not applicable to Discovery Proxy.

7.2. NSEC and NSEC3 Records

In DNSSEC, NSEC and NSEC3 records are used to assert the nonexistence of certain names, also described as "authenticated denial of existence".

Since a Discovery Proxy only knows what names exist on the local link by issuing queries for them, and since it would be impractical to issue queries for every possible name just to find out which names exist and which do not, a Discovery Proxy cannot programatically synthesize the traditional NSEC and NSEC3 records which assert the nonexistence of a large range names. Instead, when generating a negative response, a Discovery Proxy programatically synthesizes a single NSEC record assert the nonexistence of just the specific name queried, and no others. Since the Discovery Proxy has the zone signing key, it can do this on demand. Since the NSEC record asserts the nonexistence of only a single name, zone walking is not a concern, so NSEC3 is not necessary.

Note that this applies only to traditional immediate DNS queries, which may return immediate negative answers when no immediate positive answer is available. When used with a DNS Push Notification subscription [PUSH] there are no negative answers, merely the absence of answers so far, which may change in the future if answers become available.

8. IPv6 Considerations

An IPv6-only host and an IPv4-only host behave as "ships that pass in the night". Even if they are on the same Ethernet [IEEE-3], neither is aware of the other's traffic. For this reason, each link may have *two* unrelated ".local." zones, one for IPv6 and one for IPv4. Since for practical purposes, a group of IPv6-only hosts and a group of IPv4-only hosts on the same Ethernet act as if they were on two entirely separate Ethernet segments, it is unsurprising that their use of the ".local." zone should occur exactly as it would if they really were on two entirely separate Ethernet segments.

It will be desirable to have a mechanism to 'stitch' together these two unrelated ".local." zones so that they appear as one. Such mechanism will need to be able to differentiate between a dual-stack (v4/v6) host participating in both ".local." zones, and two different hosts, one IPv6-only and the other IPv4-only, which are both trying to use the same name(s). Such a mechanism will be specified in a future companion document.

At present, it is RECOMMENDED that a Discovery Proxy be configured with a single domain name for both the IPv4 and IPv6 ".local." zones on the local link, and when a unicast query is received, it should issue Multicast DNS queries using both IPv4 and IPv6 on the local link, and then combine the results.

9. Security Considerations

9.1. Authenticity

A service proves its presence on a link by its ability to answer link-local multicast queries on that link. If greater security is desired, then the Discovery Proxy mechanism should not be used, and something with stronger security should be used instead, such as authenticated secure DNS Update [RFC2136] [RFC3007].

9.2. Privacy

The Domain Name System is, generally speaking, a global public database. Records that exist in the Domain Name System name hierarchy can be queried by name from, in principle, anywhere in the world. If services on a mobile device (like a laptop computer) are made visible via the Discovery Proxy mechanism, then when those services become visible in a domain such as "My House.example.com" that might indicate to (potentially hostile) observers that the mobile device is in my house. When those services disappear from "My House.example.com" that change could be used by observers to infer when the mobile device (and possibly its owner) may have left the house. The privacy of this information may be protected using techniques like firewalls, split-view DNS, and Virtual Private Networks (VPNs), as are customarily used today to protect the privacy of corporate DNS information.

The Discovery Proxy could also provide sensitive records only to authenticated users. This is a general DNS problem, not specific to the Discovery Proxy. Work is underway in the IETF to tackle this problem [RFC7626].

9.3. Denial of Service

A remote attacker could use a rapid series of unique Unicast DNS queries to induce a Discovery Proxy to generate a rapid series of corresponding Multicast DNS queries on one or more of its local links. Multicast traffic is generally more expensive than unicast traffic -- especially on Wi-Fi links -- which makes this attack particularly serious. To limit the damage that can be caused by such attacks, a Discovery Proxy (or the underlying Multicast DNS subsystem which it utilizes) MUST implement Multicast DNS query rate limiting appropriate to the link technology in question. For today's 802.11b/g/n/ac Wi-Fi links (for which approximately 200 multicast packets per second is sufficient to consume approximately 100% of the wireless spectrum) a limit of 20 Multicast DNS query packets per second is RECOMMENDED. On other link technologies like Gigabit Ethernet higher limits may be appropriate. A consequence of this rate limiting is that a rogue remote client could issue an excessive number of queries, resuling in denial of service to other remote clients attempting to use that Discovery Proxy. However, this is preferable to a rogue remote client being able to inflict even greater harm on the local network, which could impact the correct operation of all local clients on that network.

10. Intelectual Property Rights

Apple has submitted an IPR disclosure concerning the technique proposed in this document. Details are available on the IETF IPR disclosure page [IPR2119].

11. IANA Considerations

This document has no IANA Considerations.

12. Acknowledgments

Thanks to Markus Stenberg for helping develop the policy regarding the four styles of unicast response according to what data is immediately available in the cache. Thanks to Anders Brandt, Tim Chown, Ralph Droms, Ray Hunter, Ted Lemon, Tom Pusateri, Markus Stenberg, Dave Thaler, and Andrew Yourtchenko for their comments.

13. References

13.1. Normative References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <http://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

[RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <http://www.rfc-editor.org/info/rfc1918>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2308] Andrews, M., "Negative Caching of DNS Queries (DNS NCACHE)", RFC 2308, DOI 10.17487/RFC2308, March 1998, <http://www.rfc-editor.org/info/rfc2308>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC3927] Cheshire, S., Aboba, B., and E. Guttman, "Dynamic Configuration of IPv4 Link-Local Addresses", RFC 3927, DOI 10.17487/RFC3927, May 2005, <http://www.rfc-editor.org/info/rfc3927>.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <http://www.rfc-editor.org/info/rfc4862>.

[RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, DOI 10.17487/RFC5198, March 2008, <http://www.rfc-editor.org/info/rfc5198>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, December 2012.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, December 2012.

[PUSH] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-09 (work in progress), October 2016.

13.2. Informative References

[HOME] Cheshire, S., "Special Use Top Level Domain 'home'", draft-cheshire-homenet-dot-home (work in progress), November 2015.

[IPR2119] "Apple Inc.'s Statement about IPR related to Hybrid Unicast/Multicast DNS-Based Service Discovery", <https://datatracker.ietf.org/ipr/2119/>.

[ohp] "Discovery Proxy (Hybrid Proxy) implementation for OpenWrt", <https://github.com/sbyx/ohybridproxy/>.

[LLQ] Sekar, K., "DNS Long-Lived Queries", draft-sekar-dns-llq-01 (work in progress), August 2006.

[RFC2132] Alexander, S. and R. Droms, "DHCP Options and BOOTP Vendor Extensions", RFC 2132, DOI 10.17487/RFC2132, March 1997, <http://www.rfc-editor.org/info/rfc2132>.

[RFC2136] Vixie, P., Ed., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, DOI 10.17487/RFC2136, April 1997, <http://www.rfc-editor.org/info/rfc2136>.

[RFC3007] Wellington, B., "Secure Domain Name System (DNS) Dynamic Update", RFC 3007, DOI 10.17487/RFC3007, November 2000, <http://www.rfc-editor.org/info/rfc3007>.

[RFC4193] Hinden, R. and B. Haberman, "Unique Local IPv6 Unicast Addresses", RFC 4193, DOI 10.17487/RFC4193, October 2005, <http://www.rfc-editor.org/info/rfc4193>.

[RFC7558] Lynn, K., Cheshire, S., Blanchet, M., and D. Migault, "Requirements for Scalable DNS-Based Service Discovery (DNS-SD) / Multicast DNS (mDNS) Extensions", RFC 7558, DOI 10.17487/RFC7558, July 2015, <http://www.rfc-editor.org/info/rfc7558>.

[RFC7626] Bortzmeyer, S., "DNS Privacy Considerations", RFC 7626, DOI 10.17487/RFC7626, August 2015, <http://www.rfc-editor.org/info/rfc7626>.

[RFC7788] Stenberg, M., Barth, S., and P. Pfister, "Home Networking Control Protocol", RFC 7788, DOI 10.17487/RFC7788, April 2016, <http://www.rfc-editor.org/info/rfc7788>.

[RFC6760] Cheshire, S. and M. Krochmal, "Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP)", RFC 6760, December 2012.

[ZC] Cheshire, S. and D. Steinberg, "Zero Configuration Networking: The Definitive Guide", O'Reilly Media, Inc. , ISBN 0-596-10100-7, December 2005.

[IEEE-1Q] "IEEE Standard for Local and metropolitan area networks -- Bridges and Bridged Networks", IEEE Std 802.1Q-2014, November 2014, <http://standards.ieee.org/getieee802/download/802-1Q-2014.pdf>.

[IEEE-3] "Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CMSA/CD) Access Method and Physical Layer Specifications", IEEE Std 802.3-2008, December 2008, <http://standards.ieee.org/getieee802/802.3.html>.

[IEEE-5] Institute of Electrical and Electronics Engineers, "Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 5: Token ring access method and physical layer specification", IEEE Std 802.5-1998, 1995.

[IEEE-11] "Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", IEEE Std 802.11-2007, June 2007, <http://standards.ieee.org/getieee802/802.11.html>.

Appendix A. Implementation Status

Some aspects of the mechanism specified in this document already exist in deployed software. Some aspects are new. This section outlines which aspects already exist and which are new.

A.1. Already Implemented and Deployed

Domain enumeration by the client (the "b._dns-sd._udp" queries) is already implemented and deployed.

Unicast queries to the indicated discovery domain is already implemented and deployed.

These are implemented and deployed in Mac OS X 10.4 and later (including all versions of Apple iOS, on all iPhone and iPads), in Bonjour for Windows, and in Android 4.1 "Jelly Bean" (API Level 16) and later.

Domain enumeration and unicast querying have been used for several years at IETF meetings to make Terminal Room printers discoverable from outside the Terminal room. When an IETF attendee presses Cmd-P on a Mac, or selects AirPrint on an iPad or iPhone, and the Terminal room printers appear, that is because the client is sending unicast DNS queries to the IETF DNS servers.

A.2. Already Implemented

A minimal portable Discovery Proxy implementation has been produced by Markus Stenberg and Steven Barth, which runs on OS X and several Linux variants including OpenWrt [ohp]. It was demonstrated at the Berlin IETF in July 2013.

Tom Pusateri also has an implementation that runs on any Unix/Linux. It has a RESTful interface for management and an experimental demo CLI and web interface.

A.3. Partially Implemented

The current APIs make multiple domains visible to client software, but most client UI today lumps all discovered services into a single flat list. This is largely a chicken-and-egg problem. Application writers were naturally reluctant to spend time writing domain-aware UI code when few customers today would benefit from it. If Discovery Proxy deployment becomes common, then application writers will have a reason to provide better UI. Existing applications will work with the Discovery Proxy, but will show all services in a single flat list. Applications with improved UI will group services by domain.

The Long-Lived Query mechanism [LLQ] referred to in this specification exists and is deployed, but has not been standardized by the IETF. The IETF is considering standardizing a superior Long-Lived Query mechanism called DNS Push Notifications [PUSH]. The pragmatic short-term deployment approach is for vendors to produce Discovery Proxies that implement both the deployed Long-Lived Query mechanism [LLQ] (for today's clients) and the new DNS Push Notifications mechanism [PUSH] as the preferred long-term direction.

The translating/filtering Discovery Proxy specified in this document. Implementations are under development, and operational experience with these implementations has guided updates to this document.

A.4. Not Yet Implemented

Client implementations of the new DNS Push Notifications mechanism [PUSH] are currently underway.

A mechanism to 'stitch' together multiple ".local." zones so that they appear as one. Such a stitching mechanism will be specified in a future companion document. This stitching mechanism addresses the issue that if a printer is physically moved from one link to another, then conceptually the old service has disappeared from the DNS namespace, and a new service with a similar name has appeared. This stitching mechanism will allow a service to change its point of attachment without changing the name by which it can be found.

Author's Address

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
USA

Phone: +1 408 974 3207
Email: cheshire@apple.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 8, 2017

C. Huitema
Private Octopus Inc.
D. Kaiser
University of Konstanz

March 7, 2017

Device Pairing Using Short Authentication Strings
draft-ietf-dnssd-pairing-01.txt

Abstract

This document proposes a device pairing mechanism that establishes a relationship between two devices by agreeing on a secret and manually verifying the secret's authenticity using an SAS (short authentication string). Pairing has to be performed only once per pair of devices, as for a re-discovery at any later point in time, the exchanged secret can be used for mutual authentication.

The proposed pairing method is suited for each application area where human operated devices need to establish a relation that allows configurationless and privacy preserving re-discovery at any later point in time. Since privacy preserving applications are the main suitors, we especially care about privacy.

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 September 8, 2017.

Table of Contents

1. Introduction 3
1.1. Requirements 3
1.2. Document Organization 4
2. Problem Statement and Requirements 4
2.1. Secure Pairing Over Internet Connections 4
2.2. Identity Assurance 5
2.3. Adequate User Interface 5
2.3.1. Short PIN Proved Inadequate 5
2.3.2. Push Buttons Just Work, But Are Insecure 6
2.3.3. Short Range Communication 6
2.3.4. Short Authentication Strings 7
2.4. Resist Cryptographic Attacks 8
2.5. Privacy Requirements 10
2.6. Using TLS 11
2.7. QR codes 12
2.8. Intra User Pairing and Transitive Pairing 13
3. Design of the Pairing Mechanism 14
3.1. Discovery 14
3.2. Agreement 15
3.3. Authentication 15
3.4. Public Authentication Keys 16
4. Solution 16
4.1. Discovery 16
4.2. Agreement and Authentication 16
5. Security Considerations 19
6. IANA Considerations 20
7. Acknowledgments 20
8. References 20
8.1. Normative References 20
8.2. Informative References 20
Authors' Addresses 22

1. Introduction

To engage in secure and privacy preserving communication, hosts need to differentiate between authorized peers, which must both know about the host's presence and be able to decrypt messages sent by the host, and other peers, which must not be able to decrypt the host's messages and ideally should not be aware of the host's presence. The necessary relationship between host and peer can be established by a centralized service, e.g. a certificate authority, by a web of trust, e.g. PGP, or -- without using global identities -- by device pairing.

This document proposes a device pairing mechanism that provides human operated devices with pairwise authenticated secrets, allowing mutual automatic re-discovery at any later point in time along with mutual private authentication. We especially care about privacy and user-friendliness.

The proposed pairing mechanism consists of three steps needed to establish a relationship between a host and a peer:

1. Discovering the peer device. The host needs a means to discover network parameters necessary to establish a connection to the peer. During this discovery process, neither the host nor the peer must disclose its presence.

2. Agreeing on pairing data. The devices have to agree on pairing data, which can be used by both parties at any later point in time to generate identifiers for re-discovery and to prove the authenticity of the pairing. The pairing data can e.g. be a shared secret agreed upon via a Diffie-Hellman key exchange.

3. Authenticating pairing data. Since in most cases the messages necessary to agree upon pairing data are send over an insecure channel, means that guarantee the authenticity of these messages are necessary; otherwise the pairing data is in turn not suited as a means for a later proof of authenticity. For the proposed pairing mechanism we use manual interaction involving an SAS (short authentication string) to proof the authenticity of the pairing data.

1.1. Requirements

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].

1.2. Document Organization

NOTE TO RFC EDITOR: remove or rewrite this section before publication.

This document is organized in two parts. The first part, composed of Section 1, Section 2, and Section 3 presents the pairing need, the list of requirements that shall be met, and the general design of the solution. This first part is informational in nature. The second part, composed of Section 4 and Section 5, is the actual specification of the protocol.

In his early review, Steve Kent observed that the style of the first part seems inappropriate for a standards track document, and suggested that the two parts should be split into two documents, the first part becoming an informational document, and the second focusing on standard track specification of the protocol, making reference to the informational document as appropriate. We, the authors, will seek working group approval before performing this split.

2. Problem Statement and Requirements

The general pairing requirement is easy to state: establish a trust relation between two entities in a secure manner. But details matter, and in this section we explore the detailed requirements that guide our design.

2.1. Secure Pairing Over Internet Connections

Many pairing protocols have already been developed, in particular for the pairing of devices over specific wireless networks. For example, the current Bluetooth specifications include a pairing protocol that has evolved over several revisions towards better security and usability [BTLEPairing]. The Wi-Fi Alliance defined the Wi-Fi Protected Setup process to ease the setup of security-enabled Wi-Fi networks in home and small office environments [WPS]. Other wireless standards have defined or are defining similar protocols, tailored to specific technologies.

This specification defines a pairing protocol that is independent of the underlying technology. We simply make the hypothesis that the two parties engaged in the pairing can discover each other and then establish connections over IP in order to agree on a shared secret.

[[TODO: Should we support certificates besides a shared secret?]]

2.2. Identity Assurance

The parties in the pairing must be able to identify each other. To put it simply, if Alice believes that she is establishing a pairing with Bob, she must somehow ensure that the pairing is actually established with Bob, and not with some interloper like Eve or Nessie. Providing this assurance requires designing both the protocol and the user interface (UI) with care.

Consider for example an attack in which Eve tricks Alice into engaging in a pairing process while pretending to be Bob. Alice must be able to discover that something is wrong, and refuse to establish the pairing. The parties engaged in the pairing must at least be able to verify their identities, respectively.

2.3. Adequate User Interface

Because the pairing protocol is executed without prior knowledge, it is typically vulnerable to "Man-in-the-middle" attacks. While Alice is trying to establish a pairing with Bob, Eve positions herself in the middle. Instead of getting a pairing between Alice and Bob, both Alice and Bob get paired with Eve. This requires specific features in the protocol to detect man-in-the-middle attacks, and if possible resist them. The reference [NR11] analyzes the various proposals to solve this problem, and in this document, we present a layman description of these issues in Section 2.4. The various protocols proposed in the literature impose diverse constraints on the UI interface, which we will review here.

2.3.1. Short PIN Proved Inadequate

The initial Bluetooth pairing protocol relied on a four digit PIN, displayed by one of the devices to be paired. The user would read that PIN and provide it to the other device. The PIN would then be used in a Password Authenticated Key Exchange. Wi-Fi Protected Setup [WPS] offered a similar option. There were various attacks against the actual protocol; some of the problems were caused by issues in the protocol, but most were tied to the usage of short PINs.

In the reference implementation, the PIN is picked at random by the paired device before the beginning of the exchange. But this requires that the paired device is capable of generating and displaying a four digit number. It turns out that many devices cannot do that. For example, an audio headset does not have any display capability. These limited devices ended up using static PINs, with fixed values like "0000" or "0001".

Even when the paired device could display a random PIN, that PIN will have to be copied by the user on the pairing device. It turns out that users do not like copying long series of numbers, and the usability thus dictated that the PINs be short -- four digits in practice. But there is only so much assurance as can be derived from a four digit key.

It is interesting to note that the latest revisions of the Bluetooth Pairing protocol [BTLEPairing] do not include the short PIN option anymore. The PIN entry methods have been superseded by the simple "just works" method for devices without displays, and by a procedure based on an SAS (short authentication string) when displays are available.

A further problem with these PIN based approaches is that -- in contrast to SASes -- the PIN is a secret instrumental in the security algorithm. To guarantee security, this PIN would have to be transmitted via a secure out of band channel.

2.3.2. Push Buttons Just Work, But Are Insecure

Some devices are unable to input or display any code. The industry more or less converged on a "push button" solution. When the button is pushed, devices enter a "pairing" mode, during which they will accept a pairing request from whatever other device connects to them.

The Bluetooth Pairing protocol [BTLEPairing] denotes that as the "just works" method. It does indeed work, and if the pairing succeeds the devices will later be able to use the pairing keys to authenticate connections. However, the procedure does not provide any protection against MITM attacks during the pairing process. The only protection is that pushing the button will only allow pairing for a limited time, thus limiting the opportunities of attacks.

As we set up to define a pairing protocol with a broad set of applications, we cannot limit ourselves to an insecure "push button" method. But we probably need to allow for a mode of operation that works for input-limited and display limited devices.

2.3.3. Short Range Communication

There have been several attempts to define pairing protocols that use "secure channels." Most of them are based on short range communication systems, where the short range limits the feasibility for attackers to access the channels. Example of such limited systems include for example:

- o QR codes, displayed on the screen of one device, and read by the camera of the other device.

- o Near Field Communication (NFC) systems, which provides wireless communication with a very short range.

- o Sound systems, in which one systems emits a sequence of sounds or ultrasounds that is picked by the microphone of the other system.

A common problem with these solutions is that they require special capabilities that may not be present in every device. Another problem is that they are often one-way channels. Yet another problem is that the side channel is not necessarily secret. QR codes could be read by third parties. Powerful radio antennas might be able to interfere with NFC. Sensitive microphones might pick the sounds. We will discuss the specific case of QR codes in Section 2.7.

2.3.4. Short Authentication Strings

The evolving pairing protocols seem to converge towards a "display and compare" method. This is in line with academic studies, such as [KFR09] or [USK11], and points to a very simple scenario:

1. Alice initiates pairing

2. Bob selects Alice’s device from a list.

3. Alice and Bob compare displayed strings that represent a fingerprint of the key.

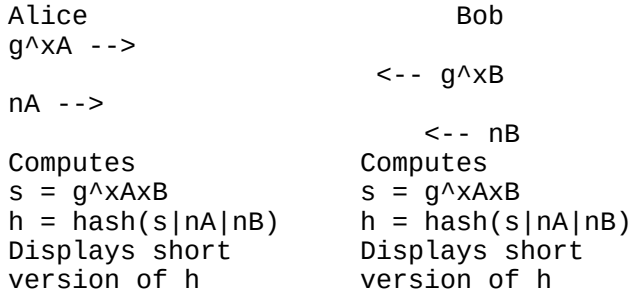
4. If the strings match, Alice and Bob accept the pairing.

Most existing pairing protocols display the fingerprint of the key as a 6 or 7 digit numbers. Usability studies show that this method gives good results, with little risk that users mistakenly accept two different numbers as matching. However, the authors of [USK11] found that people had more success comparing computer generated sentences than comparing numbers. This is in line with the argument in [XKCD936] to use sequences of randomly chosen common words as passwords. On the other hand, standardizing strings is more complicated than standardizing numbers. We would need to specify a list of common words, and the process to go from a binary fingerprint to a set of words. We would need to be concerned with internationalization issues, such as using different lists of words in German and in English. This could require the negotiation of word lists or languages inside the pairing protocols.

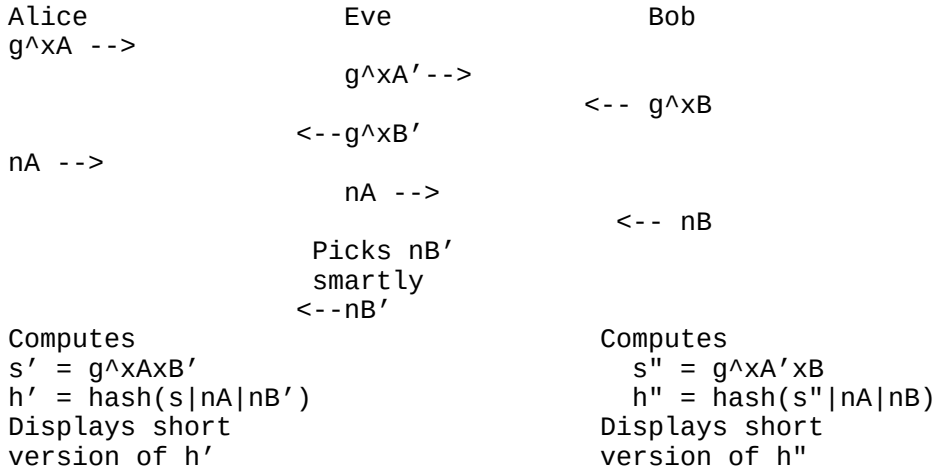
In contrast, numbers are easy to specify, as in "take a 20 bit number and display it as an integer using decimal notation".

2.4. Resist Cryptographic Attacks

It is tempting to believe that once two peers are connected, they could create a secret with a few simple steps, such as for example (1) exchange two nonces, (2) hash the concatenation of these nonces with the shared secret that is about to be established, (3) display a short authentication string composed of a short version of that hash on each device, and (4) verify that the two values match. This naive approach might yield the following sequence of messages:



If the two short hashes match, Alice and Bob are supposedly assured that they have computed the same secret, but there is a problem. The exchange may not deter a smart attacker in the middle. Let’s redraw the same message flow, this time involving Eve:



Let’s now assume that, in order to pick the nonce nB’ smartly, Eve runs the following algorithm:

```
s' = g^xAxB'
s" = g^xA'xB
repeat
   pick a new version of nB'
   h' = hash(s|nA|nB')
   h" = hash(s"|nA|nB)
until the short version of h'
matches the short version of h"
```

Of course, running this algorithm will, in theory, require as many iterations as there are possible values of the short hash. But hash algorithms are fast, and it is possible to try millions of values in less than a second. If the short string is made up of fewer than 6 digits, Eve will find a matching nonce quickly, and Alice and Bob will hardly notice the delay. Even if the matching string is as long as 8 letters, Eve will probably find a value where the short versions of h' and h" are close enough, e.g. start and end with the same two or three letters. Alice and Bob may well be fooled.

The classic solution to such problems is to "commit" a possible attacker to a nonce before sending it. This commitment can be realized by a hash. In the modified exchange, Alice sends a secure hash of her nonce before sending the actual value:

```
Alice                        Bob
g^xA -->
                          <-- g^xB

Computes                 Computes
s = g^xAxB               s = g^xAxB
h_a = hash(s|nA) -->
                             <-- nB
nA -->
                         verifies h_a == hash(s|nA)
Computes                 Computes
h = hash(s|nA|nB)        h = hash(s|nA|nB)
Displays short           Displays short
version of h             version of h
```

Alice will only disclose nA after having confirmation from Bob that hash(nA) has been received. At that point, Eve has a problem. She can still forge the values of the nonces but she needs to pick the nonce nA' before the actual value of nA has been disclosed. Eve would still have a random chance of fooling Alice and Bob, but it will be a very small chance: one in a million if the short authentication string is made of 6 digits, even fewer if that string is longer.

Nguyen et al. [NR11] survey these protocols and compare them with respect to the amount of necessary user interaction and the computation time needed on the devices. The authors state that such a protocol is optimal with respect to user interaction if it suffices for users to verify a single b-bit SAS while having a one-shot attack success probability of 2^-b. Further, n consecutive attacks on the protocol must not have a better success probability then n one-shot attacks.

There is still a theoretical problem, if Eve has somehow managed to "crack" the hash function. We build some "defense in depth" by some simple measures. In the design presented above, the hash "h_a" depends on the shared secret "s", which acts as a "salt" and reduces the effectiveness of potential attacks based on pre-computed catalogs. For simplicity, the design used a simple concatenation mechanism, but we could instead use a keyed-hash message authentication code (HMAC [RFC2104], [RFC6151]), using the shared secret as a key, since the HMAC construct has proven very robust over time. Then, we can constrain the size of the random numbers to be exactly the same as the output of the hash function. Hash attacks often require padding the input string with arbitrary data; restraining the size limits the likelyhood of such padding.

2.5. Privacy Requirements

Pairing exposes a relation between several devices and their owners. Adversaries may attempt to collect this information, for example in an attempt to track devices, their owners, or their "social graph". It is often argued that pairing could be performed in a safe place, from which adversaries are assumed absent, but experience shows that such assumptions are often misguided. It is much safer to acknowledge the privacy issues and design the pairing process accordingly.

In order to start the pairing process, devices must first discover each other. We do not have the option of using the private discovery protocol [I-D.ietf-dnssd-privacy] since the privacy of that protocol depends on a pre-existing pairing. In the simplest design, one of the devices will announce a "friendly name" using DNS-SD. Adversaries could monitor the discovery protocol, and record that name. An alternative would be for one device to announce a random name, and communicate it to the other device via some private channel. There is an obvious tradeoff here: friendly names are easier to use but less private than random names. We anticipate that different users will choose different tradeoffs, for example using friendly names if they assume that the environment is "safe," and using random names in public places.

During the pairing process, the two devices establish a connection and validate a pairing secret. As discussed in Section 2.3, we have to assume that adversaries can mount MITM attacks. The pairing protocol can detect such attacks and resist them, but the attackers will have access to all messages exchanged before validation is performed. It is important to not exchange any privacy sensitive information before that validation. This includes, for example, the identities of the parties or their public keys.

2.6. Using TLS

The pairing algorithms typically combine the establishment of a shared secret through an [EC]DH exchange with the verification of that secret through displaying and comparison of a "short authentication string" (SAS). As explained in Section 2.4, the secure comparison requires a "commit before disclose" mechanism.

We have three possible designs: (1) create a pairing algorithm from scratch, specifying our own crypto exchanges; (2) use an [EC]DH version of TLS to negotiate a shared secret, export the key to the application as specified in [RFC5705], and implement the "commit before disclose" and SAS verification as part of the pairing application; or, (3) use TLS, integrate the "commit before disclose" and SAS verification as TLS extensions, and export the verified key to the application as specified in [RFC5705].

When faced with the same choice, the designers of ZRTP [RFC6189] chose to design a new protocol integrated in the general framework of real time communications. We don’t want to follow that path, and would rather not create yet another protocol. We would need to reinvent a lot of the negotiation capabilities that are part of TLS, not to mention algorithm agility, post quantum, and all that sort of things. It is thus pretty clear that we should use TLS.

It turns out that there was already an attempt to define SAS extensions for TLS ([I-D.miers-tls-sas]). It is a very close match to our third design option, full integration of SAS in TLS, but the draft has expired, and there does not seem to be any support for the SAS options in the common TLS packages.

In our design, we will choose the middle ground option -- use TLS for [EC]DH, and implement the SAS verification as part of the pairing application. This minimizes dependencies on TLS packages to the availability of a key export API following [RFC5705]. We will need to specify the hash algorithm used for the SAS computation and validation, which carries some of the issues associated with "designing our own crypto". One solution would be to use the same hash algorithm negotiated by the TLS connection, but common TLS

packages do not always make this algorithm identifier available through standard APIs. A fallback solution is to specify a state of the art keyed MAC algorithm.

2.7. QR codes

In Section 2.3.3, we reviewed a number of short range communication systems that can be used to facilitate pairing. Out of these, QR codes stand aside because most devices that can display a short string can also display the image of a QR code, and because many pairing scenarios involve cell phones equipped with cameras capable of reading a QR code.

QR codes are displayed as images. An adversary equipped with powerful cameras could read the QR code just as well as the pairing parties. If the pairing protocol design embedded passwords or pins in the QR code, adversaries could access these data and compromise the protocol. On the other hand, there are ways to use QR codes even without assuming secrecy.

QR codes could be used at two of the three stages of pairing: Discovering the peer device, and authenticating the shared secret. Using QR codes provide advantages in both phases:

- Typical network based discovery involves interaction with two devices. The device to be discovered is placed in "server" mode, and waits for requests from the network. The device performing the discovery retrieves a list of candidates from the network. When there is more than one such candidate, the device user is expected to select the desired target from a list. In QR code mode, the discovered device will display a QR code, which the user will scan using the second device. The QR code will embed the device's name, its IP address, and the port number of the pairing service. The connection will be automatic, without relying on the network discovery. This is arguably less error-prone and safer than selecting from a network provided list.

- SAS based agreement involves displaying a short string on each device's display, and asking the user to verify that both devices display the same string. In QR code mode, one device could display a QR code containing this short string. The other device could scan it and compare it to the locally computed version. Because the procedure is automated, there is no dependency on the user diligence at comparing the short strings.

Offering QR codes as an alternative to discovery and agreement is straightforward. If QR codes are used, the pairing program on the server side might display something like:

Please connect to "Bob's phone 359"
or scan the following QR code:

If Alice's device is capable of reading the QR code, it will just scan it, establishes a connection, and run the pairing protocol. After the protocol messages have been exchanged, Bob's device will display a new QR code, encoding the hash code that should be matched. The UI might look like this:

Please scan the following QR code,
or verify that your device displays
the number: 388125

Did the number match (Yes/No)?

With the use of QR code, the pairing is established with little reliance on user judgment, which is arguably safer.

2.8. Intra User Pairing and Transitive Pairing

There are two usage modes for pairing: inter-users, and intra-user. Users have multiple devices. The simplest design is to not distinguish between pairing devices belonging to two users, e.g.,

Alice’s phone and Bob’s phone, and devices belonging to the same user, e.g., Alice’s phone and her laptop. This will most certainly work, but it raises the problem of transitivity. If Bob needs to interact with Alice, should he install just one pairing for "Alice and Bob", or should he install four pairings between Alice phone and laptop and Bob phone and laptop? Also, what happens if Alice gets a new phone?

One tempting response is to devise a synchronization mechanism that will let devices belonging to the same user share their pairings with other users. But it is fairly obvious that such service will have to be designed cautiously. The pairing system relies on shared secrets. It is much easier to understand how to manage secrets shared between exactly two parties than secrets shared with an unspecified set of devices.

Transitive pairing raises similar issues. Suppose that a group of users wants to collaborate. Will they need to set up a fully connected graph of pairings using the simple peer-to-peer mechanism, or could they use some transitive set, so that if Alice is connected with Bob and Bob with Carol, Alice automatically gets connected with Carol? Such transitive mechanisms could be designed, e.g. using a variation of Needham-Scroeder symmetric key protocol [NS1978], but it will require some extensive work. Groups can of course use simpler solution, e.g., build some star topology.

Given the time required, intra-user pairing synchronization mechanisms and transitive pairing mechanisms are left for further study.

3. Design of the Pairing Mechanism

In this section we discuss the design of pairing protocols that use manually verified short authentication strings (SAS), considering both security and user experience.

We divide pairing in three parts: discovery, agreement, and authentication, detailed in the following subsections.

3.1. Discovery

The goal of the discovery phase is establishing a connection, which is later used to exchange the pairing data, between the two devices that are about to be paired in an IP network without any prior knowledge and without publishing any private information. In accordance with TLS, we refer to the device initiating the cryptographic protocol as client, and to the other device as server; the server has to be discoverable by the client.

Granting privacy during the discovery phase without relying on prior knowledge demands another user interaction (besides the SAS verification during the authentication phase). There are two possible ways of realizing this user interaction depending on whether QR codes are supported or not. If QR codes are supported, the discovery process can be independent of DNS-SD, because QR codes allow the transmission of a sufficient amount of data. Leveraging QR codes, the discovery proceeds as follows.

1. The server displays a QR code containing the instance name, the IPv4 or IPv6 address, and the port number of the service/

2. The client scans the QR code retrieving the necessary information for establishing a connection to the server.

If QR codes are not supported, the discovery proceeds as follows.

1. The server displays its chosen instance name on its screen.

2. The client performs a discovery of all the "pairing" servers available on the local network. This may result in the discovery of several servers.

3. Among these available "pairing servers" the client’s user selects the name that matches the name displayed by the server.

4. Per DNS-SD, the client then retrieves the SRV records of the selected instance, select one of the document servers, retrieves its A or AAAA records, and establishes the connection.

3.2. Agreement

Once the server has been selected, the client connects to it without further user intervention. Client and server use this connection for exchanging data that allows them to agree on a shared secret by using a cryptographic protocol that yields an SAS. We discussed design aspects of such protocols in Section 2.4.

3.3. Authentication

In the authentication phase, the users are asked to validate the pairing by comparing the SASes -- typically represented by a number encoded over up to 7 decimal digits. If the SASes match, each user enters an agreement, for example by pressing a button labeled "OK", which results in the pairing being remembered. If they do not match, each user should cancel the pairing, for example by pressing a button labeled "CANCEL".

Depending on whether QR codes are supported, the SAS may also be represented as QR code. Despite the fact that using QR codes to represent the authentication string renders using longer authentication strings feasible, we suggest to always generate an SAS during the agreement phase, because this makes realizations of the agreement phase and the authentication phase independent. Devices may display the "real" name of the other device alongside the SAS.

3.4. Public Authentication Keys

[[TODO: Should we discuss public authentication keys whose fingerprints are verified during pairing?]]

4. Solution

In the proposed pairing protocol, one of the devices acts as a "server", and the other acts as a "client". The server will publish a "pairing service". The client will discover the service instance during the discovery phase, as explained in Section 4.1. The pairing service itself is specified in Section 4.2.

4.1. Discovery

The discovery uses DNS-SD [RFC6763] over mDNS [RFC6762]. The pairing service is identified in DNS SD as "_pairing._tcp". When the pairing service starts, the server starts publishing the chosen instance name. The client will discover that name and the corresponding connection parameters.

If QR code scanning is available as OOB channel, the discovery data is directly transmitted via QR codes instead of DNS-SD over mDNS. The QR data contains connection data otherwise found in the SRV and A or AAAA records: IPv4 or IPv6 address, port number, and optionally host name.

[[TODO: We should precisely specify the data layout of this QR code. It could either be the wire format of the corresponding resource records (which would be easier for us), or a more efficient representation. If we chose the wire format, we could use a fix name as instance name.]]

4.2. Agreement and Authentication

The pairing protocol is built using TLS. The following description uses the presentation language defined in section 4 of [RFC5246]. The protocol uses five message types, defined in the following enum:

```
enum {
   ClientHash(1),
   ServerRandom(2),
   ClientRandom(3),
   ServerSuccess(4),
   ClientSuccess(5)
} PairingMessageType;
```

Devices implementing the service MUST support TLS 1.2 [RFC5246], and MAY negotiate TLS 1.3 when it becomes available. When using TLS, the client and server MUST negotiate a ciphersuite providing forward secrecy (PFS), and strong encryption (256 bits symmetric key). All implementations using TLS 1.2 MUST be able to negotiate the cipher suite TLS_DH_anon_WITH_AES_256_CBC_SHA256.

Once the TLS connection has been established, each party extracts the pairing secret S_p from the connection context per [RFC5705], using the following parameters:

Disambiguating label string: "PAIRING SECRET"

Context value: empty.

Length value: 32 bytes (256 bits).

Once S_p has been obtained, the client picks a random number R_c, exactly 32 bytes long. The client then selects a hash algorithm, which SHOULD be the same algorithm as negotiated for building the PRF in the TLS connection. If there is no suitable API to retrieve that algorithm, the client MAY use SHA256 instead. The client then computes the hash value H_c as:

```
H_c = HMAC_hash(S_p, R_c)
```

Where "HMAC_hash" is the HMAC function constructed with the selected algorithm.

The client transmits the selected hash function and the computed value of H_c in the Client Hash message, over the TLS connection:

```
struct {
   PairingMessageType messageType;
   hashAlgorithm hash;
   uint8 hashLength;
   opaque H_c[hashLength];
} ClientHashMessage;
```

messageType Set to "ClientHash".

hash The code of the selected hash algorithm, per definition of HashAlgorithm in section 7.4.1.1.1 of [RFC5246].

hashLength The length of the hash H_c, which MUST be consistent with the selected algorithm "hash".

H_c The value of the client hash.

Upon reception of this message, the server stores its value. The server picks a random number R_s, exactly 32 bytes long, and transmits it to the client in the server random message, over the TLS connection:

```
struct {
   PairingMessageType messageType;
   opaque R_s[32];
} ServerRandomMessage;
```

messageType Set to "ServerRandom".

R_s The value of the random number chosen by the server.

Upon reception of this message, the client discloses its own random number by transmitting the client random message:

```
struct {
   PairingMessageType messageType;
   opaque R_c[32];
} ClientRandomMessage;
```

messageType Set to "ClientRandom".

R_c The value of the random number chosen by the client.

Upon reception of this message, the server verifies that the number R_c hashes to the previously received value H_c. If the number does not match, the server MUST abandon the pairing attempt and abort the TLS connection.

At this stage, both client and server can compute the short hash SAS as:

SAS = first 20 bits of HMAC_hash(S_p, R_c + R_s)

Where "HMAC_hash" is the HMAC function constructed with the hash algorithm selected by the client in the ClientHashMessage.

Both client and server display the SAS as a decimal integer, and ask the user to compare the values. If the server supports QR codes, the server displays a QR code encoding the decimal string representation of the SAS. If the client is capable of scanning QR codes, it may scan the value and compare it to the locally computed value.

If the values do not match, the user cancels the pairing. Otherwise, the protocol continues with the exchange of names, both server and client announcing their own preferred name in a Success message

```
struct {
   PairingMessageType messageType;
   uint8 nameLength;
   opaque name[nameLength];
} ClientSuccessMessage;
```

messageType Set to "ClientSuccess" if transmitted by the client, "ServerSuccess" if by the server.

nameLength The length of the string encoding the selected name.

name The selected name of the client or the server, encoded as a string of UTF8 characters.

After receiving these messages, client and servers can orderly close the TLS connection, terminating the pairing exchange.

5. Security Considerations

We need to consider two types of attacks against a pairing system: attacks that occur during the establishment of the pairing relation, and attacks that occur after that establishment.

During the establishment of the pairing system, we are concerned with privacy attacks and with MITM attacks. Privacy attacks reveal the existence of a pairing between two devices, which can be used to track graphs of relations. MITM attacks result in compromised pairing keys. The discovery procedures specified in Section 4.1 and the authentication procedures specified in Section 4.2 are specifically designed to mitigate such attacks, assuming that the client and user are in close, physical proximity and thus a human user can visually acquire and verify the pairing information.

The establishment of the pairing results in the creation of a shared secret. After the establishment of the pairing relation, attackers who compromise one of the devices could access the shared secret. This will enable them to either track or spoof the devices. To mitigate such attacks, nodes MUST store the secret safely, and MUST

be able to quickly revoke a compromised pairing. This is however not sufficient, as the compromise of the pairing key could remain undetected for a long time. For further safety, nodes SHOULD assign a time limit to the validity of pairings, discard the corresponding keys when the time has passed, and establish new pairings.

6. IANA Considerations

This draft does not require any IANA action.

7. Acknowledgments

We would like to thank Steve Kent for a detailed early review of this document.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <http://www.rfc-editor.org/info/rfc5705>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <http://www.rfc-editor.org/info/rfc6762>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <http://www.rfc-editor.org/info/rfc6763>.

8.2. Informative References

[BTLEPairing] Bluetooth SIG, "Bluetooth Low Energy Security Overview", 2016, <https://developer.bluetooth.org/TechnologyOverview/Pages/LE-Security.aspx>.

[I-D.ietf-dnssd-privacy]
Huitema, C. and D. Kaiser, "Privacy Extensions for DNS-SD", draft-ietf-dnssd-privacy-00 (work in progress), October 2016.

[I-D.miers-tls-sas]
Miers, I., Green, M., and E. Rescorla, "Short Authentication Strings for TLS", draft-miers-tls-sas-00 (work in progress), February 2014.

[KFR09] Kainda, R., Flechais, I., and A. Roscoe, "Usability and Security of Out-Of-Band Channels in Secure Device Pairing Protocols", DOI: 10.1145/1572532.1572547, SOUPS 09, Proceedings of the 5th Symposium on Usable Privacy and Security, Mountain View, CA, January 2009.

[NR11] Nguyen, L. and A. Roscoe, "Authentication protocols based on low-bandwidth unspoofable channels: a comparative survey", DOI: 10.3233/JCS-2010-0403, Journal of Computer Security, Volume 19 Issue 1, Pages 139-201, January 2011.

[NS1978] Needham, R. and M. Schroeder, ". Using encryption for authentication in large networks of computers", Communications of the ACM 21 (12): 993-999, DOI: 10.1145/359657.359659, December 1978.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <http://www.rfc-editor.org/info/rfc6151>.

[RFC6189] Zimmermann, P., Johnston, A., Ed., and J. Callas, "ZRTP: Media Path Key Agreement for Unicast Secure RTP", RFC 6189, DOI 10.17487/RFC6189, April 2011, <http://www.rfc-editor.org/info/rfc6189>.

[USK11] Uzun, E., Saxena, N., and A. Kumar, "Pairing devices for social interactions: a comparative usability evaluation", DOI: 10.1145/1978942.1979282, Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 2011.

[WPS] Wi-Fi Alliance, "Wi-Fi Protected Setup", 2016, <http://www.wi-fi.org/discover-wi-fi/wi-fi-protected-setup>.

[XKCD936] Munroe, R., "XKCD: Password Strength", 2011, <https://www.xkcd.com/936/>.

Authors' Addresses

Christian Huitema
Private Octopus Inc.
Friday Harbor, WA 98250
U.S.A.

Email: huitema@huitema.net

Daniel Kaiser
University of Konstanz
Konstanz 78457
Germany

Email: daniel.kaiser@uni-konstanz.de

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 11, 2017

C. Huitema
Private Octopus Inc.
D. Kaiser
University of Konstanz

March 10, 2017

Privacy Extensions for DNS-SD
draft-ietf-dnssd-privacy-01.txt

Abstract

DNS-SD (DNS Service Discovery) normally discloses information about both the devices offering services and the devices requesting services. This information includes host names, network parameters, and possibly a further description of the corresponding service instance. Especially when mobile devices engage in DNS Service Discovery over Multicast DNS at a public hotspot, a serious privacy problem arises.

We propose to solve this problem by a two-stage approach. In the first stage, hosts discover Private Discovery Service Instances via DNS-SD using special formats to protect their privacy. These service instances correspond to Private Discovery Servers running on peers. In the second stage, hosts directly query these Private Discovery Servers via DNS-SD over TLS. A pairwise shared secret necessary to establish these connections is only known to hosts authorized by a pairing system.

Status of This Memo

Table of Contents

1. Introduction 3
1.1. Requirements 4
2. Privacy Implications of DNS-SD 4
2.1. Privacy Implication of Publishing Service Instance Names 4
2.2. Privacy Implication of Publishing Node Names 5
2.3. Privacy Implication of Publishing Service Attributes . . 5
2.4. Device Fingerprinting 6
2.5. Privacy Implication of Discovering Services 6
3. Design of the Private DNS-SD Discovery Service 7
3.1. Device Pairing 8
3.2. Discovery of the Private Discovery Service 8
3.2.1. Obfuscated Instance Names 8
3.2.2. Using a Predictable Nonce 9
3.2.3. Using a Short Proof 10
3.2.4. Direct Queries 11
3.3. Private Discovery Service 11
3.3.1. A Note on Private DNS Services 12
3.4. Randomized Host Names 13
3.5. Timing of Obfuscation and Randomization 13
4. Private Discovery Service Specification 14
4.1. Host Name Randomization 14
4.2. Device Pairing 14
4.3. Private Discovery Server 15
4.3.1. Establishing TLS Connections 15
4.4. Publishing Private Discovery Service Instances 15
4.5. Discovering Private Discovery Service Instances 16
4.6. Direct Discovery of Private Discovery Service Instances . 17
4.7. Using the Private Discovery Service 17
5. Security Considerations 17
5.1. Attacks Against the Pairing System 18
5.2. Denial of Discovery of the Private Discovery Service . . 18

5.3. Replay Attacks Against Discovery of the Private Discovery Service 18
5.4. Denial of Private Discovery Service 19
5.5. Replay Attacks against the Private Discovery Service . . 19
6. IANA Considerations 20
7. Acknowledgments 20
8. References 20
8.1. Normative References 20
8.2. Informative References 21
Authors' Addresses 22

1. Introduction

DNS-SD [RFC6763] over mDNS [RFC6762] enables configurationless service discovery in local networks. It is very convenient for users, but it requires the public exposure of the offering and requesting identities along with information about the offered and requested services. Some of the information published by the announcements can be very revealing. These privacy issues and potential solutions are discussed in [KW14a] and [KW14b].

There are cases when nodes connected to a network want to provide or consume services without exposing their identity to the other parties connected to the same network. Consider for example a traveler wanting to upload pictures from a phone to a laptop when connected to the Wi-Fi network of an Internet cafe, or two travelers who want to share files between their laptops when waiting for their plane in an airport lounge.

We expect that these exchanges will start with a discovery procedure using DNS-SD [RFC6763] over mDNS [RFC6762]. One of the devices will publish the availability of a service, such as a picture library or a file store in our examples. The user of the other device will discover this service, and then connect to it.

When analyzing these scenarios in Section 2, we find that the DNS-SD messages leak identifying information such as the instance name, the host name or service properties. We review the design constraint of a solution in Section 3, and describe the proposed solution in Section 4.

While we focus on a mDNS-based distribution of the DNS-SD resource records, our solution is agnostic about the distribution method and also works with other distribution methods, e.g. the classical hierarchical DNS.

1.1. Requirements

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].

2. Privacy Implications of DNS-SD

DNS-Based Service Discovery (DNS-SD) is defined in [RFC6763]. It allows nodes to publish the availability of an instance of a service by inserting specific records in the DNS ([RFC1033], [RFC1034], [RFC1035]) or by publishing these records locally using multicast DNS (mDNS) [RFC6762]. Available services are described using three types of records:

PTR Record: Associates a service type in the domain with an "instance" name of this service type.

SRV Record: Provides the node name, port number, priority and weight associated with the service instance, in conformance with [RFC2782].

TXT Record: Provides a set of attribute-value pairs describing specific properties of the service instance.

In the remaining subsections, we will review the privacy issues related to publishing instance names, node names, service attributes and other data, as well as review the implications of using the discovery service as a client.

2.1. Privacy Implication of Publishing Service Instance Names

In the first phase of discovery, the client obtains all the PTR records associated with a service type in a given naming domain. Each PTR record contains a Service Instance Name defined in Section 4 of [RFC6763]:

Service Instance Name = <Instance> . <Service> . <Domain>

The <Instance> portion of the Service Instance Name is meant to convey enough information for users of discovery clients to easily select the desired service instance. Nodes that use DNS-SD over mDNS [RFC6762] in a mobile environment will rely on the specificity of the instance name to identify the desired service instance. In our example of users wanting to upload pictures to a laptop in an Internet Cafe, the list of available service instances may look like:

```
Alice's Images          . _imageStore._tcp . local
Alice's Mobile Phone    . _presence._tcp   . local
Alice's Notebook        . _presence._tcp   . local
Bob's Notebook          . _presence._tcp   . local
Carol's Notebook        . _presence._tcp   . local
```

Alice will see the list on her phone and understand intuitively that she should pick the first item. The discovery will "just work".

However, DNS-SD/mDNS will reveal to anybody that Alice is currently visiting the Internet Cafe. It further discloses the fact that she uses two devices, shares an image store, and uses a chat application supporting the _presence protocol on both of her devices. She might currently chat with Bob or Carol, as they are also using a _presence supporting chat application. This information is not just available to devices actively browsing for and offering services, but to anybody passively listing to the network traffic.

2.2. Privacy Implication of Publishing Node Names

The SRV records contain the DNS name of the node publishing the service. Typical implementations construct this DNS name by concatenating the "host name" of the node with the name of the local domain. The privacy implications of this practice are reviewed in [RFC8117]. Depending on naming practices, the host name is either a strong identifier of the device, or at a minimum a partial identifier. It enables tracking of the device, and by extension of the device's owner.

2.3. Privacy Implication of Publishing Service Attributes

The TXT record's attribute and value pairs contain information on the characteristics of the corresponding service instance. This in turn reveals information about the devices that publish services. The amount of information varies widely with the particular service and its implementation:

- Some attributes like the paper size available in a printer, are the same on many devices, and thus only provide limited information to a tracker.

- Attributes that have freeform values, such as the name of a directory, may reveal much more information.

Combinations of attributes have more information power than specific attributes, and can potentially be used for "fingerprinting" a specific device.

Information contained in TXT records does not only breach privacy by making devices trackable, but might directly contain private information about the user. For instance the _presence service reveals the "chat status" to everyone in the same network. Users might not be aware of that.

Further, TXT records often contain version information about services allowing potential attackers to identify devices running exploit-prone versions of a certain service.

2.4. Device Fingerprinting

The combination of information published in DNS-SD has the potential to provide a "fingerprint" of a specific device. Such information includes:

- The list of services published by the device, which can be retrieved because the SRV records will point to the same host name.
- The specific attributes describing these services.
- The port numbers used by the services.
- The values of the priority and weight attributes in the SRV records.

This combination of services and attributes will often be sufficient to identify the version of the software running on a device. If a device publishes many services with rich sets of attributes, the combination may be sufficient to identify the specific device.

There is however an argument that devices providing services can be discovered by observing the local traffic, and that trying to hide the presence of the service is futile. The same argument can be extended to say that the pattern of services offered by a device allows for fingerprinting the device. This may or may not be true, since we can expect that services will be designed or updated to avoid leaking fingerprints. In any case, the design of the discovery service should avoid making a bad situation worse, and should as much as possible avoid providing new fingerprinting information.

2.5. Privacy Implication of Discovering Services

The consumers of services engage in discovery, and in doing so reveal some information such as the list of services they are interested in and the domains in which they are looking for the services. When the clients select specific instances of services, they reveal their

preference for these instances. This can be benign if the service type is very common, but it could be more problematic for sensitive services, such as for example some private messaging services.

One way to protect clients would be to somehow encrypt the requested service types. Of course, just as we noted in Section 2.4, traffic analysis can often reveal the service.

3. Design of the Private DNS-SD Discovery Service

In this section, we present the design of a two-stage solution that enables private use of DNS-SD, without affecting existing users. The solution is largely based on the architecture proposed in [KW14b], which separates the general private discovery problem in three components. The first component is an offline pairing mechanism, which is performed only once per pair of users. It establishes a shared secret over an authenticated channel, allowing devices to authenticate using this secret without user interaction at any later point in time. We use the pairing system proposed in [I-D.ietf-dnssd-pairing].

The further two components are online (in contrast to pairing they are performed anew each time joining a network) and compose the two service discovery stages, namely

- Discovery of the Private Discovery Service -- the first stage -- in which hosts discover the Private Discovery Service (PDS), a special service offered by every host supporting our extension. After the discovery, hosts connect to the PSD offered by paired peers.

- Actual Service Discovery -- the second stage -- is performed through the Private Discovery Service, which only accepts encrypted messages associated with an authenticated session; thus not compromising privacy.

In other words, the hosts first discover paired peers and then directly engage in privacy preserving service discovery.

The stages are independent with respect to means used for transmitting the necessary data. While in our extension the messages for the first stage are transmitted using IP multicast, the messages for the second stage are transmitted via unicast. One could also imagine using a Distributed Hash Table for the first stage, being completely independent of multicast.

3.1. Device Pairing

Any private discovery solution needs to differentiate between authorized devices, which are allowed to get information about discoverable entities, and other devices, which should not be aware of the availability of private entities. The commonly used solution to this problem is establishing a "device pairing".

Device pairing has to be performed only once per pair of users. This is important for user-friendliness, as it is the only step that demands user-interaction. After this single pairing, privacy preserving service discovery works fully automatically. In this document, we leverage [I-D.ietf-dnssd-pairing] as pairing mechanism.

The pairing yields a mutually authenticated shared secret, and optionally mutually authenticated public keys or certificates added to a local web of trust. Public key technology has many advantages, but shared secrets are typically easier to handle on small devices.

3.2. Discovery of the Private Discovery Service

The first stage of service discovery is to check whether instances of compatible Private Discovery Services are available in the local scope. The goal of that stage is to identify devices that share a pairing with the querier, and are available locally. The service instances can be discovered using regular DNS-SD procedures, but the list of discovered services will have to be filtered so only paired devices are retained.

3.2.1. Obfuscated Instance Names

The instance names for the Private Discovery Service are obfuscated, so that authorized peers can associate the instance with its publisher, but unauthorized peers can only observe what looks like a random name. To achieve this, the names are composed as the concatenation of a nonce and a proof, which is composed by hashing the nonce with a pairing key:

```
PrivateInstanceName = <nonce>|<proof>
proof = hash(<nonce>|<key>)
```

The publisher will publish as many instances as it has established pairings.

The discovering party that looks for instances of the service will receive lists of advertisements from nodes present on the network. For each advertisement, it will parse the instance name, and then, for each available pairing key, compares the proof to the hash of the

nonce concatenated with this pairing key. If there is no match, it discards the instance name. If there is a match, it has discovered a peer.

3.2.2. Using a Predictable Nonce

Assume that there are N nodes on the local scope, and that each node has on average M pairings. Each node will publish on average M records, and the node engaging in discovery may have to process on average N*M instance names. The discovering node will have to compute on average M potential hashes for each nonce. The number of hash computations would scale as O(N*M*M), which means that it could cause a significant drain of resource in large networks.

In order to minimize the amount of computing resource, we suggest that the nonce be derived from the current time, for example set to a representation of the current time rounded to some period. With this convention, receivers can predict the nonces that will appear in the published instances. They will only need to compute O(M) hashes, instead of O(N*M*M).

The publishers will have to create new records at the end of each rounding period. If the rounding period is set too short, they will have to repeat that very often, which is inefficient. On the other hand, if the rounding period is too long, the system may be exposed to replay attacks. We propose to set a value of about 5 minutes, which seems to be a reasonable compromise.

Unix defines a 32 bit time stamp as the number of seconds elapsed since January 1st, 1970 not counting leap seconds. The most significant 24 bits of this 32 bit number represent the number of 256 seconds intervals since the epoch. 256 seconds correspond to 4 minutes and 16 seconds, which is close enough to our design goal of 5 minutes. We will thus use this 24 bit number as nonce, represented as 3 octets.

Publishers will need to compute O(M) hashes at most once per time stamp interval. If records can be created "on the fly", publishers will only need to perform that computation upon receipt of the first query during a given interval, and cache the computed results for the remainder of the interval. There are however scenarios in which records have to be produced in advance, for example when records are published within a scope defined by a domain name and managed by a "classic" DNS server. In such scenarios, publishers will need to perform the computations and publication exactly once per time stamp interval.

3.2.3. Using a Short Proof

Devices will have to publish as many instance names as they have peers. The instance names will have to be represented via a text string, which means that the binary concatenation of nonce and proof will have to be encoded using a binary-to-text conversion such as BASE64 ([RFC2045] section 6.8) or BASE32 ([RFC4648] section 6).

Using long proofs, such as the full output of SHA256 [RFC4055], would generate fairly long instance names: 48 characters using BASE64, or 56 using BASE56. These long names would inflate the network traffic required when discovering the privacy service. They would also limit the number of DNS-SD PTR records that could be packed in a single 1500 octet sized packet, to 23 or fewer with BASE64, or 20 or fewer with BASE32.

Shorter proofs lead to shorter messages, which is more efficient as long as we do not encounter too many collisions. A collision will happen if the proof computed by the publisher using one key matches a proof computed by a receiver using another key. If a receiver mistakenly believes that a proof fits one of its peers, it will attempt to connect to the service as explained in section Section 4.5 but in the absence of the proper pairwise shared key, the connection will fail. This will not create an actual error, but the probability of such events should be kept low.

The following table provides the probability that a discovery agent maintaining 100 pairings will observe a collision after receiving 100000 advertisement records. It also provides the number of characters required for the encoding of the corresponding instance name in BASE64 or BASE32, assuming 24 bit nonces.

Proof	Collisions	BASE64	BASE32
24	5.96046%	8	16
32	0.02328%	11	16
40	0.00009%	12	16
48	3.6E-09	12	16
56	1.4E-11	15	16

Table 1

The table shows that for a proof, 24 bits would be too short. 32 bits might be long enough, but the BASE64 encoding requires padding if the input is not an even multiple of 24 bits, and BASE32 requires padding if the input is not a multiple of 40 bits. Given that, the desirable

proof lengths are thus 48 bits if using BASE64, or 56 bits if using BASE32. The resulting instance name will be either 12 characters long with BASE64, allowing 54 advertisements in an 1500 byte mDNS message, or 16 characters long with BASE32, allowing 47 advertisements per message.

In the specification section, we will assume BASE64, and 48 bit proofs composed of the first 6 bytes of a SHA256 hash.

3.2.4. Direct Queries

The preceding sections assume that the discovery is performed using the classic DNS-SD process, in which a query for all available "instance names" of a service provides a list of PTR records. The discoverer will then select the instance names that correspond to its peers, and request the SRV and TXT records corresponding to the service instance, and then obtain the relevant A or AAAA records. This is generally required in DNS-SD because the instance names are not known in advance, but for the Private Discovery Service the instance names can be predicted, and a more efficient Direct Query method can be used.

At a given time, the node engaged in discovery can predict the nonce that its peer will use, since that nonce is composed by rounding the current time. The node can also compute the proofs that its peers might use, since it knows the nonce and the keys. The node can thus build a list of instance names, and directly query the SRV records corresponding to these names. If peers are present, they will answer directly.

This "direct query" process will result in fewer network messages than the regular DNS-SD query process in some circumstances, depending on the number of peers per node and the number of nodes publishing the presence discovery service in the desired scope.

When using mDNS, it is possible to pack multiple queries in a single broadcast message. Using name compression and 12 characters per instance name, it is possible to pack 70 queries in a 1500 octet mDNS multicast message. It is also possible to request unicast replies to the queries, resulting in significant efficiency gains in wireless networks.

3.3. Private Discovery Service

The Private Discovery Service discovery allows discovering a list of available paired devices, and verifying that either party knows the corresponding shared secret. At that point, the querier can engage in a series of directed discoveries.

We have considered defining an ad-hoc protocol for the private discovery service, but found that just using TLS would be much simpler. The Directed Private Discovery service is just a regular DNS-SD service, accessed over TLS, using the encapsulation of DNS over TLS defined in [RFC7858]. The main difference with simple DNS over TLS is the need for authentication.

We assume that the pairing process has provided each pair of authorized client and server with a shared secret. We can use that shared secret to provide mutual authentication of clients and servers using "Pre Shared Key" authentication, as defined in [RFC4279] and incorporated in the latest version of TLS [I-D.ietf-tls-tls13].

One difficulty is the reliance on a key identifier in the protocol. For example, in TLS 1.3 the PSK extension is defined as:

```
opaque psk_identity<0..2^16-1>;

struct {
    select (Role) {
        case client:
            psk_identity identities<2..2^16-1>;

        case server:
            uint16 selected_identity;
    }
} PreSharedKeyExtension
```

According to the protocol, the PSK identity is passed in clear text at the beginning of the key exchange. This is logical, since server and clients need to identify the secret that will be used to protect the connection. But if we used a static identifier for the key, adversaries could use that identifier to track server and clients. The solution is to use a time-varying identifier, constructed exactly like the "proof" described in Section 3.2, by concatenating a nonce and the hash of the nonce with the shared secret.

3.3.1. A Note on Private DNS Services

Our solution uses a variant of the DNS over TLS protocol [RFC7858] defined by the DNS Private Exchange working group (DPRIVE). DPRIVE is also working on an UDP variant, DNS over DTLS [I-D.ietf-dprive-dnsodtls], which would also be a candidate.

DPRIVE and Private Discovery solve however two somewhat different problems. DPRIVE is concerned with the confidentiality of DNS transactions, addressing the problems outlined in [RFC7626]. However, DPRIVE does not address the confidentiality or privacy

issues with publication of services, and is not a direct solution to DNS-SD privacy:

- Discovery queries are scoped by the domain name within which services are published. As nodes move and visit arbitrary networks, there is no guarantee that the domain services for these networks will be accessible using DNS over TLS or DNS over DTLS.

- Information placed in the DNS is considered public. Even if the server does support DNS over TLS, third parties will still be able to discover the content of PTR, SRV and TXT records.

- Neither DNS over TLS nor DNS over DTLS applies to MDNS.

In contrast, we propose using mutual authentication of the client and server as part of the TLS solution, to ensure that only authorized parties learn the presence of a service.

3.4. Randomized Host Names

Instead of publishing their actual name in the SRV records, nodes could publish a randomized name. That is the solution argued for in [RFC8117].

Randomized host names will prevent some of the tracking. Host names are typically not visible by the users, and randomizing host names will probably not cause much usability issues.

3.5. Timing of Obfuscation and Randomization

It is important that the obfuscation of instance names is performed at the right time, and that the obfuscated names change in synchrony with other identifiers, such as MAC Addresses, IP Addresses or host names. If the randomized host name changed but the instance name remained constant, an adversary would have no difficulty linking the old and new host names. Similarly, if IP or MAC addresses changed but host names remained constant, the adversary could link the new addresses to the old ones using the published name.

The problem is handled in [RFC8117], which recommends to pick a new random host name at the time of connecting to a new network. New instance names for the Private Discovery Services should be composed at the same time.

4. Private Discovery Service Specification

The proposed solution uses the following components:

- Host name randomization to prevent tracking.
- Device pairing yielding pairwise shared secrets.
- A Private Discovery Server (PDS) running on each host.
- Discovery of the PDS instances using DNS-SD.

These components are detailed in the following subsections.

4.1. Host Name Randomization

Nodes publishing services with DNS-SD and concerned about their privacy MUST use a randomized host name. The randomized name MUST be changed when network connectivity changes, to avoid the correlation issues described in Section 3.5. The randomized host name MUST be used in the SRV records describing the service instance, and the corresponding A or AAAA records MUST be made available through DNS or MDNS, within the same scope as the PTR, SRV and TXT records used by DNS-SD.

If the link-layer address of the network connection is properly obfuscated (e.g. using MAC Address Randomization), the Randomized Host Name MAY be computed using the algorithm described in section 3.7 of [RFC7844]. If this is not possible, the randomized host name SHOULD be constructed by simply picking a 48 bit random number meeting the Randomness Requirements for Security expressed in [RFC4075], and then use the hexadecimal representation of this number as the obfuscated host name.

4.2. Device Pairing

Nodes that want to leverage the Private Directory Service for private service discovery among peers MUST share a secret with each of these peers. Each shared secret MUST be a 256 bit randomly chosen number. We RECOMMEND using the pairing mechanism proposed in [I-D.ietf-dnssd-pairing] to establish these secrets.

[[TODO: Should we support mutually authenticated certificates? They can also be used to initiate TLS and have several advantages, i.e. allow setting an expiry date.]]

4.3. Private Discovery Server

A Private Discovery Server (PDS) is a minimal DNS server running on each host. Its task is to offer resource records corresponding to private services only to authorized peers. These peers MUST share a secret with the host (see Section 4.2). To ensure privacy of the requests, the service is only available over TLS [RFC5246], and the shared secrets are used to mutually authenticate peers and servers.

The Private Name Server SHOULD support DNS push notifications [I-D.ietf-dnssd-push], e.g. to facilitate an up-to-date contact list in a chat application without polling.

4.3.1. Establishing TLS Connections

The PDS MUST only answer queries via DNS over TLS [RFC7858] and MUST use a PSK authenticated TLS handshake [RFC4279]. The client and server SHOULD negotiate a forward secure cipher suite such as DHE-PSK or ECDHE-PSK when available. The shared secret exchanged during pairing MUST be used as PSK. To guarantee interoperability, implementations of the Private Name Server MUST support TLS_PSK_WITH_AES_256_GCM_SHA384.

When using the PSK based authentication, the "psk_identity" parameter identifying the pre-shared key MUST be identical to the "Instance Identifier" defined in Section 4.4, i.e. 24 bit nonce and 48 bit proof encoded in BASE64 as 12 character string. The server will use the pairing key associated with this instance identifier.

4.4. Publishing Private Discovery Service Instances

Nodes that provide the Private Discovery Service SHOULD advertise their availability by publishing instances of the service through DNS-SD.

The DNS-SD service type for the Private Discovery Service is "_pds._tcp".

Each published instance describes one server and one pairing. In the case where a node manages more than one pairing, it should publish as many instances as necessary to advertise all available pairings.

Each instance name is composed as follows:

```
pick a 24 bit nonce, set to the 24 most
significant bits of the 32 bit Unix GMT time.

compute a 48 bit proof:
   proof = first 48 bits of HASH(<nonce>|<pairing key>)

set the 72 bit binary identifier as the concatenation
of nonce and proof

set instance-ID = BASE64(binary identifier)
```

In this formula, HASH SHOULD be the function SHA256 defined in [RFC4055], and BASE64 is defined in section 6.8 of [RFC2045]. The concatenation of a 24 bit nonce and 48 bit proof result in a 72 bit string. The BASE64 conversion is 12 characters long per [RFC6763].

4.5. Discovering Private Discovery Service Instances

Nodes that wish to discover Private Discovery Service Instances SHOULD issue a DNS-SD discovery request for the service type "_pds._tcp". They MAY, as an alternative, use the Direct Discovery procedure defined in Section 4.6. If nodes send a DNS-SD discovery request, they will receive in response a series of PTR records, providing the names of the instances present in the scope.

The querier SHOULD examine each instance to see whether it corresponds to one of its available pairings, according to the following conceptual algorithm:

```
for each received instance name:
   convert the instance name to binary using BASE64
   if the conversion fails,
      discard the instance.
   if the binary instance length is not multiple 72 bits,
      discard the instance.

   nonce = first 24 bits of binary.

   if nonce does not match the first 24 bits of the current
   time plus or minus 1 minute, discard the instance.

   for each available pairing
      retrieve the key Xj of pairing number j
      compute F = first 48 bits of hash(nonce, Xj)
      if F is equal to the last 48 bits of
      the binary instance ID
         mark the pairing number j as available
```

The check of the current time is meant to mitigate replay attacks, while not mandating a time synchronization precision better than one minute.

Once a pairing has been marked available, the querier SHOULD try connecting to the corresponding instance, using the selected key. The connection is likely to succeed, but it MAY fail for a variety of reasons. One of these reasons is the probabilistic nature of the hint, which entails a small chance of "false positive" match. This will occur if the hash of the nonce with two different keys produces the same result. In that case, the TLS connection will fail with an authentication error or a decryption error.

4.6. Direct Discovery of Private Discovery Service Instances

Nodes that wish to discover Private Discovery Service Instances MAY use the following Direct Discovery procedure instead of the regular DNS-SD Discovery explained in Section 4.5.

To perform Direct Discovery, nodes should compose a list of Private Discovery Service Instances Names. There will be one name for each pairing available to the node. The Instance ID for each name will be composed of a nonce and a proof, using the algorithm specified in Section 4.4.

The querier will issue SRV record queries for each of these names. The queries will only succeed if the corresponding instance is present, in which case a pairing is discovered. After that, the querier SHOULD try connecting to the corresponding instance, as explained in Section 4.4.

4.7. Using the Private Discovery Service

Once instances of the Private Discovery Service have been discovered, peers can establish TLS connections and send DNS requests over these connections, as specified in DNS-SD.

5. Security Considerations

This document specifies a method to protect the privacy of service publishing nodes. This is especially useful when operating in a public space. Hiding the identity of the publishing nodes prevents some forms of "targeting" of high value nodes. However, adversaries can attempt various attacks to break the anonymity of the service, or to deny it. A list of these attacks and their mitigations are described in the following sections.

5.1. Attacks Against the Pairing System

There are a variety of attacks against pairing systems, which may result in compromised pairing secrets. If an adversary manages to acquire a compromised key, the adversary will be able to perform private service discovery according to Section 4.5. This will allow tracking of the service. The adversary will also be able to discover which private services are available for the compromised pairing.

Attacks on pairing systems are detailed in [I-D.ietf-dnssd-pairing].

5.2. Denial of Discovery of the Private Discovery Service

The algorithm described in Section 4.5 scales as O(M*N), where M is the number of pairings per node and N is the number of nodes in the local scope. Adversaries can attack this service by publishing "fake" instances, effectively increasing the number N in that scaling equation.

Similar attacks can be mounted against DNS-SD: creating fake instances will generally increase the noise in the system and make discovery less usable. Private Discovery Service discovery SHOULD use the same mitigations as DNS-SD.

The attack could be amplified if the clients needed to compute proofs for all the nonces presented in Private Discovery Service Instance names. This is mitigated by the specification of nonces as rounded time stamps in Section 4.5. If we assume that timestamps must not be too old, there will be a finite number of valid rounded timestamps at any time. Even if there are many instances present, they would all pick their nonces from this small number of rounded timestamps, and a smart client will make sure that proofs are only computed once per valid time stamp.

5.3. Replay Attacks Against Discovery of the Private Discovery Service

Adversaries can record the service instance names published by Private Discovery Service instances, and replay them later in different contexts. Peers engaging in discovery can be misled into believing that a paired server is present. They will attempt to connect to the absent peer, and in doing so will disclose their presence in a monitored scope.

The binary instance identifiers defined in Section 4.4 start with 24 bits encoding the most significant bits of the "UNIX" time. In order to protect against replay attacks, clients SHOULD verify that this time is reasonably recent, as specified in Section 4.5.

[[TODO: Should we somehow encode the scope in the identifier? Having both scope and time would really mitigate that attack. For example, one could add a local IPv4 or IPv6 prefix in the nonce. However, this won't work in networks behind NAT. It would also increase the size of the instance ID.]]

5.4. Denial of Private Discovery Service

The Private Discovery Service is only available through a mutually authenticated TLS connection, which provides state-of-the-art protection mechanisms. However, adversaries can mount a denial of service attack against the service. In the absence of shared secrets, the connections will fail, but the servers will expend some CPU cycles defending against them.

To mitigate such attacks, nodes SHOULD restrict the range of network addresses from which they accept connections, matching the expected scope of the service.

This mitigation will not prevent denial of service attacks performed by locally connected adversaries; but protecting against local denial of service attacks is generally very difficult. For example, local attackers can also attack mDNS and DNS-SD by generating a large number of multicast requests.

5.5. Replay Attacks against the Private Discovery Service

Adversaries may record the PSK Key Identifiers used in successful connections to a private discovery service. They could attempt to replay them later against nodes advertising the private service at other times or at other locations. If the PSK Identifier is still valid, the server will accept the TLS connection, and in doing so will reveal being the same server observed at a previous time or location.

The PSK identifiers defined in Section 4.3.1 start with the 24 most significant bits of the "UNIX" time. In order to mitigate replay attacks, servers SHOULD verify that this time is reasonably recent, and fail the connection if it is too old, or if it occurs too far in the future.

The processing of timestamps is however affected by the accuracy of computer clocks. If the check is too strict, reasonable connections could fail. To further mitigate replay attacks, servers MAY record the list of valid PSK identifiers received in a recent past, and fail connections if one of these identifiers is replayed.

6. IANA Considerations

This draft does not require any IANA action. (Or does it? What about the _pds tag?)

7. Acknowledgments

This draft results from initial discussions with Dave Thaler, and encouragements from the DNS-SD working group members.

8. References

8.1. Normative References

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <http://www.rfc-editor.org/info/rfc2045>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4055] Schaad, J., Kaliski, B., and R. Housley, "Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 4055, DOI 10.17487/RFC4055, June 2005, <http://www.rfc-editor.org/info/rfc4055>.

[RFC4075] Kalusivalingam, V., "Simple Network Time Protocol (SNTP) Configuration Option for DHCPv6", RFC 4075, DOI 10.17487/RFC4075, May 2005, <http://www.rfc-editor.org/info/rfc4075>.

[RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <http://www.rfc-editor.org/info/rfc4279>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <http://www.rfc-editor.org/info/rfc6763>.

8.2. Informative References

[I-D.ietf-dnssd-pairing] Huitema, C. and D. Kaiser, "Device Pairing Using Short Authentication Strings", draft-ietf-dnssd-pairing-01 (work in progress), March 2017.

[I-D.ietf-dnssd-push] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-09 (work in progress), October 2016.

[I-D.ietf-dprive-dnsodtls] Reddy, T., Wing, D., and P. Patil, "Specification for DNS over Datagram Transport Layer Security (DTLS)", draft-ietf-dprive-dnsodtls-15 (work in progress), December 2016.

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

[KW14a] Kaiser, D. and M. Waldvogel, "Adding Privacy to Multicast DNS Service Discovery", DOI 10.1109/TrustCom.2014.107, 2014, <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7011331>.

[KW14b] Kaiser, D. and M. Waldvogel, "Efficient Privacy Preserving Multicast DNS Service Discovery", DOI 10.1109/HPCC.2014.141, 2014, <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7056899>.

[RFC1033] Lottor, M., "Domain Administrators Operations Guide", RFC 1033, DOI 10.17487/RFC1033, November 1987, <http://www.rfc-editor.org/info/rfc1033>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <http://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <http://www.rfc-editor.org/info/rfc2782>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <http://www.rfc-editor.org/info/rfc6762>.

[RFC7626] Bortzmeyer, S., "DNS Privacy Considerations", RFC 7626, DOI 10.17487/RFC7626, August 2015, <http://www.rfc-editor.org/info/rfc7626>.

[RFC7844] Huitema, C., Mrugalski, T., and S. Krishnan, "Anonymity Profiles for DHCP Clients", RFC 7844, DOI 10.17487/RFC7844, May 2016, <http://www.rfc-editor.org/info/rfc7844>.

[RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <http://www.rfc-editor.org/info/rfc7858>.

[RFC8117] Huitema, C., Thaler, D., and R. Winter, "Current Hostname Practice Considered Harmful", RFC 8117, DOI 10.17487/RFC8117, March 2017, <http://www.rfc-editor.org/info/rfc8117>.

Authors' Addresses

Christian Huitema
Private Octopus Inc.
Friday Harbor, WA 98250
U.S.A.

Email: huitema@huitema.net
URI: http://privateoctopus.com/

Daniel Kaiser
University of Konstanz
Konstanz 78457
Germany

Email: daniel.kaiser@uni-konstanz.de

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

T. Pusateri
Seeking affiliation
S. Cheshire
Apple Inc.

March 13, 2017

DNS Push Notifications
draft-ietf-dnssd-push-10

Abstract

The Domain Name System (DNS) was designed to return matching records efficiently for queries for data that is relatively static. When those records change frequently, DNS is still efficient at returning the updated results when polled. But there exists no mechanism for a client to be asynchronously notified when these changes occur. This document defines a mechanism for a client to be notified of such changes to DNS records, called DNS Push Notifications.

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 September 14, 2017.

Copyright Notice

Table of Contents

1. Introduction 3
1.1. Requirements Language 3
2. Motivation 4
3. Overview 5
4. Transport 7
5. State Considerations 8
6. Protocol Operation 9
6.1. Discovery 10
6.2. DNS Push Notification SUBSCRIBE 12
6.2.1. SUBSCRIBE Request 13
6.2.2. SUBSCRIBE Response 15
6.3. DNS Push Notification Updates 18
6.3.1. PUSH Message 19
6.3.2. PUSH Response 21
6.4. DNS Push Notification UNSUBSCRIBE 22
6.4.1. UNSUBSCRIBE Request 23
6.4.2. UNSUBSCRIBE Response 24
6.5. DNS Push Notification RECONFIRM 26
6.5.1. RECONFIRM Request 26
6.5.2. RECONFIRM Response 28
6.6. Client-Initiated Termination 30
7. Security Considerations 31
7.1. Security Services 31
7.2. TLS Name Authentication 31
7.3. TLS Compression 32
7.4. TLS Session Resumption 32
8. IANA Considerations 32
9. Acknowledgements 32
10. References 33
10.1. Normative References 33
10.2. Informative References 34
Authors' Addresses 36

1. Introduction

DNS records may be updated using DNS Update [RFC2136]. Other mechanisms such as a Discovery Proxy [DisProx] can also generate changes to a DNS zone. This document specifies a protocol for DNS clients to subscribe to receive asynchronous notifications of changes to RRSets of interest. It is immediately relevant in the case of DNS Service Discovery [RFC6763] but is not limited to that use case, and provides a general DNS mechanism for DNS record change notifications. Familiarity with the DNS protocol and DNS packet formats is assumed [RFC1034] [RFC1035] [RFC6895].

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in "Key words for use in RFCs to Indicate Requirement Levels" [RFC2119].

2. Motivation

As the domain name system continues to adapt to new uses and changes in deployment, polling has the potential to burden DNS servers at many levels throughout the network. Other network protocols have successfully deployed a publish/subscribe model to state changes following the Observer design pattern [obs]. XMPP Publish-Subscribe [XEP0060] and Atom [RFC4287] are examples. While DNS servers are generally highly tuned and capable of a high rate of query/response traffic, adding a publish/subscribe model for tracking changes to DNS records can result in more timely notification of changes with reduced CPU usage and lower network traffic.

Multicast DNS [RFC6762] implementations always listen on a well known link-local IP multicast group, and new services and updates are sent for all group members to receive. Therefore, Multicast DNS already has asynchronous change notification capability. However, when DNS Service Discovery [RFC6763] is used across a wide area network using Unicast DNS (possibly facilitated via a Discovery Proxy [DisProx]) it would be beneficial to have an equivalent capability for Unicast DNS, to allow clients to learn about DNS record changes in a timely manner without polling.

The DNS Long-Lived Queries (LLQ) [I-D.sekar-dns-llq] mechanism is an existing deployed solution to provide asynchronous change notifications, used by Apple’s Back to My Mac Service [RFC6281]. Back to My Mac was designed in an era when the data centre operations staff asserted that it was impossible for a server to handle large numbers of mostly-idle TCP connections, so LLQ had to defined as a UDP-based protocol, effectively replicating much of TCP’s connection state management logic in user space, and creating its own poor imitations of existing TCP features like the three-way handshake, flow control, and reliability.

This document builds on experience gained with the LLQ protocol, with an improved design. Instead of using UDP, this specification uses TCP, and therefore doesn’t need to reinvent existing TCP functionality. Using TCP also gives long-lived low-traffic connections better longevity through NAT gateways without resorting to excessive keepalive traffic [SessSig]. Instead of inventing a new vocabulary of messages to communicate DNS zone changes as LLQ did, this specification adopts the syntax and semantics of DNS Update messages [RFC2136].

3. Overview

The existing DNS Update protocol [RFC2136] provides a mechanism for clients to add or delete individual resource records (RRs) or entire resource record sets (RRSets) on the zone's server.

This specification adopts a simplified subset of these existing syntax and semantics, and uses them for DNS Push Notification messages going in the opposite direction, from server to client, to communicate changes to a zone. The client subscribes for Push Notifications by connecting to the server and sending DNS message(s) indicating the RRSet(s) of interest. When the client loses interest in updates to these records, it unsubscribes.

The DNS Push Notification server for a zone is any server capable of generating the correct change notifications for a name. It may be a master, slave, or stealth name server [RFC1996]. Consequently, the "_dns-push-tls._tcp.<zone>" SRV record for a zone MAY reference the same target host and port as that zone's "_dns-update-tls._tcp.<zone>" SRV record. When the same target host and port is offered for both DNS Updates and DNS Push Notifications, a client MAY use a single TCP connection to that server for both DNS Updates and DNS Push Notification Queries.

Supporting DNS Updates and DNS Push Notifications on the same server is OPTIONAL. A DNS Push Notification server does NOT also have to support DNS Update.

DNS Updates and DNS Push Notifications may be handled on different ports on the same target host, in which case they are not considered to be the "same server" for the purposes of this specification, and communications with these two ports are handled independently.

Standard DNS Queries MAY be sent over a DNS Push Notification connection, provided that these are queries for names falling within the server's zone (the <zone> in the "_dns-push-tls._tcp.<zone>" SRV record). The RD (Recursion Desired) bit MUST be zero.

DNS Push Notification clients are NOT required to implement DNS Update Prerequisite processing. Prerequisites are used to perform tentative atomic test-and-set type operations when a client updates records on a server, and that concept has no applicability when it comes to an authoritative server informing a client of changes to DNS records.

This DNS Push Notification specification includes support for DNS classes, for completeness. However, in practice, it is anticipated that for the foreseeable future the only DNS class in use will be DNS

class "IN", as is the reality today with existing DNS servers and clients. A DNS Push Notification server MAY choose to implement only DNS class "IN".

DNS Push Notifications impose less load on the responding server than rapid polling would, but Push Notifications do still have a cost, so DNS Push Notification clients MUST NOT recklessly create an excessive number of Push Notification subscriptions. A subscription SHOULD only be active when there is a valid reason to need live data (for example, an on-screen display is currently showing the results to the user) and the subscription SHOULD be cancelled as soon as the need for that data ends (for example, when the user dismisses that display). Implementations MAY want to implement idle timeouts, so that if the user ceases interacting with the device, the display showing the result of the DNS Push Notification subscription is automatically dismissed after a certain period of inactivity. For example, if a user presses the "Print" button on their smartphone, and then leaves the phone showing the printer discovery screen until the phone goes to sleep, then the printer discovery screen should be automatically dismissed as the device goes to sleep. If the user does still intend to print, this will require them to press the "Print" button again when they wake their phone up.

A DNS Push Notification client MUST NOT routinely keep a DNS Push Notification subscription active 24 hours a day, 7 days a week, just to keep a list in memory up to date so that if the user does choose to bring up an on-screen display of that data, it can be displayed really fast. DNS Push Notifications are designed to be fast enough that there is no need to pre-load a "warm" list in memory just in case it might be needed later.

Generally, as described in the DNS Session Signaling specification [SessSig], a client MUST NOT keep a connection to a server open indefinitely if it has no subscriptions (or other operations) active on that connection. A client MAY close a connection as soon as it becomes idle, and then if needed in the future, open a new connection when required. Alternatively, a client MAY speculatively keep an idle connection open for some time, subject to the constraint that it MUST NOT keep a connection open that has been idle for more than the session’s idle timeout (15 seconds by default).

4. Transport

Implementations of DNS Update [RFC2136] MAY use either User Datagram Protocol (UDP) [RFC0768] or Transmission Control Protocol (TCP) [RFC0793] as the transport protocol, in keeping with the historical precedent that DNS queries must first be sent over UDP [RFC1123]. This requirement to use UDP has subsequently been relaxed [RFC7766].

In keeping with the more recent precedent, DNS Push Notification is defined only for TCP. DNS Push Notification clients MUST use TLS over TCP.

Connection setup over TCP ensures return reachability and alleviates concerns of state overload at the server through anonymous subscriptions. All subscribers are guaranteed to be reachable by the server by virtue of the TCP three-way handshake. Flooding attacks are possible with any protocol, and a benefit of TCP is that there are already established industry best practices to guard against SYN flooding and similar attacks [IPJ.9-4-TCPSYN] [RFC4953].

Use of TCP also allows DNS Push Notifications to take advantage of current and future developments in TCP, such as Multipath TCP (MPTCP) [RFC6824], TCP Fast Open (TFO) [RFC7413], Tail Loss Probe (TLP) [I-D.dukkipati-tcpm-tcp-loss-probe], and so on.

Transport Layer Security (TLS) [RFC5246] is well understood and deployed across many protocols running over TCP. It is designed to prevent eavesdropping, tampering, or message forgery. TLS is REQUIRED for every connection between a client subscriber and server in this protocol specification. Additional security measures such as client authentication during TLS negotiation MAY also be employed to increase the trust relationship between client and server.

Additional authentication of the SRV target using DNSSEC verification and DANE TLSA records [RFC7673] is strongly encouraged. See below in Section 7.2 for details.

5. State Considerations

Each DNS Push Notification server is capable of handling some finite number of Push Notification subscriptions. This number will vary from server to server and is based on physical machine characteristics, network bandwidth, and operating system resource allocation. After a client establishes a connection to a DNS server, each subscription is individually accepted or rejected. Servers may employ various techniques to limit subscriptions to a manageable level. Correspondingly, the client is free to establish simultaneous connections to alternate DNS servers that support DNS Push Notifications for the zone and distribute subscriptions at its discretion. In this way, both clients and servers can react to resource constraints. Token bucket rate limiting schemes are also effective in providing fairness by a server across numerous client requests.

6. Protocol Operation

The DNS Push Notification protocol is a session-oriented protocol, and makes use of DNS Session Signaling [SessSig].

For details of the DNS Session Signaling message format refer to the DNS Session Signaling specification [SessSig]. Those details are not repeated here.

DNS Push Notification clients and servers MUST support DNS Session Signaling, but the server MUST NOT issue any DNS Session Signaling operations until after the client has first initiated a DNS Session Signaling operation of its own. A single server can support DNS Queries, DNS Updates, and DNS Push Notifications (using DNS Session Signaling) on the same TCP port, and until the client has sent at least one DNS Session Signaling operation the server does not know what kind of client has connected to it. Once the client has indicated willingness to use DNS Session Signaling operations by sending one of its own, either side of the connection may then initiate further Session Signaling operations at any time.

A DNS Push Notification exchange begins with the client discovering the appropriate server, using the procedure described in Section 6.1, and then making a TLS/TCP connection to it.

A typical DNS Push Notification client will immediately issue a DNS Session Signaling Keepalive operation to request a session timeout or keepalive interval longer than the the 15-second defaults, but this is NOT REQUIRED. A DNS Push Notification client MAY issue other requests on the connection first, and only issue a DNS Session Signaling Keepalive operation later if it determines that to be necessary.

Once the connection is made, the client may then add and remove Push Notification subscriptions. In accordance with the current set of active subscriptions the server sends relevant asynchronous Push Notifications to the client. Note that a client MUST be prepared to receive (and silently ignore) Push Notifications for subscriptions it has previously removed, since there is no way to prevent the situation where a Push Notification is in flight from server to client while the client’s UNSUBSCRIBE message cancelling that subscription is simultaneously in flight from client to server.

The exchange between client and server terminates when either end closes the TCP connection with a TCP FIN or RST.

6.1. Discovery

The first step in DNS Push Notification subscription is to discover an appropriate DNS server that supports DNS Push Notifications for the desired zone. The client MUST also determine which TCP port on the server is listening for connections, which need not be (and often is not) the typical TCP port 53 used for conventional DNS, or TCP port 853 used for DNS over TLS [RFC7858].

1. The client begins the discovery by sending a DNS query to its local resolver, with record type SOA [RFC1035], for the domain name to which it wishes to subscribe.

2. If the SOA record exists, it MUST be returned in the Answer Section of the response. If not, the local resolver SHOULD include the SOA record for the zone of the requested name in the Authority Section.

3. If no SOA record is returned, the client then strips off the leading label from the requested name. If the resulting name has at least one label in it, the client sends a new SOA query and processing continues at step 2 above. If the resulting name is empty (the root label) then this is a network configuration error and the client gives up. The client MAY retry the operation at a later time, of the client’s choosing, such after a change in network attachment.

4. Once the SOA is known (either by virtue of being seen in the Answer Section, or in the Authority Section), the client sends a DNS query with type SRV [RFC2782] for the record name "_dns-push-tls._tcp.<zone>", where <zone> is the owner name of the discovered SOA record.

5. If the zone in question does not offer DNS Push Notifications then SRV record MUST NOT exist and the SRV query will return a negative answer.

6. If the zone in question is set up to offer DNS Push Notifications then this SRV record MUST exist. The SRV "target" contains the name of the server providing DNS Push Notifications for the zone. The port number on which to contact the server is in the SRV record "port" field. The address(es) of the target host MAY be included in the Additional Section, however, the address records SHOULD be authenticated before use as described below in Section 7.2 [RFC7673].

7. More than one SRV record may be returned. In this case, the "priority" and "weight" values in the returned SRV records are

used to determine the order in which to contact the servers for subscription requests. As described in the SRV specification [RFC2782], the server with the lowest "priority" is first contacted. If more than one server has the same "priority", the "weight" indicates the weighted probability that the client should contact that server. Higher weights have higher probabilities of being selected. If a server is not reachable or is not willing to accept a subscription request, then a subsequent server is to be contacted.

Each time a client makes a new DNS Push Notification subscription connection, it SHOULD repeat the discovery process in order to determine the preferred DNS server for subscriptions at that time.

Note that this repeated discovery step is typically very fast and typically results in no queries on the network. The client device MUST respect the DNS TTL values on records it receives, and store them in its local cache with this lifetime. This means that, as long as the DNS TTL values on the authoritative records were set to reasonable values, repeated application of this discovery process can be completed nearly instantaneously by the client, using only locally-stored cached data.

6.2. DNS Push Notification SUBSCRIBE

After connecting, and requesting a longer idle timeout and/or keepalive interval if necessary, a DNS Push Notification client then indicates its desire to receive DNS Push Notifications for a given domain name by sending a SUBSCRIBE request over the established TLS connection to the server. A SUBSCRIBE request is encoded in a DNS Session Signaling [SessSig] message. This specification defines a DNS Session Signaling TLV for DNS Push Notification SUBSCRIBE Requests/Responses (tentatively Session Signaling Type Code 0x40).

A server MUST NOT initiate a SUBSCRIBE request.

6.2.1. SUBSCRIBE Request

A SUBSCRIBE request message begins with the standard DNS Session Signaling 12-byte header [SessSig], followed by the SUBSCRIBE TLV. The SSOP-DATA for the the SUBSCRIBE TLV is as follows:

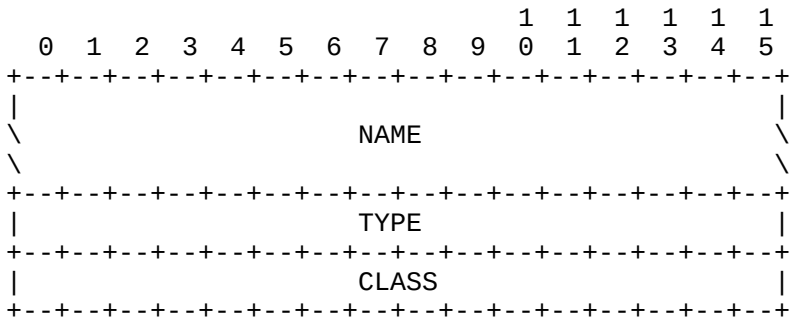


Figure 1

The MESSAGE ID field MUST be set to a unique value, that the client is not using for any other active operation on this connection. For the purposes here, a MESSAGE ID is in use on this connection if the client has used it in a request for which it has not yet received a response, or if if the client has used it for a subscription which it has not yet cancelled using UNSUBSCRIBE. In the SUBSCRIBE response the server MUST echo back the MESSAGE ID value unchanged.

In the SUBSCRIBE TLV the SSOP-TYPE is SUBSCRIBE (tentatively 0x40). The SSOP-LENGTH is the length of the SSOP-DATA that follows, which specifies the name, type, and class of the record(s) being sought.

A SUBSCRIBE request MUST contain exactly one question. The SUBSCRIBE TLV has no QDCOUNT field to specify more than one question. Since SUBSCRIBE requests are sent over TCP, multiple SUBSCRIBE request messages can be concatenated in a single TCP stream and packed efficiently into TCP segments.

If accepted, the subscription will stay in effect until the client cancels the subscription using UNSUBSCRIBE or until the connection between the client and the server is closed.

SUBSCRIBE requests on a given connection MUST be unique. A client MUST NOT send a SUBSCRIBE message that duplicates the NAME, TYPE and CLASS of an existing active subscription on that TLS/TCP connection. For the purpose of this matching, the established DNS case-insensitivity for US-ASCII letters applies (e.g., "foo.com" and "Foo.com" are the same). If a server receives such a duplicate

SUBSCRIBE message this is an error and the server MUST immediately immediately terminate the connection with a TCP RST (or equivalent for other protocols).

DNS wildcarding is not supported. That is, a wildcard ("*") in a SUBSCRIBE message matches only a literal wildcard character ("*") in the zone, and nothing else.

Aliasing is not supported. That is, a CNAME in a SUBSCRIBE message matches only a literal CNAME record in the zone, and nothing else.

A client may SUBSCRIBE to records that are unknown to the server at the time of the request (providing that the name falls within one of the zone(s) the server is responsible for) and this is not an error. The server MUST accept these requests and send Push Notifications if and when matching records are found in the future.

If neither TYPE nor CLASS are ANY (255) then this is a specific subscription to changes for the given NAME, TYPE and CLASS. If one or both of TYPE or CLASS are ANY (255) then this subscription matches any type and/or any class, as appropriate.

NOTE: A little-known quirk of DNS is that in DNS QUERY requests, QTYPE and QCLASS 255 mean "ANY" not "ALL". They indicate that the server should respond with ANY matching records of its choosing, not necessarily ALL matching records. This can lead to some surprising and unexpected results, were a query returns some valid answers but not all of them, and makes QTYPE=ANY queries less useful than people sometimes imagine.

When used in conjunction with SUBSCRIBE, TYPE and CLASS 255 should be interpreted to mean "ALL", not "ANY". After accepting a subscription where one or both of TYPE or CLASS are 255, the server MUST send Push Notification Updates for ALL record changes that match the subscription, not just some of them.

6.2.2. SUBSCRIBE Response

Each SUBSCRIBE request generates exactly one SUBSCRIBE response from the server.

A SUBSCRIBE response message begins with the standard DNS Session Signaling 12-byte header [SessSig], possibly followed by one or more optional Modifier TLVs, such as a Retry Delay Modifier TLV.

The MESSAGE ID field MUST echo the value given in the ID field of the SUBSCRIBE request. This is how the client knows which request is being responded to.

A SUBSCRIBE response message MUST NOT contain a Session Signaling Operation TLV. The Session Signaling Operation TLV is NOT copied from the SUBSCRIBE request.

In the SUBSCRIBE response the RCODE indicates whether or not the subscription was accepted. Supported RCODEs are as follows:

Mnemonic	Value	Description
NOERROR	0	SUBSCRIBE successful.
FORMERR	1	Server failed to process request due to a malformed request.
SERVFAIL	2	Server failed to process request due to resource exhaustion.
NXDOMAIN	3	NOT APPLICABLE. DNS Push Notification servers MUST NOT return NXDOMAIN errors in response to SUBSCRIBE requests.
NOTIMP	4	Server does not recognize DNS Session Signaling Opcode.
REFUSED	5	Server refuses to process request for policy or security reasons.
NOTAUTH	9	Server is not authoritative for the requested name.
SSOPNOTIMP	11	SUBSCRIBE operation not supported.

SUBSCRIBE Response codes

This document specifies only these RCODE values for SUBSCRIBE Responses. Servers sending SUBSCRIBE Responses SHOULD use one of these values. However, future circumstances may create situations where other RCODE values are appropriate in SUBSCRIBE Responses, so clients MUST be prepared to accept SUBSCRIBE Responses with any RCODE value.

If the server sends a nonzero RCODE in the SUBSCRIBE response, either the client is (at least partially) misconfigured or the server resources are exhausted. In either case, the client shouldn't retry the subscription right away. Either end can terminate the connection, but the client may want to try this subscription again or it may have other successful subscriptions that it doesn't want to abandon. If the server sends a nonzero RCODE then it SHOULD append a Retry Delay Modifier TLV [SessSig] to the response specifying a delay before the client attempts this operation again. Recommended values for the delay for different RCODE values are given below:

For RCODE = 1 (FORMERR) the delay may be any value selected by the implementer. A value of five minutes is RECOMMENDED, to reduce the risk of high load from defective clients.

For RCODE = 2 (SERVFAIL), which occurs due to resource exhaustion, the delay should be chosen according to the level of server overload and the anticipated duration of that overload. By default, a value of one minute is RECOMMENDED.

For RCODE = 4 (NOTIMP), which occurs on a server that doesn't implement DNS Session Signaling [SessSig], it is unlikely that the server will begin supporting DNS Session Signaling in the next few minutes, so the retry delay SHOULD be one hour.

For RCODE = 5 (REFUSED), which occurs on a server that implements DNS Push Notifications, but is currently configured to disallow DNS Push Notifications, the retry delay may be any value selected by the implementer and/or configured by the operator.
This is a misconfiguration, since this server is listed in a "_dns-push-tls._tcp.<zone>" SRV record, but the server itself is not currently configured to support DNS Push Notifications. Since it is possible that the misconfiguration may be repaired at any time, the retry delay should not be set too high. By default, a value of 5 minutes is RECOMMENDED.

For RCODE = 9 (NOTAUTH), which occurs on a server that implements DNS Push Notifications, but is not configured to be authoritative for the requested name, the retry delay may be any value selected by the implementer and/or configured by the operator.
This is a misconfiguration, since this server is listed in a "_dns-push-tls._tcp.<zone>" SRV record, but the server itself is not currently configured to support DNS Push Notifications for that zone. Since it is possible that the misconfiguration may be repaired at any time, the retry delay should not be set too high. By default, a value of 5 minutes is RECOMMENDED.

For RCODE = 11 (DNS Push SUBSCRIBE operation not supported), which occurs on a server that doesn't implement DNS Push Notifications, it is unlikely that the server will begin supporting DNS Push Notifications in the next few minutes, so the retry delay SHOULD be one hour.

For other RCODE values, the retry delay should be set by the server as appropriate for that error condition. By default, a value of 5 minutes is RECOMMENDED.

For RCODE = 9 (NOTAUTH), the time delay applies to requests for other names falling within the same zone. Requests for names falling within other zones are not subject to the delay. For all other RCODEs the time delay applies to all subsequent requests to this server.

After sending an error response the server MAY allow the connection to remain open, or MAY send a DNS Push Notification Retry Delay Operation TLV and then close the TCP connection, as described in the DNS Session Signaling specification [SessSig]. Clients MUST correctly handle both cases.

6.3. DNS Push Notification Updates

Once a subscription has been successfully established, the server generates PUSH messages to send to the client as appropriate. In the case that the answer set was non-empty at the moment the subscription was established, an initial PUSH message will be sent immediately following the SUBSCRIBE Response. Subsequent changes to the answer set are then communicated to the client in subsequent PUSH messages.

6.3.1. PUSH Message

A PUSH message begins with the standard DNS Session Signaling 12-byte header [SessSig], followed by the PUSH TLV.

The MESSAGE ID field MUST be set to a unique value, that the server is not currently using for any other active outgoing request that it has sent on this connection. The MESSAGE ID in the outgoing PUSH message is selected by the server and has no relationship to the MESSAGE ID in any of the client subscriptions it may relate to. In the PUSH response the client MUST echo back the MESSAGE ID value unchanged.

In the PUSH TLV the SSOP-TYPE is PUSH (tentatively 0x41). The SSOP-LENGTH is the length of the SSOP-DATA that follows, which specifies the changes being communicated.

The SSOP-DATA contains one or more Update records, in customary Resource Record format, as used in DNS Update [RFC2136] messages. A PUSH Message MUST contain at least one Update record. If a PUSH Message is received that contains no Update records this is a fatal error, and the receiver MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

The SSOP-DATA contains the relevant change information for the client, formatted identically to a DNS Update [RFC2136]. To recap:

Delete all RRsets from a name:
TTL=0, CLASS=ANY, RDLENGTH=0, TYPE=ANY.

Delete an RRset from a name:
TTL=0, CLASS=ANY, RDLENGTH=0;
TYPE specifies the RRset being deleted.

Delete an individual RR from a name:
TTL=0, CLASS=NONE;
TYPE, RDLENGTH and RDATA specifies the RR being deleted.

Add to an RRset:
TTL, CLASS, TYPE, RDLENGTH and RDATA specifies the RR being added.

When processing the records received in a PUSH Message, the receiving client MUST validate that the records being added or deleted correspond with at least one currently active subscription on that connection. Specifically, the record name MUST match the name given in the SUBSCRIBE request, subject to the usual established DNS case-insensitivity for US-ASCII letters. If the TYPE in the SUBSCRIBE request was not ANY (255) then the TYPE of the record must match the

TYPE given in the SUBSCRIBE request. If the CLASS in the SUBSCRIBE request was not ANY (255) then the CLASS of the record must match the CLASS given in the SUBSCRIBE request. If a matching active subscription on that connection is not found, then that individual record addition/deletion is silently ignored. Processing of other additions and deletions in this message is not affected. The TCP connection is not closed. This is to allow for the unavoidable race condition where a client sends an outbound UNSUBSCRIBE while inbound PUSH messages for that subscription from the server are still in flight.

In the case where a single change affects more than one active subscription, only one PUSH message is sent. For example, a PUSH message adding a given record may match both a SUBSCRIBE request with the same TYPE and a different SUBSCRIBE request with TYPE=ANY. It is not the case that two PUSH messages are sent because the new record matches two active subscriptions.

The server SHOULD encode change notifications in the most efficient manner possible. For example, when three AAAA records are deleted from a given name, and no other AAAA records exist for that name, the server SHOULD send a "delete an RRset from a name" PUSH message, not three separate "delete an individual RR from a name" PUSH messages. Similarly, when both an SRV and a TXT record are deleted from a given name, and no other records of any kind exist for that name, the server SHOULD send a "delete all RRsets from a name" PUSH message, not two separate "delete an RRset from a name" PUSH messages.

A server SHOULD combine multiple change notifications in a single PUSH message when possible, even if those change notifications apply to different subscriptions. Conceptually, a PUSH message is a connection-level mechanism, not a subscription-level mechanism.

Reception of a PUSH message by a client generates a PUSH response back to the server.

The TTL of an added record is stored by the client and decremented as time passes, with the caveat that for as long as a relevant subscription is active, the TTL does not decrement below 1 second. For as long as a relevant subscription remains active, the client SHOULD assume that when a record goes away the server will notify it of that fact. Consequently, a client does not have to poll to verify that the record is still there. Once a subscription is cancelled (individually, or as a result of the TCP connection being closed) record ageing resumes and records are removed from the local cache when their TTL reaches zero.

6.3.2. PUSH Response

Each PUSH message generates exactly one PUSH response from the receiver.

A PUSH response message begins with the standard DNS Session Signaling 12-byte header [SessSig], possibly followed by one or more optional Modifier TLVs, such as a Retry Delay Modifier TLV.

The MESSAGE ID field MUST echo the value given in the ID field of the PUSH message.

A PUSH response message MUST NOT contain a Session Signaling Operation TLV. The Session Signaling Operation TLV is NOT copied from the PUSH message.

In a PUSH response the RCODE MUST be zero. Receiving a PUSH response with a nonzero RCODE is a fatal error, and the receiver MUST immediately terminate the connection with a TCP RST (or equivalent for other protocols).

6.4. DNS Push Notification UNSUBSCRIBE

To cancel an individual subscription without closing the entire connection, the client sends an UNSUBSCRIBE message over the established TCP connection to the server. The UNSUBSCRIBE message is encoded in a DNS Session Signaling [SessSig] message. This specification defines a DNS Session Signaling TLV for DNS Push Notification UNSUBSCRIBE Requests/Responses (tentatively Session Signaling Type Code 0x42).

A server MUST NOT initiate an UNSUBSCRIBE request.

6.4.1. UNSUBSCRIBE Request

An UNSUBSCRIBE request message begins with the standard DNS Session Signaling 12-byte header [SessSig], followed by the UNSUBSCRIBE TLV.

In the UNSUBSCRIBE TLV the SSOP-TYPE is UNSUBSCRIBE (tentatively 0x42). The SSOP-LENGTH is zero. There is no SSOP-DATA for UNSUBSCRIBE.

The MESSAGE ID field MUST match the value given in the ID field of an active SUBSCRIBE request. This is how the server knows which SUBSCRIBE request is being cancelled. After receipt of the UNSUBSCRIBE request, the SUBSCRIBE request is no longer active. If a server receives an UNSUBSCRIBE message where the MESSAGE ID does not match the ID of an active SUBSCRIBE request the server MUST return a response containing RCODE = 3 (NXDOMAIN).

It is allowable for the client to issue an UNSUBSCRIBE request for a previous SUBSCRIBE request for which the client has not yet received a SUBSCRIBE response. This is to allow for the case where a client starts and stops a subscription in less than the round-trip time to the server. The client is NOT required to wait for the SUBSCRIBE response before issuing the UNSUBSCRIBE request. A consequence of this is that if the client issues an UNSUBSCRIBE request for an as-yet unacknowledged SUBSCRIBE request, and the SUBSCRIBE request is subsequently unsuccessful for some reason, then when the UNSUBSCRIBE request is eventually processed it will be an UNSUBSCRIBE request for a nonexistent subscription, which will result NXDOMAIN response.

Note that when the client issues an UNSUBSCRIBE request for an as-yet unacknowledged SUBSCRIBE request, at that moment the client will have two outstanding DNS Session Signaling operations with same MESSAGE ID, a SUBSCRIBE request and an UNSUBSCRIBE request, which will both receive responses, in that order. When the client has multiple outstanding DNS Session Signaling operations with same MESSAGE ID, care should be taken that when a DNS Session Signaling response message is received for that MESSAGE ID, it is associated with the *first* unacknowledged request.

6.4.2. UNSUBSCRIBE Response

Each UNSUBSCRIBE request generates exactly one UNSUBSCRIBE response from the server.

An UNSUBSCRIBE response message begins with the standard DNS Session Signaling 12-byte header [SessSig], possibly followed by one or more optional Modifier TLVs, such as a Retry Delay Modifier TLV.

The MESSAGE ID field MUST echo the value given in the ID field of the UNSUBSCRIBE request. This is how the client knows which request is being responded to.

An UNSUBSCRIBE response message MUST NOT contain a Session Signaling Operation TLV. The Session Signaling Operation TLV is NOT copied from the UNSUBSCRIBE request.

In the UNSUBSCRIBE response the RCODE indicates whether or not the unsubscribe request was successful. Supported RCODEs are as follows:

Mnemonic	Value	Description
NOERROR	0	UNSUBSCRIBE successful.
FORMERR	1	Server failed to process request due to a malformed request.
NXDOMAIN	3	Specified subscription does not exist.
NOTIMP	4	Server does not recognize DNS Session Signaling Opcode.
SSOPNOTIMP	11	UNSUBSCRIBE operation not supported.

UNSUBSCRIBE Response codes

This document specifies only these RCODE values for UNSUBSCRIBE Responses. Servers sending UNSUBSCRIBE Responses SHOULD use one of these values. However, future circumstances may create situations where other RCODE values are appropriate in UNSUBSCRIBE Responses, so clients MUST be prepared to accept UNSUBSCRIBE Responses with any RCODE value.

Having being successfully revoked with a correctly-formatted UNSUBSCRIBE message (resulting in a response with RCODE NOERROR) the previously referenced subscription is no longer active and the server MAY discard the state associated with it immediately, or later, at the server’s discretion.

Nonzero RCODE values signal some kind of error.

RCODE value FORMERR indicates a message format error.

RCODE value NXDOMAIN indicates a MESSAGE ID that does not correspond to any active subscription.

RCODE values NOTIMP and SSOPNOTIMP should not occur in practice.

A server would only generate NOTIMP if it did not support Session Signaling, and if the server does not support Session Signaling then it should not be possible for a client to have an active subscription to cancel.

Similarly, a server would only generate SSOPNOTIMP if it did not support Push Notifications, and if the server does not support Push Notifications then it should not be possible for a client to have an active subscription to cancel.

Nonzero RCODE values other than NXDOMAIN indicate a serious problem with the client. After sending an error response other than NXDOMAIN, the server SHOULD send a DNS Session Signaling Retry Delay Operation TLV and then close the TCP connection, as described in the DNS Session Signaling specification [SessSig].

6.5. DNS Push Notification RECONFIRM

Sometimes, particularly when used with a Discovery Proxy [DisProx], a DNS Zone may contain stale data. When a client encounters data that it believe may be stale (e.g., an SRV record referencing a target host+port that is not responding to connection requests) the client can send a RECONFIRM request to ask the server to re-verify that the data is still valid. For a Discovery Proxy, this causes it to issue new Multicast DNS requests to ascertain whether the target device is still present. For other types of DNS server, the RECONFIRM operation is currently undefined, and SHOULD result in a NOERROR response, but otherwise need not cause any action to occur. Frequent RECONFIRM operations may be a sign of network unreliability, or some kind of misconfiguration, so RECONFIRM operations MAY be logged or otherwise communicated to a human administrator to assist in detecting, and remedying, such network problems.

If, after receiving a valid RECONFIRM request, the server determines that the disputed records are in fact no longer valid, then subsequent DNS PUSH Messages will be generated to inform interested clients. Thus, one client discovering that a previously-advertised device (like a network printer) is no longer present has the side effect of informing all other interested clients that the device in question is now gone.

6.5.1. RECONFIRM Request

A RECONFIRM request message begins with the standard DNS Session Signaling 12-byte header [SessSig], followed by the RECONFIRM TLV. The SSOP-DATA for the the RECONFIRM TLV is as follows:

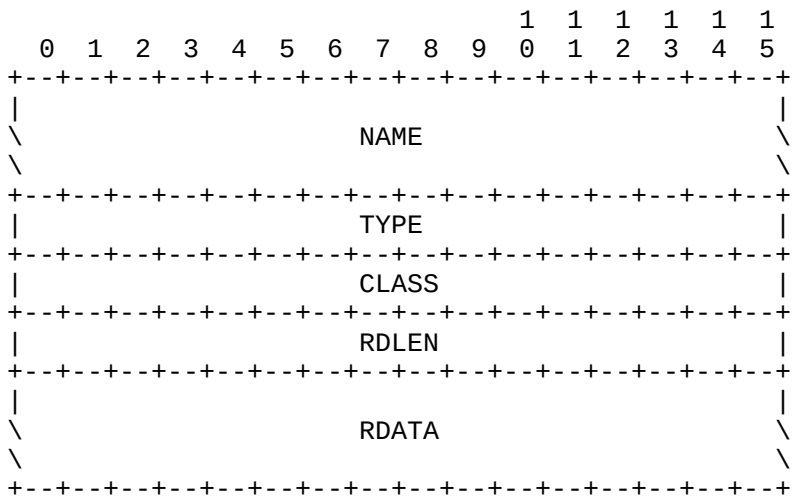


Figure 2

The MESSAGE ID field MUST be set to a unique value, that the client is not using for any other active operation on this connection. For the purposes here, a MESSAGE ID is in use on this connection if the client has used it in a request for which it has not yet received a response, or if if the client has used it for a subscription which it has not yet cancelled using UNSUBSCRIBE. In the RECONFIRM response the server MUST echo back the MESSAGE ID value unchanged.

In the RECONFIRM TLV the SSOP-TYPE is RECONFIRM (tentatively 0x43). The SSOP-LENGTH is the length of the data that follows, which specifies the name, type, class, and content of the record being disputed.

A RECONFIRM request MUST contain exactly one record. The RECONFIRM TLV has no count field to specify more than one record. Since RECONFIRM requests are sent over TCP, multiple RECONFIRM request messages can be concatenated in a single TCP stream and packed efficiently into TCP segments.

TYPE MUST NOT be the value ANY (255) and CLASS MUST NOT be the value ANY (255).

DNS wildcarding is not supported. That is, a wildcard ("*") in a RECONFIRM message matches only a literal wildcard character ("*") in the zone, and nothing else.

Aliasing is not supported. That is, a CNAME in a RECONFIRM message matches only a literal CNAME record in the zone, and nothing else.

6.5.2. RECONFIRM Response

Each RECONFIRM request generates exactly one RECONFIRM response from the server.

A RECONFIRM response message begins with the standard DNS Session Signaling 12-byte header [SessSig], possibly followed by one or more optional Modifier TLVs, such as a Retry Delay Modifier TLV.

The MESSAGE ID field MUST echo the value given in the ID field of the RECONFIRM request. This is how the client knows which request is being responded to.

A RECONFIRM response message MUST NOT contain a Session Signaling Operation TLV. The Session Signaling Operation TLV is NOT copied from the RECONFIRM request.

In the RECONFIRM response the RCODE confirms receipt of the reconfirmation request. Supported RCODEs are as follows:

Mnemonic	Value	Description
NOERROR	0	RECONFIRM accepted.
FORMERR	1	Server failed to process request due to a malformed request.
SERVFAIL	2	Server failed to process request due to resource exhaustion.
NXDOMAIN	3	NOT APPLICABLE. DNS Push Notification servers MUST NOT return NXDOMAIN errors in response to RECONFIRM requests.
NOTIMP	4	Server does not recognize DNS Session Signaling Opcode.
REFUSED	5	Server refuses to process request for policy or security reasons.
NOTAUTH	9	Server is not authoritative for the requested name.
SSOPNOTIMP	11	RECONFIRM operation not supported.

RECONFIRM Response codes

This document specifies only these RCODE values for RECONFIRM Responses. Servers sending RECONFIRM Responses SHOULD use one of these values. However, future circumstances may create situations where other RCODE values are appropriate in RECONFIRM Responses, so clients MUST be prepared to accept RECONFIRM Responses with any RCODE value.

Nonzero RCODE values signal some kind of error.

RCODE value FORMERR indicates a message format error, for example TYPE or CLASS being ANY (255).

RCODE value SERVFAIL indicates that the server is overloaded.

RCODE values NOTIMP indicates that the server does not support Session Signaling, and Session Signaling is required for RECONFIRM requests.

RCODE value REFUSED indicates that the server supports RECONFIRM requests but is currently not configured to accept them from this client.

RCODE value NOTAUTH indicates that the server is not authoritative for the requested name, and can do nothing to remedy the apparent error. Note that there may be future cases in which a server is able to pass on the RECONFIRM request to the ultimate source of the information, and in these cases the server should return NOERROR.

RCODE value SSOPNOTIMP indicates that the server does not support RECONFIRM requests.

Similarly, a server would only generate SSOPNOTIMP if it did not support Push Notifications, and if the server does not support Push Notifications then it should not be possible for a client to have an active subscription to cancel.

Nonzero RCODE values SERVFAIL, REFUSED and SSOPNOTIMP are benign from the client’s point of view. The client may log them to aid in debugging, but otherwise they require no special action.

Nonzero RCODE values other than these three indicate a serious problem with the client. After sending an error response other than one of these three, the server SHOULD send a DNS Session Signaling Retry Delay Operation TLV and then close the TCP connection, as described in the DNS Session Signaling specification [SessSig].

6.6. Client-Initiated Termination

An individual subscription is terminated by sending an UNSUBSCRIBE TLV for that specific subscription, or all subscriptions can be cancelled at once by the client closing the connection. When a client terminates an individual subscription (via UNSUBSCRIBE) or all subscriptions on that connection (by closing the connection) it is signaling to the server that it is longer interested in receiving those particular updates. It is informing the server that the server may release any state information it has been keeping with regards to these particular subscriptions.

After terminating its last subscription on a connection via UNSUBSCRIBE, a client MAY close the connection immediately, or it may keep it open if it anticipates performing further operations on that connection in the future. If a client wishes to keep an idle connection open, it MUST respect the maximum idle time required by the server [SessSig].

If a client plans to terminate one or more subscriptions on a connection and doesn't intend to keep that connection open, then as an efficiency optimization it MAY instead choose to simply close the connection, which implicitly terminates all subscriptions on that connection. This may occur because the client computer is being shut down, is going to sleep, the application requiring the subscriptions has terminated, or simply because the last active subscription on that connection has been cancelled.

When closing a connection, a client will generally do an abortive disconnect, sending a TCP RST. This immediately discards all remaining inbound and outbound data, which is appropriate if the client no longer has any interest in this data. In the BSD Sockets API, sending a TCP RST is achieved by setting the SO_LINGER option with a time of 0 seconds and then closing the socket.

If a client has performed operations on this connection that it would not want lost (like DNS updates) then the client SHOULD do an orderly disconnect, sending a TCP FIN. In the BSD Sockets API, sending a TCP FIN is achieved by calling "shutdown(s,SHUT_WR)" and keeping the socket open until all remaining data has been read from it.

7. Security Considerations

TLS support is REQUIRED in DNS Push Notifications. There is no provision for opportunistic encryption using a mechanism like "STARTTLS".

DNSSEC is RECOMMENDED for DNS Push Notifications. TLS alone does not provide complete security. TLS certificate verification can provide reasonable assurance that the client is really talking to the server associated with the desired host name, but since the desired host name is learned via a DNS SRV query, if the SRV query is subverted then the client may have a secure connection to a rogue server. DNSSEC can provided added confidence that the SRV query has not been subverted.

7.1. Security Services

It is the goal of using TLS to provide the following security services:

Confidentiality: All application-layer communication is encrypted with the goal that no party should be able to decrypt it except the intended receiver.

Data integrity protection: Any changes made to the communication in transit are detectable by the receiver.

Authentication: An end-point of the TLS communication is authenticated as the intended entity to communicate with.

Deployment recommendations on the appropriate key lengths and cypher suites are beyond the scope of this document. Please refer to TLS Recommendations [RFC7525] for the best current practices. Keep in mind that best practices only exist for a snapshot in time and recommendations will continue to change. Updated versions or errata may exist for these recommendations.

7.2. TLS Name Authentication

As described in Section 6.1, the client discovers the DNS Push Notification server using an SRV lookup for the record name "_dns-push-tls._tcp.<zone>". The server connection endpoint SHOULD then be authenticated using DANE TLSA records for the associated SRV record. This associates the target’s name and port number with a trusted TLS certificate [RFC7673]. This procedure uses the TLS Sever Name Indication (SNI) extension [RFC6066] to inform the server of the name the client has authenticated through the use of TLSA records. Therefore, if the SRV record passes DNSSEC validation and a TLSA

record matching the target name is useable, an SNI extension MUST be used for the target name to ensure the client is connecting to the server it has authenticated. If the target name does not have a usable TLSA record, then the use of the SNI extension is optional.

7.3. TLS Compression

In order to reduce the chances of compression-related attacks, TLS-level compression SHOULD be disabled when using TLS versions 1.2 and earlier. In the draft version of TLS 1.3 [I-D.ietf-tls-tls13], TLS-level compression has been removed completely.

7.4. TLS Session Resumption

TLS Session Resumption is permissible on DNS Push Notification servers. The server may keep TLS state with Session IDs [RFC5246] or operate in stateless mode by sending a Session Ticket [RFC5077] to the client for it to store. However, once the connection is closed, any existing subscriptions will be dropped. When the TLS session is resumed, the DNS Push Notification server will not have any subscription state and will proceed as with any other new connection. Use of TLS Session Resumption allows a new TLS connection to be set up more quickly, but the client will still have to recreate any desired subscriptions.

8. IANA Considerations

This document defines the service name: "_dns-push-tls._tcp".
It is only applicable for the TCP protocol.
This name is to be published in the IANA Service Name Registry [RFC6335][SN].

This document defines three DNS Session Signaling TLV types: SUBSCRIBE with (tentative) value 0x40 (64), PUSH with (tentative) value 0x41 (65), UNSUBSCRIBE with (tentative) value 0x42 (66), and RECONFIRM with (tentative) value 0x43 (67).

9. Acknowledgements

The authors would like to thank Kiren Sekar and Marc Krochmal for previous work completed in this field.

This draft has been improved due to comments from Ran Atkinson, Tim Chown, Mark Delany, Ralph Droms, Bernie Volz, Jan Komissar, Manju Shankar Rao, Markus Stenberg, Dave Thaler, and Soraia Zlatkovic.

10. References

10.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <http://www.rfc-editor.org/info/rfc768>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <http://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <http://www.rfc-editor.org/info/rfc1123>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2136] Vixie, P., Ed., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, DOI 10.17487/RFC2136, April 1997, <http://www.rfc-editor.org/info/rfc2136>.

[RFC2782] Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, DOI 10.17487/RFC2782, February 2000, <http://www.rfc-editor.org/info/rfc2782>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <http://www.rfc-editor.org/info/rfc6066>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <http://www.rfc-editor.org/info/rfc6335>.

[RFC6895] Eastlake 3rd, D., "Domain Name System (DNS) IANA Considerations", BCP 42, RFC 6895, DOI 10.17487/RFC6895, April 2013, <http://www.rfc-editor.org/info/rfc6895>.

[RFC7673] Finch, T., Miller, M., and P. Saint-Andre, "Using DNS-Based Authentication of Named Entities (DANE) TLSA Records with SRV Records", RFC 7673, DOI 10.17487/RFC7673, October 2015, <http://www.rfc-editor.org/info/rfc7673>.

[RFC7766] Dickinson, J., Dickinson, S., Bellis, R., Mankin, A., and D. Wessels, "DNS Transport over TCP - Implementation Requirements", RFC 7766, DOI 10.17487/RFC7766, March 2016, <http://www.rfc-editor.org/info/rfc7766>.

[SessSig] Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S., Mankin, A., and T. Pusateri, "DNS Session Signaling", draft-ietf-dnsop-session-signal-02 (work in progress), March 2017.

[SN] "Service Name and Transport Protocol Port Number Registry", <http://www.iana.org/assignments/service-names-port-numbers/>.

10.2. Informative References

[DisProx] Cheshire, S., "Hybrid Unicast/Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-06 (work in progress), March 2017.

[I-D.dukkipati-tcpm-tcp-loss-probe]
Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), February 2013.

[I-D.sekar-dns-llq]
Sekar, K., "DNS Long-Lived Queries", draft-sekar-dns-llq-01 (work in progress), August 2006.

[IPJ.9-4-TCPSYN]
Eddy, W., "Defenses Against TCP SYN Flooding Attacks", The Internet Protocol Journal, Cisco Systems, Volume 9, Number 4, December 2006.

[obs] "Observer Pattern", <https://en.wikipedia.org/wiki/Observer_pattern>.

[RFC1996] Vixie, P., "A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)", RFC 1996, DOI 10.17487/RFC1996, August 1996, <http://www.rfc-editor.org/info/rfc1996>.

[RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", RFC 4287, DOI 10.17487/RFC4287, December 2005, <http://www.rfc-editor.org/info/rfc4287>.

[RFC4953] Touch, J., "Defending TCP Against Spoofing Attacks", RFC 4953, DOI 10.17487/RFC4953, July 2007, <http://www.rfc-editor.org/info/rfc4953>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <http://www.rfc-editor.org/info/rfc5077>.

[RFC6281] Cheshire, S., Zhu, Z., Wakikawa, R., and L. Zhang, "Understanding Apple’s Back to My Mac (BTMM) Service", RFC 6281, DOI 10.17487/RFC6281, June 2011, <http://www.rfc-editor.org/info/rfc6281>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <http://www.rfc-editor.org/info/rfc6762>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <http://www.rfc-editor.org/info/rfc6763>.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <http://www.rfc-editor.org/info/rfc6824>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <http://www.rfc-editor.org/info/rfc7525>.

[RFC7858] Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., and P. Hoffman, "Specification for DNS over Transport Layer Security (TLS)", RFC 7858, DOI 10.17487/RFC7858, May 2016, <http://www.rfc-editor.org/info/rfc7858>.

[XEP0060] Millard, P., Saint-Andre, P., and R. Meijer, "Publish-Subscribe", XSF XEP 0060, July 2010.

Authors' Addresses

Tom Pusateri
Seeking affiliation
Hilton Head Island, SC
USA

Phone: +1 843 473 7394
Email: pusateri@bangj.com

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
USA

Phone: +1 408 974 3207
Email: cheshire@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 4, 2017

T. Lemon
Nominum, Inc.

October 31, 2016

Stateful Multi-Link DNS Service Discovery
draft-lemon-stateful-dnssd-00

Abstract

This document proposes a stateful model for automating Multi-Link DNS Service Discovery, as an extension to the existing solution, which relies entirely on multicast DNS for discovering services, and does not formally maintain DNS zone state. When fully deployed this will confer several advantages: the ability to do DNS zone transfers, integrating with existing DNS infrastructure; the elimination of the need for regular multicast queries; and the ability for services to securely register and defend their names, preventing malicious spoofing of services on the network.

Table of Contents

1. Introduction 2
2. Terminology 3
3. Overview 3
4. Service Behavior 4
4.1. Detecting Stateful ML-DNSSD 4
4.2. Publishing Services when Stateful ML-DNSSD is present . . 4
4.3. Maintenance 5
5. Discovery 6
6. DNS Service Infrastructure 6
7. Legacy Service Discovery 6
8. Security Considerations 6
9. Normative References 7
Author's Address 7

1. Introduction

This document describes a way of doing DNS service discovery using DNS updates [RFC2136] rather than Multicast DNS[RFC6762]. Update validation is done on the same basis as Multicast DNS validation: the assumption is that a device connected to a local link is permitted to advertise services. However, in contrast to mDNS, which provides no mechanism for defending claims made by services, we propose that services should publish keys when initially registering names, and use SIG(0) authentication [RFC2931] when issuing DNS updates, using the published key.

Advantages of this proposal over the Multicast DNS Hybrid proposal [I-D.ietf-dnssd-hybrid] are:

- Service advertisement does not require multicast.
- Names are stored in DNS zone databases, and therefore can be published using standard DNS protocol features such as zone transfers.
- Names can be defended by services that register them, so that it is difficult for an attacker to spoof an existing service.

There are, however, disadvantages to this approach. The first disadvantage is that this proposal does not actually eliminate multicast except in the case that all local services implement the

new update mechanism. Because this approach maintains state, and that state must include existing services that only support advertising via Multicast DNS, additional complexity is required to avoid retaining stale information; this complexity is not required for the stateless model proposed in the mDNS hybrid specification.

Another disadvantage of this approach is that it requires a stable naming infrastructure, and requires forwarders on each local link.

Some sites may find it preferable to rely on the stateless model for this reason. However, the stateful model provides sufficient advantages that it will make sense for some sites to implement it, even in the legacy mode that still supports service discovery using Multicast DNS.

2. Terminology

For the sake of brevity this document uses a number of abbreviations. These are expanded here:

mDNS Multicast DNS

ML-DNSSD Multi-Link DNS Service Discovery

3. Overview

Stateful Multi-Link DNS service discovery attempts to provide stateful service that is otherwise equivalent to Hybrid Unicast/ Multicast DNS-Based Service Discovery, except that where possible multicast is avoided, and DNS zones are maintained such that full interoperation with the DNS is possible.

In order to accomplish this, service providers detect whether the local network supports stateful operation. If not, they simply provide service using mDNS as before. If so, they advertise services solely using DNS updates.

The DNS infrastructure is prepared to take DNS updates from devices on served networks; each unique link has a DNS forwarder that can detect that a packet originated locally and was not forwarded; this serves as validation that the service can be advertised.

Legacy services are supported using the same query process used in the hybrid model. Unlike with the hybrid model, however, discovered services are added to DNS zones.

As with the Hybrid model, services are discovered using unicast DNS. Multicast DNS service discovery is not usable on networks offering stateful multi-link DNS service discovery.

4. Service Behavior

Hosts offering services using DNS service discovery must advertise these services. When a host offering services is connected to a network that does not offer stateful ML-DNSSD, it offers service discovery using Multicast DNS. When stateful ML-DNSSD is offered, the host does not offer service discovery using Multicast DNS.

4.1. Detecting Stateful ML-DNSSD

In order to detect the presence of stateful ML-DNSSD, the host first performs registration domain discovery as in section 11 of [RFC6763] to acquire the name of the recommended default domain for registering services. If this process fails, Stateful ML-DNSSD is not present. If the process succeeds, the host looks for a PTR record using the well-known name "_mldnssd.<domain>". If a PTR record is present, stateful ML-DNSSD is present.

Whenever a host detects a change to the link, a change to the IP addresses of the DNS resolvers provided on the link, or a change to the set of prefixes available on the link, the host re-tries the ML-DNSSD detection process.

4.2. Publishing Services when Stateful ML-DNSSD is present

When stateful ML-DNSSD is present, a host adds its own information into the DNS. This information is added into three separate domains, as described in [I-D.ietf-dnssd-hybrid], section 4. The subdomain for services and the subdomain for names are a single subdomain, the recommended default domain for registering services. The IPv4 and IPv6 reverse-mapping zones are discovered by querying the well-known names "_inaddr_zone.<domain>" and "_ipv6_zone.<domain>" for PTR records. Each PTR record points to a specific zone to which updates are sent for IPv4 and IPv6 PTR records.

When a host that offers service first starts, it generates a key that is used to authenticate its DNS updates. This key is included whenever updating the service’s name.

There are four domain names that are updated when a service advertises itself: the human-readable name, the machine-readable name, the service entry or entries, and the resverse-mapping pointers. The update proceeds by first adding or updating the machine-readable name, then adding a PTR record from the human-

readable name to the machine-readable name, then adding the reverse-mapping pointers, then adding the service. All updates are signed using SIG(0), authenticated with the private half of the host's key.

To add the machine-readable name, the host creates a DNS update that adds its name. The update is predicated on the nonexistence of the name. The update includes A and AAAA records for all of the hosts IP addresses except its link-local addresses. If this update succeeds, the machine-readable name has been added.

The update can fail for one of two reasons: either the signature was invalid, or the name already exists. In the former case, there is a bug, and the host should revert to providing service using mDNS.

Otherwise, if the update fails, the name already exists. The host creates a new update that deletes any A and AAAA RRsets and adds A and AAAA as before. There is no predicate for this update because the server should reject it if the name belongs to some other host (that is, has a different key). If this update fails, the host chooses a new machine-readable name and restarts the process.

The host then creates a PTR record under "Human Readable Name.<domain>" pointing to the machine-readable name. If this fails, the host must choose a different name and attempt to add it, until successful.

The host now creates updates for the reverse-mapping name of every IPv4 address it has that is not a link-local address, and adds a PTR record for each, pointing back at the machine-readable name. These adds should not fail. The process is repeated for every IPv6 address that is not link-local.

Finally, the host updates the well-known name for its service or services, adding an entry for each one. These names may already have SRV RRtypes, so this update must add records.

TODO: consider whether this is really the right way to do this--it's really complicated, and might be better done as a single HTTP request.

4.3. Maintenance

Whenever the host adds its service to the DNS, it queries the machine-readable name to see what the TTL is. When 80% of that TTL has expired, the host refreshes all of its records. This prevents the records from being cleaned up by the DNS server as stale.

If the host is being shut down cleanly, it may remove all names and SRV records that it has added, or may remove all SRV records, leaving everything else intact in order to reserve the name. In most cases, it is better to leave the name.

5. Discovery

Service Discovery is done as per RFC 6763. Service discovery defaults to '.local', which is resolved using mDNS. If ML-DNSSD is present in any form, hosts doing service discovery should successfully discover this following the method described in RFC 6763. The service appears to the host doing service discovery the same way whether the hybrid model or the stateful model is being used. Hosts do not do mDNS if ML-DNSSD is present.

In order to support progressive queries in situations where legacy service discovery is in operation, hosts should use DNS push [I-D.ietf-dnssd-push].

6. DNS Service Infrastructure

Updates are sent to a forwarder on the local link. The forwarder uses neighbor discovery or ARP to validate each of the IP addresses presented in an A or AAAA record. Updates that do not come from local hosts are silently discarded. Other updates are forwarded to the primary name server without changes.

The primary server validates all updates by using the key stored on the machine-readable name to which the update points. If the update is an update of the machine-readable name, the update is validated based on the key stored at that name, if any, or else using the key contained in the update.

Any number of secondaries may be configured. Secondaries may also serve as forwarders if appropriate.

7. Legacy Service Discovery

Service discovery done as in mdns-hybrid, except that state is retained. State is periodically probed; stale state is discarded. Discovery service listens for initial service announcements.

8. Security Considerations

Any host on a network on which service discovery is supported can advertise services, which might be spoofed so as to capture private information. One solution to this is to only accept updates from designated infrastructure networks, so that networks to which regular

users connect are not permitted to advertise services. This will, however, limit the usefulness of various services which may be present on user devices.

It may be possible to only allow anonymous pairing [I-D.ietf-dnssd-pairing] on public-facing networks, so that infrastructure services cannot be advertised, but users can still rendezvous.

9. Normative References

[RFC2136] Vixie, P., Ed., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, DOI 10.17487/RFC2136, April 1997, <http://www.rfc-editor.org/info/rfc2136>.

[RFC2931] Eastlake 3rd, D., "DNS Request and Transaction Signatures (SIG(0)s)", RFC 2931, DOI 10.17487/RFC2931, September 2000, <http://www.rfc-editor.org/info/rfc2931>.

[RFC6762] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <http://www.rfc-editor.org/info/rfc6762>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <http://www.rfc-editor.org/info/rfc6763>.

[I-D.ietf-dnssd-pairing] Huitema, C. and D. Kaiser, "Device Pairing Using Short Authentication Strings", draft-ietf-dnssd-pairing-00 (work in progress), October 2016.

[I-D.ietf-dnssd-hybrid] Cheshire, S., "Hybrid Unicast/Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-03 (work in progress), February 2016.

[I-D.ietf-dnssd-push] Pusateri, T. and S. Cheshire, "DNS Push Notifications", draft-ietf-dnssd-push-08 (work in progress), July 2016.

Author’s Address

Ted Lemon
Nominum, Inc.
800 Bridge Parkway
Redwood City, California 94065
United States of America

Phone: +1 650 381 6000
Email: ted.lemon@nominum.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 14, 2017

T. Lemon
Nominum, Inc.
D. Migault
Ericsson
March 13, 2017

Simple Homenet Naming and Service Discovery Architecture
draft-tldm-simple-homenet-naming-00

Abstract

This document describes a simple name resolution and service discovery architecture for homenets. This architecture covers local publication of names, as well as name resolution for local and global names.

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 September 14, 2017.

Copyright Notice

Table of Contents

1. Introduction 2
1.1. Existing solutions 3
2. Terminology 4
3. Name Resolution 4
3.1. Configuring Resolvers 4
3.2. Configuring Service Discovery 5
3.3. Resolution of local names 5
3.4. DNSSEC Validation 6
3.5. Support for Multiple Provisioning Domains 6
3.6. Using the Local Namespace While Away From Home 7
4. Management Considerations 7
5. Privacy Considerations 7
6. Security Considerations 8
7. IANA considerations 8
8. Normative References 8
Authors' Addresses 9

1. Introduction

Associating domain names with hosts on the Internet is a key factor in enabling communication with hosts, particularly for service discovery. This document describes a simple way of providing name service and service discovery for homenets. In principle, it may make sense to be able to publish names of devices on the homenet, so that services on the homenet can be accessed outside of the homenet. Such publication is out of scope for this document. It may be desirable to secure the homenet zone using DNSSEC. This is likewise out of scope for this document.

In order to provide name service, several provisioning mechanisms must be available:

- Provisioning of a domain name under which names can be published and services advertised
- Associating names that are subdomains of that name with hosts.
- Advertising services available on the local network by publishing resource records on those names.
- Distribution of names published in that namespace to servers that can be queried in order to resolve names
- Correct advertisement of name servers that can be queried in order to resolve names

- Timely removal of published names and resource records when they are no longer in use

Homenet adds the following considerations:

1. Some names may be published in a broader scope than others. For example, it may be desirable to advertise some homenet services to users who are not connected to the homenet. However, it is unlikely that all services published on the home network would be appropriate to publish outside of the home network. In many cases, no services will be appropriate to publish outside of the network, but the ability to do so is required.

2. Users cannot be assumed to be skilled or knowledgeable in name service operation, or even to have any sort of mental model of how these functions work. All of the operations mentioned here must reliably function automatically, without any user intervention or debugging.

3. Because user intervention cannot be required, naming conflicts must be resolved automatically, and, to the extent possible, transparently.

4. Hosts that do not implement any homenet-specific capabilities must still be able to discover and access services on the homenet, to the extent possible.

5. Devices that provide services must be able to publish those services on the homenet, and those services must be available from any part of the homenet, not just the link to which the device is attached.

6. Homenet explicitly supports multihoming--connecting to more than one Internet Service Provider--and therefore support for multiple provisioning domains [6] is required to deal with situations where the DNS may give a different answer depending on whether caching resolvers at one ISP or another are queried.

1.1. Existing solutions

Previous attempts to automate naming and service discovery in the context of a home network are able to function with varying degrees of success depending on the topology of the home network. For example, Multicast DNS [4] can provide naming and service discovery [5], but only within a single multicast domain.

The Domain Name System provides a hierarchical namespace [1], a mechanism for querying name servers to resolve names [2], a mechanism

for updating namespaces by adding and removing names [3], and a mechanism for discovering services [5]. Unfortunately, DNS provides no mechanism for automatically provisioning new namespaces, and secure updates to namespaces require pre-shared keys, which won't work for an unmanaged network. DHCP can be used to populate names in a DNS namespace; however at present DHCP cannot provision service discovery information.

Hybrid Multicast DNS [7] proposes a mechanism for extending multicast DNS beyond a single multicast domain.. However, it has serious shortcomings as a solution to the Homenet naming problem. The most obvious shortcoming is that it requires that every multicast domain have a separate name. This then requires that the homenet generate names for every multicast domain, and requires that the end user have a mental model of the topology of the network in order to guess on which link a given service may appear. [xxx is this really true at the UI?]

2. Terminology

This document uses the following terms and abbreviations:

HNR Homenet Router

ISP Internet Service Provider

GNRP Global Name Registration Provider

3. Name Resolution

3.1. Configuring Resolvers

Hosts on the homenet receive a set of resolver IP addresses using either DHCP or RA. IPv4-only hosts will receive IPv4 addresses of resolvers, if available, over DHCP. IPv6-only hosts will receive resolver IPv6 addresses using either stateful (if available) or stateless DHCPv6, or through the domain name option in router advertisements. All homenet routers provide resolver information using both stateless DHCPv6 and RA; support for stateful DHCPv6 and DHCPv4 is optional, however if either service is offered, resolver addresses will be provided using that mechanism as well. Resolver IP addresses will always be IP addresses on the local link: every HNR is required to provide name resolution service. This is necessary to allow DNS update using presence on-link as a mechanism for rejecting off-network attacks.

3.2. Configuring Service Discovery

DNS-SD uses several default domains for advertising local zones that are available for service discovery. These include the '.local' domain, which is searched using mDNS, and also the IPv4 and IPv6 reverse zone corresponding to the prefixes in use on the local network. For the homenet, no support for queries against the ".local" zone is provided by HNRs: a ".local" query will be satisfied or not by services present on the local link. This should not be an issue: all known implementations of DNSSD will do unicast queries using the DNS protocol.

Service discovery is configured using the technique described in Section 11 of DNS-Based Service Discovery [5]. HNRs will answer domain enumeration ueries against every IPv4 address prefix advertised on a homenet link, and every IPv6 address prefix advertised on a homenet link, including prefixes derived from the homenet's ULA(s). Whenever the "<domain>" sequence appears in this section, it references each of the domains mentioned in this paragraph.

Homenets advertise the availability of several browsing zones in the "b._dns_sd.<domain>" subdomain. By default, the TBD1 domain is advertised. Similarly, TBD1 is advertised as the default browsing and service registration domain under "db._dns_sd.<domain>", "r._dns_sd.<domain>", "dr._dns_sd.<domain>" and "lb._dns_sd.<domain>".

3.3. Resolution of local names

Local names appear as subdomains of [TBD1]. These names can only be resolved within the homenet; not only is [TBD1] not a globally unique name, but queries from outside of the homenet for any name, on or off the homenet, must be rejected with a REFUSED response.

In addition, names can appear as subdomains of the locally-served 'in-addr.arpa' or 'ip6.addr' zone that corresponding to the ULA that is in use on the homenet. IP addresses and names advertised locally MUST use the homenet's ULA.

It is possible that local services may number themselves using more than one of the prefixes advertised locally. Homenet hybrid proxies MUST filter out global IP addresses, providing only ULA addresses, similar to the process described in section 5.5.2 of [7]. [xxx is this going to be a problem?]

The Hybrid Proxy model relies on each link having its own name. However, homenets do not actually have a way to name local links that

will make any sense to the end user. Consequently, this mechanism will not work. In order to paper over this, some changes are required:

- The Hybrid Proxy function is divided into two: relaying proxies, and aggregating proxies. There must be exactly one querying proxy per link; there can be as few as one aggregating proxy per homenet.

- Relaying proxies do no translation, for example from ".local" to "bldg1.example.com" as shown in section 5.3 of [7]. They simply take queries over the DNS protocol for names in subdomains of '.local', the link-specific 'ip6.addr', and the link-specific 'in-addr.arpa' zones, and respond with the exact answers received.

- There must be exactly one querying proxy per internal link on the homenet; for links that are connected to more than one homenet router, HNCP is used to choose which router will provide the service.

- Querying proxies perform translation. Machine readable names are presented as subdomains of the TBD1 domain. Human readable names are presented as subdomains of the _hr.TBD1 domain.

- Every homenet router can provide a querying proxy, or only one router can. This is determined by HNCP; all homenet routers must provide this capability, but some homenet routers may provide enhanced querying proxy capabilities such that homenet routers providing only those capabilities described in this document must be disabled. Therefore, all homenet routers must be able to act as a querying proxy, or forward DNS queries to a central querying proxy, according to what is specified through HNCP.

3.4. DNSSEC Validation

DNSSEC Validation for the TBD1 zone and for the locally-served 'ip6.arpa and 'in-adr.arpa' domains is not possible without a trust anchor. Establishment of a trust anchor for such validation is out of scope for this document.

3.5. Support for Multiple Provisioning Domains

Homenets must support the Multiple Provisioning Domain Architecture [6]. In order to support this architecture, each homenet router that provides name resolution must provide one resolver for each provisioning domain (PvD). Each homenet router will advertise one resolver IP address for each PvD. DNS requests to the resolver associated with a particular PvD, e.g. using RA options [8] will be

resolved using the external resolver(s) provisioned by the service provider responsible for that PvD.

The homenet is a separate provisioning domain from any of the service providers. The global name of the homenet can be used as a provisioning domain identifier, if one is configured. Homenets should allow the name of the local provisioning domain to be configured; otherwise by default it should be "Home Network xxx", where xxx is the generated portion of the homenet's ULA prefix, represented as a base64 string.

The resolver for the homenet PvD is offered as the primary resolver in RAs and through DHCPv4 and DHCPv6. When queries are made to the homenet-PvD-specific resolver for names that are not local to the homenet, the resolver will use a round-robin technique, alternating between service providers with each step in the round-robin process, and then also between external resolvers at a particular service provider if a service provider provides more than one. The round-robining should be done in such a way that no service provider is preferred, so if service provider A provides one caching resolver (A), and service provider B provides two (B1, B2), the round robin order will be (A, B1, A, B2), not (A, B1, B2).

Every resolver provided by the homenet, regardless of which provisioning domain it is intended to serve, will accept updates for subdomains of the TBD1 and locally-served 'ip6.arpa' and 'in-addr.arpa' domains from hosts on the local link.

3.6. Using the Local Namespace While Away From Home

This architecture does not provide a way for service discovery to be performed on the homenet by devices that are not directly connected to a link that is part of the homenet.

4. Management Considerations

This architecture is intended to be self-healing, and should not require management. That said, a great deal of debugging and management can be done simply using the DNS service discovery protocol.

5. Privacy Considerations

Privacy is somewhat protected in the sense that names published on the homenet are only visible to devices connected to the homenet. This may be insufficient privacy in some cases.

The privacy of host information on the local net is left to hosts. Various mechanisms are available to hosts to ensure that tracking does not occur if it is not desired. However, devices that need to have special permission to manage the homenet will inevitably reveal something about themselves when doing so. It may be possible to use something like HTTP token binding[9] to mitigate this risk.

6. Security Considerations

There are some clear issues with the security model described in this document, which will be documented in a future version of this section. A full analysis of the avenues of attack for the security model presented here have not yet been done, and must be done before the document is published.

7. IANA considerations

This document is relying on the allocation of [TBD1] described in Special Use Top Level Domain '.homenet' [10]. As such, no new actions are required by IANA, but this document can't proceed until that allocation is done. At that time, the name [TBD1] can be substituted for the name that is eventually allocated during the processing of that document.

8. Normative References

[1] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <http://www.rfc-editor.org/info/rfc1034>.

[2] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

[3] Vixie, P., Ed., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, DOI 10.17487/RFC2136, April 1997, <http://www.rfc-editor.org/info/rfc2136>.

[4] Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762, DOI 10.17487/RFC6762, February 2013, <http://www.rfc-editor.org/info/rfc6762>.

[5] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <http://www.rfc-editor.org/info/rfc6763>.

[6] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <http://www.rfc-editor.org/info/rfc7556>.

[7] Cheshire, S., "Hybrid Unicast/Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-05 (work in progress), November 2016.

[8] Korhonen, J., Krishnan, S., and S. Gundavelli, "Support for multiple provisioning domains in IPv6 Neighbor Discovery Protocol", draft-ietf-mif-mpvd-ndp-support-03 (work in progress), February 2016.

[9] Popov, A., Nystrom, M., Balfanz, D., Langley, A., and J. Hodges, "Token Binding over HTTP", draft-ietf-tokbind-https-08 (work in progress), February 2017.

[10] Pfister, P. and T. Lemon, "Special Use Top Level Domain '.homenet'", draft-ietf-homenet-dot-03 (work in progress), March 2017.

Authors' Addresses

Ted Lemon
Nominum, Inc.
800 Bridge Parkway
Redwood City, California 94065
United States of America

Phone: +1 650 381 6000
Email: ted.lemon@nominum.com

Daniel Migault
Ericsson
8400 boulevard Decarie
Montreal, QC H4P 2N2
Canada

Email: daniel.migault@ericsson.com