Abstract

This document provides a framework for evaluating and comparing solutions for privacy-respecting discovery mechanisms.

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 https://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 May 17, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
1. Introduction

When AppleTalk was introduced in 1986, privacy concerns were not foremost in most people’s minds. The fact that a printer was offering printing service was not considered a secret, and the fact that a computer was seeking printing service was not considered a secret. The fact that the computer could discover the printer without expert configuration was considered remarkable.

Thirty years later, the landscape has changed. We now have many more network service types, and mobile wireless devices offering and consuming those services are common. Those mobile wireless devices and the services they offer or use often involve sensitive financial or medical data. Furthermore, the ubiquity of such mobile wireless devices makes them an attractive target for mischievous or outright criminal activity. The fact that a person’s smartphone is communicating with their implanted glucose monitor or insulin pump is not something that should be public information.

Hence there is now a need for discovery mechanisms that utilize privacy-preserving techniques. There have been various different efforts to address this, but they tend to offer solutions based on assumptions of what privacy aspects are important, without articulating what those assumptions are. Without knowing the assumptions and design goals of a particular proposal it is hard to evaluate whether that proposal meets those goals, or indeed whether they are the right goals.

Without advocating for any particular solution, this document presents an overview of the various aspects of device discovery and service discovery, and outlines the privacy concerns of each. Any given proposal may not address all possible privacy concerns. Depending on the scenario, it may not be necessary to address every privacy concern. Indeed, it may turn out to be impossible, or at least impractical, to address all possible privacy concerns. This document provides a framework to help evaluate whether a given solution meets the privacy needs of some particular usage scenario.
2. Discovery Operations

Device discovery and service discovery involve three principal operations:

1. Offer
2. Discover
3. Use

The "Offer" operation is how a device offers a service on the network. Typically this involves, using today’s terminology, (a) a "listening" UDP or TCP socket, which accepts incoming packets or connections, and (b) a way of advertising to other local and remote devices what kind of service is being offered, its name, and other metadata including how to reach it. Observe that there are three levels of information in use here: (i) the type of service, (ii) the name of the particular instance of that type of service, and (iii) the operational details of how to connect to and make use of that particular instance.

The "Discover" operation is how a client device learns what service instances are being offered (by local devices, and/or remote devices, depending on the discovery mechanism being used). Typically a client device knows what kind of service it is seeking, and wants to discover named instances of that service. The "Discover" operation is linking information level (i) type of service, with information level (ii) names of specific instances offering that type of service. The "Discover" operation can be viewed as providing a little information (just the name) about many different instances. In terms of complexity and efficiency, it’s a 1 x n operation, getting one piece of information about n instances.

The "Use" operation is how a client device requests additional information (IP address(es), port number, and possibly other metadata), and then uses this information to communicate with the service instance and make use of the service it offers. The "Use" operation is linking information level (ii) specific instance name, with information level (iii) detailed information about that individual instance. The "Use" operation can be viewed as providing a lot of information about one particular instance. In terms of complexity and efficiency, it’s an m x 1 operation, getting m pieces of information about 1 instance, and then proceeding to use that instance.

All three operations, and the three levels of information they use, need to be considered from a privacy perspective.
Note that some discovery mechanisms conflate "Discover" and "Use" into a single operation. Instead of requesting a little information about a lot of instances, or a lot of information about a single instance, they are only able to request everything about everything. They replace a 1 x n operation and an m x 1 operation with a combined m x n operation, always requesting m pieces of information each about n different instances.
3. Trust Granularity

When we talk about entities trusting other entities, what entities are we talking about?

Are the entities physical devices, like a smartphone or laptop computer?

Are the entities human users? If a device like a laptop computer has multiple users, we should not assume that because one user is authorized to discover certain services that means that all other users of that laptop are also authorized to discover those services.

Are the entities software applications? If a device like a smartphone has multiple apps installed, we should not assume that because one app is authorized to discover certain services that means that all other apps on that smartphone are also authorized to discover those services. For example, just because a medical app on a smartphone is authorized to discover and communicate with the user’s medical devices such as an implanted insulin monitor, that doesn’t mean that social network apps or games on that same smartphone are also authorized to discover and communicate with those medical devices.

Note that when the text above talks about a user or app being "authorized" we’re not talking about authorization controls being enforced by the laptop or smartphone. Controls enforced by the laptop or smartphone operating system are appropriate and have their place, but the kind of authorization controls we’re talking about here are enforced by the entity being discovered. When the entity being discovered receives a query from an authorized source, it answers the query. When the entity being discovered receives a query from an unauthorized source, it does not answer the query. The important question is the granularity of the "source" referred to — is it a physical device, a user, or an app? (This analysis presupposes that the host operating system on the device has sufficient memory protection and access controls to protect one user's secret key material from being accessed and abused by another user, or one app's secret key material from being accessed and abused by another app. For a device without such protection, only the per-device granularity of trust is applicable.)
4. Desirable Security Properties

For each of the operations and information levels described above, we need to consider what threats we are concerned about.

Authenticity & Integrity
Can we trust the information we receive? Has it been modified in flight by an adversary? Do we trust the source of the information?

Confidentiality
Who can read the information sent in messages? Ideally this should only be the appropriate trusted parties, but it can be hard to define who "the appropriate trusted parties" are. The "Discover" operation in particular is often used to discover new entities that the device did not previously know about. It may be tricky to work out how a device can have an established trust relationship with a new entity it has never previously communicated with.

Anonymity
Does the information exchange reveal the identity of either participant? In this context "identity" can mean things like the name, email address, or phone number of the human user. It could mean things like the hostname or MAC address of the device. Even when information is authenticated and confidential, there can be unexpected sources of information leakage. For example, if suitable precautions are not taken, the source MAC address in data packets can reveal the identity of the device manufacturer, which can yield clues about the nature of the device.

Resistance to Dictionary Attacks
It can be tempting to use simple one-way hash functions to obscure sensitive identifiers. This transforms a sensitive unique identifier such as an email address into a scrambled (but still unique) identifier. Unfortunately simple solutions may be vulnerable to offline dictionary attacks. Given a scrambled unique identifier, it may be possible to do a brute-force attack, trying billions of known and speculative email addresses until a match is found.

Resistance to Tracking
In today’s world, we have to be sensitive to any unchanging unique identifier, no matter how thoroughly and irreversibly scrambled it may be. Even though an attacker may not be able to divine the origin of a scrambled unique identifier, the unchanging unique identifier may still be correlated with other things. If a given unchanging unique identifier appears on a cafe network every
morning when a certain person comes in to get coffee, then with some certainty that unchanging unique identifier can be associated with that person, and used to track their movements around the city for the rest of their workday. Consequently, in cases where this threat is a concern, all cleartext identifiers used on the network need to be rotated according to some policy, so that a given identifier is not reused for too long or in different locations. These changing identifiers can be decoded by trusted entities, but are meaningless to anyone else.

Resistance to Message Linking
Is it possible to link or correlate exchanges across discovery operations? For example, do Discovery messages reveal information about future Use messages, or vice versa? This can be done via sender MAC address, for example. An adversary can use linkability information to de-anonymize service users or providers, even in the event that, individually, no information leaks from any particular message alone (e.g., because it’s encrypted in transit). For example, even if persistent identifiers are rotated periodically, if all identifiers are not rotated in unison then the overlap period can be used to track the user across identifier rotations.

Resistance to Denial-of-Service Attack
In any protocol where the receiver of messages has to perform cryptographic operations on those messages, there is a risk of a brute-force flooding attack causing the receiver to expend excessive amounts of CPU time (and battery power) just processing and discarding those messages.
5. Other Operational Requirements

5.1. Power Management

Many modern devices, especially battery-powered devices, use power management techniques to conserve energy. One such technique is for a device to transfer information about itself to a proxy, which will act on behalf of the device for some functions, while the device itself goes to sleep to reduce power consumption. When the proxy determines that some action is required which only the device itself can perform, the proxy may have some way (such as Ethernet "Magic Packet") to wake the device.

In many cases, the device may not trust the network proxy sufficiently to share all its confidential key material with the proxy. This poses challenges for combining private discovery that relies on per-query cryptographic operations, with energy-saving techniques that rely on having (somewhat untrusted) network proxies answer queries on behalf of sleeping devices.

5.2. Protocol Efficiency

Creating a discovery protocol that has the desired security properties may result in a design that is not efficient. To perform the necessary operations the protocol may need to send and receive a large number of network packets. This may consume an unreasonable amount of network capacity (particularly problematic when it’s shared wireless spectrum), cause an unnecessary level of power consumption (particularly problematic on battery devices) and may result in the discovery process being slow.

It is a difficult challenge to design a discovery protocol that has the property of obscuring the details of what it is doing from unauthorized observers, while also managing to do that quickly and efficiently.

5.3. Secure Initialization

One of the challenges implicit in the preceding discussions is that whenever we discuss "trusted entities" versus "untrusted entities", there needs to be some way that trust is initially established, to convert an "untrusted entity" into a "trusted entity".

One way to establish trust between two entities is to trust a third party to make that determination for us. For example, the X.509 certificates used by TLS and HTTPS web browsing are based on the model of trusting a third party to tell us who to trust. There are some difficulties in using this model for establishing trust for
service discovery uses. If we want to print our tax returns or medical documents on "our" printer, then we need to know which printer on the network we can trust to be "our" printer. All of the printers we discover on the network may be legitimate printers made by legitimate printer manufacturers, but not all of them are "our" printer. A third-party certificate authority cannot tell us which one of the printers is ours.

Another common way to establish a trust relationship is Trust On First Use (TOFU), as used by ssh. The first usage is a Leap Of Faith, but after that public keys are exchanged and at least we can confirm that subsequent communications are with the same entity. In today’s world, where there may be attackers present even at that first use, it would be preferable to be able to establish a trust relationship without requiring an initial Leap Of Faith.

Techniques now exist for securely establishing a trust relationship without requiring an initial Leap Of Faith. Trust can be established securely using a short passphrase or PIN with cryptographic algorithms such as Secure Remote Password (SRP) [RFC5054] or a Password Authenticated Key Exchange like J-PAKE [RFC8236] using a Schnorr Non-interactive Zero-Knowledge Proof [RFC8235].

Such techniques require a user to enter the correct passphrase or PIN in order for the cryptographic algorithms to establish working communication. This avoids the human tendency to simply press the "OK" button when asked if they want to do something on their electronic device. It removes the human fallibility element from the equation, and avoids the human users inadvertently sabotaging their own security.

Using these techniques, if a user tries to print their tax return on a printer they’ve never used before (even though the name looks right) they’ll be prompted to enter a pairing PIN, and the user *cannot* ignore that warning. They can’t just press an "OK" button. They have to walk to the printer and read the displayed PIN and enter it. And if the intended printer is not displaying a pairing PIN, or is displaying a different pairing PIN, that means the user may be being spoofed, and the connection will not succeed, and the failure will not reveal any secret information to the attacker. As much as the human desires to "just give me an OK button to make it print" (and the attacker desires them to click that OK button too) the cryptographic algorithms do not give the user the ability to opt out of the security, and consequently do not give the attacker any way to persuade the user to opt out of the security protections.
6. Informative References


Author’s Address

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

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

Over the course of several years, a rich collection of technologies has developed around DNS-Based Service Discovery, described across multiple documents. This "Road Map" document gives an overview of how these related but separate technologies (and their documents) fit together, to facilitate service discovery in various environments.

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 https://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 19, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
1. Road Map

DNS-Based Service Discovery [RFC6763] is a component of Zero Configuration Networking [RFC6760] [ZC].

Over the course of several years, a rich collection of technologies has developed around DNS-Based Service Discovery. These various related but separate technologies are described across multiple documents. This "Road Map" document gives an overview of how these technologies (and their documents) fit together to facilitate service discovery across a broad range of operating environments, from small scale zero-configuration networks to large scale administered networks, from local area to wide area, and from low-speed wireless links in the kb/s range to high-speed wired links operating at multiple Gb/s.

Not all of the available components are necessary or appropriate in all scenarios. One goal of this "Road Map" document is to provide guidance about which components to use depending on the problem being solved.

2. Namespace of Service Types

The single most important concept in service discovery is the namespace specifying how different service types are identified. This is how a client communicates what it needs, and how a server communicates what it offers. For a client to discover a server, the client and server need to have a common language to describe what they need and what they offer. The need to use the same namespace of service types, otherwise they may actually speak the same application protocol over the air or on the wire, and may in fact be completely compatible, and yet may be unable to detect this because they are using different names to refer to the same actual service. Hence, having a consistent namespace of service types is the essential prerequisite for any useful service discovery.

IANA manages the registry of Service Types [RFC6335][STR]. This registry of Service Types can (and should) be used in any service discovery protocol as the vocabulary for describing *all* IP-based services, not only DNS-Based Service Discovery [RFC6763].

In this document we focus on the use of the IANA Service Type Registry [STR] in conjunction with DNS-Based Service Discovery, though that should not be taken in any way to imply any criticism of other service discovery protocols sharing the same namespace of service types. In different circumstances different Service Discovery protocols are appropriate.
For example, for service discovery of services potentially available via a Wi-Fi access point, prior to association with that Wi-Fi access point, when no IP link has yet been established, a service discovery protocol may use raw 802.11 frames, not necessarily IP, UDP, or DNS-formatted messages. For Service Discovery using peer-to-peer Wi-Fi technologies, without any Wi-Fi access point at all, it may also be preferable to use raw 802.11 frames instead of IP, UDP, or DNS-formatted messages. Service Discovery using IEEE 802.15.4 radios may use yet another over-the-air protocol. What is important is that they all share the same vocabulary to describe all IP-based services. Using the same service type vocabulary means that client and server software, using agnostic APIs to consume and offer services on the network, has a common language to identify those services, independent of the medium or the particular service discovery protocol in use on that medium. Just as TCP/IP runs on many different link layers, and the concept of using an IP address to identify a particular peer is consistent across many different link layers, the concept of using a name from the IANA Service Type Registry to identify a particular service type also needs to be consistent across all IP-supporting link layers.

Originally, the IANA Service Type Registry [RFC6335][STR] used the term "Service Name" rather than "Service Type". Later it became clear that this term could be ambiguous. For a given service instance on the network, there is the machine-visible name of the type of service it provides, and the human-visible name of the particular instance of that type of service. For clarity, this document and related specifications use the term "Service Type" to denote the machine-visible name of the type of service, and the term "Instance Name" to denote the human-visible name of a particular instance.
3. Service Discovery Operational Model

The original DNS-Based Service Discovery specifications [RFC6763] used the terms "register" (advertise a service), "browse" (discover service instances), and "resolve" (get IP address and port for a specific service instance). This terminology is reflective of the thinking at the time, which viewed service discovery as a new and separate step, added to existing networking code. For example, a server would first open a listening socket as it always had, and then "register" that listening socket with the service discovery engine. Similarly, a client would first "resolve" a service instance to an IP address and port, and then, having done that, "connect" to that IP address and port.

More recent thinking in this area [RFC8305] has come to the conclusion that it is preferable wherever possible to insulate application software from networking details like having to decide between IPv4 and IPv6, having to decide among multiple IP addresses of either or both address families, and having to decide among multiple available network interfaces. Consequently this document and related specifications adopt newer terminology as follows:

1. Offer
2. Enumerate
3. Use

The first step, "Offer", is when a server is offering a service using some application-layer protocol, on a listening TCP or UDP (or other transport protocol) port, and wishes to make that known to other devices. This encompasses both making a listening socket (or the equivalent concept in whatever underlying networking API is being used) and advertising the existence of that listening socket via a service discovery mechanism.

The second step, "Enumerate", is when a client device wishes to perform some action, but does not yet know which particular service instance will be used to perform that action. For example, when a user taps the "AirPrint" button on an iPhone or iPad, the iPhone or iPad knows that the user wishes to print, but not which particular printer to use. The desired "function" is known (IPP printing), but not the particular instance. In this case, the client device needs to enumerate the list of available service instances that are able to perform the desired task. In most cases this list of service instances is presented to a human user to choose from; in some cases it is software that examines the list of available service instances and determines the best one to use. This second step is the operation that was called "browsing" in the original specifications.
The third step, "Use", is when particular service instance has been selected, and the client wants to make use of that service instance. This encompasses both the "resolve" step (finding IP address(es) and port(s) for the service instance) and the subsequent steps to establish communication with it, which may include details like address family selection, interface selection, transport protocol selection, etc. Ideally, application-layer code should never be exposed to IP addresses at all, just as application-layer code today is generally not exposed to details like MAC addresses [RFC8305].

The second and third steps are intentionally separate. In the second step, a limited amount of information (typically just the name) is requested about a large number of service instances. In the third step more detailed information (e.g., target host IP address, port number, etc.) is requested about one specific service instance. Requesting all the detailed information about all available service instances would be inefficient and wasteful on the network. If the information about services on the network is imagined as a table, then the second step is requesting just one column from that table (the name column) and the third step is requesting just one row from that table (the information pertaining to just one named service instance).

To give an example, clicking the "+" button in the printer settings on macOS is an operation performing the second step. It is requesting the names of all available printers. Once a desired printer has been chosen and configured, subsequent printing of documents is an operation performing the third step. It only needs to request information about the specific printer in question. It is not necessary to repeatedly discover the list of every printer on the network if the client device already knows which one it intends to use.

DNS-Based Service Discovery [RFC6763] implements these three principal service discovery operations using DNS records and queries, either using Multicast DNS [RFC6762] (for queries limited to the local link) or conventional unicast DNS [RFC1034] [RFC1035] (for queries beyond the local link).

Other service discovery protocol achieve the same semantics using different packet formats and mechanisms.

One incidental benefit of using DNS as the foundation layer for service discovery, in cases where that makes sense, is that both Multicast DNS and conventional unicast DNS are also used provide name resolution (mapping host names to IP addresses). There is some efficiency and code reuse gained by using the same underlying protocol for both service discovery and naming.
A final requirement is that the service discovery protocol perform
discovery not only at a single moment in time, but also ongoing
change notification (sometimes called "Publish & Subscribe").
Without support for ongoing change notification, clients would be
forced to resort to polling to keep data up to date, which is
inefficient and wasteful on the network.

Multicast DNS [RFC6762] implicitly includes change notification by
virtue of announcing record changes via IP Multicast, which allows
these changes to be seen by all peers on the same link (i.e., same
broadcast domain).

Conventional unicast DNS [RFC1034] [RFC1035] has historically not had
broad support for change notification. This capability is added via
the new mechanism for DNS Push Notifications [Push].

When using DNS-Based Service Discovery [RFC6763] there are two
aspects to consider: firstly how the clients choose what DNS names to
query, and what query mechanisms to use, and secondly how the
relevant information got into the DNS namespace in the first place,
so as to be available when clients query for it.

The available namespaces are discussed below in Section 4. Client
operation is discussed in Section 5 and server operation is discussed
in Section 6.
4. Service Discovery Namespace

When used with Multicast DNS [RFC6762] queries are automatically performed in the ".local" parent domain.

When used with conventional unicast DNS [RFC1034] [RFC1035] some other domain must be used.

For individuals and organizations with a globally-unique domain name registered to them, their globally-unique domain name, or a subdomain of it, can be used for service discovery.

However, it would be convenient for capable service discovery to be available even to people who haven’t taken the step of registering and paying for a globally-unique domain name. For these people it would be useful if devices arrived preconfigured with some suitable factory-default service discovery domain, such as "services.home.arpa" [I-D.ietf-homenet-dot]. Services published in this factory-default service discovery domain would not be globally unique or globally resolvable, but they could have scope larger than the single link provided by Multicast DNS.
5. Client Configuration and Operation

When using DNS-Based Service Discovery [RFC6763], clients have to choose what DNS names to query.

When used with Multicast DNS [RFC6762] queries are automatically performed in the ".local" parent domain.

For discovery beyond the local link, a unicast DNS domain must be used. This unicast DNS domain can be configured manually by the user, or it can be learned dynamically from the network (as has been done for many years at IETF meetings to facilitate discovery of the IETF Terminal Room printer, from outside the IETF Terminal Room). In the DNS-SD specification [RFC6763] section 11, "Discovery of Browsing and Registration Domains (Domain Enumeration)", describes how a client device learns one or more recommended service discovery domains from the network, using the special "lb._dns-sd._udp" query. All of the details from that specification are not repeated here. A walk-through describing one real-world example of how this works, using discovery of the IETF Terminal Room printer as a specific concrete case study, is given in Appendix A.

Given the service type that the user or client device is seeking (see Section 2) and one or more service discovery domains to look in, the client then sends its DNS queries, and processes the responses.

For some uses, one-shot conventional DNS queries and responses are perfectly adequate, but for service discovery, where a list may be displayed on a screen for a user to see, it is desirable to keep that list up to date without the user having to repeatedly tap a "refresh" button, and without the software repeatedly polling the network on the user’s behalf.

And early solution to provide asynchronous change notifications for unicast DNS was the UDP-based protocol DNS Long-Lived Queries [DNS-LLQ]. This was used, among other things, by Apple’s Back to My Mac Service [RFC6281] introduced in Mac OS X 10.5 Leopard in 2007.

Recent experience has shown that an asynchronous change notification protocol built on TCP would be preferable, so the IETF is now developing DNS Push Notifications [Push].

Because DNS Push Notifications is built on top of a DNS TCP connection, DNS Push Notifications adopts the conventions specified by DNS Stateful Operations [DSO] rather than inventing its own session management mechanisms.
6. Server Configuration and Operation

Section 5 above describes how clients perform their queries. The related question is how the relevant information got into the DNS namespace in the first place, so as to be available when clients query for it.

One way that relevant service discovery information can get into the DNS namespace is simply via manual configuration, creating the necessary PTR, SRV and TXT records [RFC6763], and indeed this is how the IETF Terminal Room printer has been advertised to IETF meeting attendees for many years. While this is easy for the experienced network operators at the IETF, it can be onerous to others less familiar with how to set up DNS-SD records.

Hence it would be convenient to automate this process of populating the DNS namespace with relevant service discovery information. Two efforts are underway to address this need, the Service Discovery Proxy [DisProx] (see Section 6.1) and the Service Registration Protocol [RegProt] (see Section 6.4).

6.1. Service Discovery Proxy

The first effort in the direction of automatically populating the DNS namespace is the Service Discovery Proxy [DisProx]. This technology is designed to work with today’s existing devices that advertise services using Multicast DNS only (such as almost all network printers sold in the last decade). A Service Discovery Proxy is a device colocated on the same link as the devices we wish to be able to discover from afar. A remote client sends unicast queries to the Discovery Proxy, which performs local Multicast DNS queries on behalf of the remote client, and then sends back the answers it discovers.

Because the time it takes to receive Multicast DNS responses is uncertain, this mechanism benefits from being able to deliver asynchronous change notifications as new answers come in, using DNS Long-Lived Queries [DNS-LLQ] or the newer DNS Push Notifications [Push] on top of DNS Stateful Operations [DSO].
6.2. Multicast DNS Discovery Relay

As an alternative to having to be physically connected to the desired network link, a Service Discovery Proxy [DisProx] can use a Multicast DNS Discovery Relay [Relay] to give it a ‘virtual’ presence on a remote link. Indeed, when using Discovery Relays, a single Discovery Proxy can have a ‘virtual’ presence on hundreds of remote links. A single Discovery Proxy in the data center can serve the needs of an entire enterprise. This is modeled after the DHCP protocol. In simple residential scenarios the DHCP server resides in the home gateway, which is physically attached to the (single) local link. In complex enterprise networks, it is common to have a single centralized DHCP server, which resides in the data center and communicates with a multitude of simple lightweight BOOTP relay agents, implemented in the routers on each physical link.

6.3. Service Discovery Broker

Finally, when clients are making TCP connections to multiple Service Discovery Proxies at the same time, this can be burdensome for the clients (which may be mobile and battery powered) and for the Service Discovery Proxies (which may have to serve hundreds of clients). This situation is remedied by use of a Service Discovery Broker [Broker]. A Service Discovery Broker is an intermediary between client and server. A client can issue a single query to the Service Discovery Broker and have the Service Discovery Broker do the hard work of issuing multiple queries on behalf of the client. And a Service Discovery Broker can shield a Service Discovery Proxy from excessive load by collapsing multiple duplicate queries from different client down to a single query to the Service Discovery Proxy.
6.4. Service Registration Protocol

The second effort in the direction of automatically populating the DNS namespace is the Service Registration Protocol [RegProt]. This technology is designed to enable future devices that will explicitly cooperate with the network infrastructure to advertise their services.

The Service Registration Protocol is effectively DNS Update, with some minor additions.

One addition is the introduction of a lifetime on DNS Updates, using the Dynamic DNS Update Lease EDNS(0) option [DNS-UL]. This option has similar semantics to a DHCP address lease, where a device is granted an address with with a certain lease lifetime, and if the device fails to renew the lease before it expires then the address will be reclaimed and become available to be allocated to a different device. In cases where DHCP is being used, a device will generally request a DNS Update Lease with the same expiration time as its DHCP address lease. This way, if the device is abruptly disconnected from the network, around the same time as its address gets reclaimed its DNS records will also be garbage collected.

The second addition is the introduction of information that tells the Service Registration server that the device will be going to sleep to save power, combined with information specifying how to wake it up again on demand, using the EDNS(0) OWNER Option [Owner].

The use of an explicit Service Registration Protocol is beneficial in networks where multicast is expensive, inefficient, or outright blocked, such as many Wi-Fi networks. An explicit Service Registration Protocol is also beneficial in networks where multicast and broadcast are supported poorly, if at all, such as mesh networks like those using IEEE 802.15.4.

The use of power management information in the Service Registration messages allows devices to sleep to save power, which is especially beneficial for battery-powered devices in the home.

7. Security Considerations

As an informational document, this document introduces no new Security Considerations of its own. The various referenced documents each describe their own relevant Security Considerations as appropriate.
8. Informative References


Appendix A. IETF Terminal Room Printer Discovery Walk-through

For about a decade now, the capable IETF network staff have provided off-link DNS Service Discovery for the Terminal Room printer at IETF meetings three times a year. In the case of the IETF meetings the necessary DNS records are entered manually, whereas this document advocates for increased automation of that task, but the process by which clients query to discover services is the same either way.

This appendix gives a detailed step-by-step account of how this works. It starts with joining the Wi-Fi network and doing a DHCP request, and ends with paper coming out of the printer. The reason the explanation is so detailed is to avoid inadvertently having a hand-waving "and then a miracle occurs" part, which skips over important details. And one of the reasons for asking the IETF network team to set this up for IETF meetings is that operational use is an important reality check. When standing in front of a room, giving a presentation, if you miss out some vital step, people may not notice. When running an actual service used by actual people, if you miss out some vital step, no paper comes out of the printer, and everyone notices.

Using a macOS computer, at an IETF meeting, you can repeat the steps illustrated here to see exactly how it works. Or you can simply press Cmd-P in any application and see that "term-printer" appears as an available printer, to confirm that it does in fact work.

First, let’s see what the macOS computer learned from the local DHCP server:

% scutil
> list
...
subKey [74] = State:/Network/Service/21B5304C...54B28F4CA1D2/DHCP
...

> show State:/Network/Service/21B5304C...54B28F4CA1D2/DHCP
<dictionary> {
  Option_15 : <data> 0x6d656574696e672e696574662e6f7267
  ...
}

Option_15 is Domain Name. To see what domain name, we need to decode the hexadecimal data to ASCII.

% echo 6d656574696e672e696574662e6f7267 OA | xxd -r -p
meeting.ietf.org
Our DHCP domain name is meeting.ietf.org. Does meeting.ietf.org recommend that we look in any Wide Area Service Discovery domains?

% dig lb._dns-sd._udp.meeting.ietf.org. ptr

; <<>> DiG 9.6-ESV-R4-P3 <<>> lb._dns-sd._udp.meeting.ietf.org. ptr
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35624
;; flags: qr aa rd ra
-query: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 4

;; QUESTION SECTION:
;lb._dns-sd._udp.meeting.ietf.org. IN PTR

;; ANSWER SECTION:
lb._dns-sd._udp.meeting.ietf.org. 3600 IN PTR meeting.ietf.org.

...

;; Query time: 8 msec
;; SERVER: 130.129.5.6#53(130.129.5.6)
;; WHEN: Wed Mar 13 10:16:40 2013
;; MSG SIZE  rcvd: 188

In the middle there you’ll see that the answer is "meeting.ietf.org". In this case the answer is self-referential -- "meeting.ietf.org" is inviting us to look for services in "meeting.ietf.org", but the PTR record(s) could equally well point at any other domain, such as "services.ietf.org", or anything else.
Note that this answer does not depend on the client device being "on" the IETF meeting network, which is in any case a loosely defined concept at best. Nor does it depend on sending the DNS query to a DNS server that is "on" the IETF meeting network. Any capable DNS recursive resolver anywhere on the planet will give the same answer. We can test this by sending the same DNS query to Google’s 8.8.8.8 public resolver:

```bash
% dig @8.8.8.8 lb._dns-sd._udp.meeting.ietf.org. ptr

; <<>> DiG 9.6-ESV-R4-P3 <<>>
@8.8.8.8 lb._dns-sd._udp.meeting.ietf.org. ptr
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>>HEADER<<- opcode: QUERY, status: NOERROR, id: 24571
;; flags: qr rd ra; QUERY:1, ANSWER:1, AUTHORITY:0, ADDITIONAL:0

;; QUESTION SECTION:
;lb._dns-sd._udp.meeting.ietf.org. IN PTR

;; ANSWER SECTION:
lb._dns-sd._udp.meeting.ietf.org. 1532 IN PTR meeting.ietf.org.

;; Query time: 21 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; MSG SIZE  rcvd: 64
```

In the middle there you’ll see that the answer is still "meeting.ietf.org".
In this example, this particular test was done at the 86th IETF in Orlando, Florida, in March 2013. The Google 8.8.8.8 public resolver still gave the correct answer, even though it was 13 hops away:

```
% traceroute -q 1 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 64 hops max, 52 byte packets
  1 rtra (130.129.80.2)  1.369 ms
  2 75-112-170-148.net.bhntampa.com (75.112.170.148)  14.494 ms
  3 bun2.tamp20-car1.bhn.net (71.44.3.73)  19.558 ms
  4 hun0-0-0-0-tamp20-cbr1.bhn.net (72.31.117.156)  20.730 ms
  5 xe-8-2-0.bar1.tampal.level3.net (4.53.172.9)  13.052 ms
  6 ae-5-5.ebr1.miami1.level3.net (4.69.148.213)  27.413 ms
  7 ae-1-51.edge1.miami2.level3.net (4.69.138.75)  15.552 ms
  8 google-inc.edge1.miami2.level3.net (4.59.240.26)  40.852 ms
  9 209.85.253.118 (209.85.253.118)  21.118 ms
 10 216.239.48.192 (216.239.48.192)  21.890 ms
 11 216.239.48.192 (216.239.48.192)  23.221 ms
 12 *
 13 google-public-dns-a.google.com (8.8.8.8)  32.961 ms
```

For the rest of this example we use the Google 8.8.8.8 public resolver for all the queries.

In the case of IETF meetings the PTR is self-referential -- meeting.ietf.org is advising us to look in meeting.ietf.org, but it could easily be set up to direct us elsewhere. However, since it’s suggesting we look for services in meeting.ietf.org, we’ll do that.

A macOS computer with appropriate printer drivers installed will look for instances of the "_pdl-datastream._tcp" service type at "meeting.ietf.org"

```
% dig +short @8.8.8.8 _pdl-datastream._tcp.meeting.ietf.org. ptr
term-printer._pdl-datastream._tcp.meeting.ietf.org.
```

There’s one printing service available here, called "term-printer". That’s what you see when you press the "+" button in the Print & Fax Preference Pane on macOS.
When the user actually prints something, macOS does these queries:

```bash
% dig +short @8.8.8.8 \
term-printer._pdl-datastream._tcp.meeting.ietf.org. srv 0 0 9100 term-printer.meeting.ietf.org.

% dig +short @8.8.8.8 term-printer.meeting.ietf.org. AAAA 2001:df8::48:200:74ff:fee0:6cf8
```

This tells the computer that to use this printer, it must connect to [2001:df8::48:200:74ff:fee0:6cf8]:9100, using the installed printer driver, which speaks the appropriate vendor-specific printing protocol for that printer.

Printing from an iPhone or iPad is similar, except there are no vendor-specific printer drivers installed. Instead, printing from an iPhone or iPad uses the IETF Standard IPP printing protocol, using an IPP printer that supports at least URF (Universal Raster Format):

```bash
% dig +short @8.8.8.8 _universal._sub._ipp._tcp.meeting.ietf.org. ptr 
term-printer._ipp._tcp.meeting.ietf.org.
```

An iPhone or iPad will discover that there’s one IPP-based printing service available here, called "term-printer". It has the same name as the pdl-datastream printing service, and exists on the same physical hardware, but uses a different printing protocol.

When the user prints from their iPhone or iPad using AirPrint, iOS does these queries:

```bash
% dig +short @8.8.8.8 term-printer._ipp._tcp.meeting.ietf.org. srv 0 0 631 term-printer.meeting.ietf.org.

% dig +short @8.8.8.8 term-printer.meeting.ietf.org. aaaa 2001:df8::48:200:74ff:fee0:6cf8
```

Note that the ".ipp._tcp" service has the same target hostname and IPv6 address as the ".pdl-datastream" service, but is accessed at a different TCP port on that hardware device.

To use this printer, the iPhone or iPad connects to [2001:df8::48:200:74ff:fee0:6cf8]:631, and uses IPP to print.
Author’s Address

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

Phone: +1 408 974 3207
Email: cheshire@apple.com
DNS-SD compatible service discovery in GRASP
draft-eckert-anima-grasp-dnssd-01

Abstract

DNS Service Discovery (DNS-SD) defines the common framework for applications to announce and discover services. This includes service names, service instance names, common parameters for selecting a service instance (weight, priority) as well as service specific parameters.

GRASP is intended to also be used for service discovery. Reinventing service discovery for GRASP with a similar set of features would result in duplication of work. Therefore, this document defines how to use GRASP to announce and discover services in a way that inherits DNS-SD features and also tries to be compatible in spirit as much as possible while still maintaining the intended simplicity of GRASP.

The goal of this document is to permit defining service and their parameters once and then use that in GRASP, mDNS and (unicast) DNS. Future work can also define DNS-SD <-> GRASP gateway functions.

In support of service discovery, this document also defines name discovery and schemes for reusable elements in GRASP objectives which are designed to be extensible so that future work that identifies elements required across multiple objectives do not need to define a scheme how to do this.

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 https://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."
1. Overview

DNS Service Discovery (DNS-SD) defines the common framework for applications to announce and discover services. This includes service names, service instance names, common parameters for
selecting a service instance (weight, priority) as well as service specific parameters.

GRASP is intended to also be used for service discovery. Reinventing service discovery for GRASP with a similar set of features would result in duplication of work. Therefore, this document defines how to use GRASP to announce and discover services in a way that inherits DNS-SD features and also tries to be compatible in spirit as much as possible while still maintaining the intended simplicity of GRASP.

The goal of this document is to permit defining service and their parameters once and then use that in GRASP, mDNS and (unicast) DNS. Future work can also define DNS-SD <-> GRASP gateway functions.

GRASP exists as so-called GRASP-Domains, which are networks across which GRASP is run. This document primarily defines how to perform service discovery across such a domain leveraging GRASPs options to perform unsolicited flooding of announcements or flooding of requests and finding the closest service instances. The initial use case of this document is to support what in DNS-SD is done via mDNS but in larger networks - GRASP-Domains. Beside the efficient flooding, GRASP provides reliability and security (depending on the so called substrate used by GRASP, such as the autonomic control plane - ACP). Providing compatibility with existing mDNS service announcer or clients is possible, but not described in this version of the document.

The encoding of information chosen in this document does not try to use GRASP solely as a transport layer, but to also leverage the CBOR structure of GRASP messages to natively encode the message elements required for services in a way that is most simple - instead of using GRASP only as e.g.: an encapsulation of otherwise unchanged DNS message encodings. This is done to minimize the amount of coding required (and not require any DNS code unless future gateway functions are required), to increase the simplicity, minimize the amount of data on the wire and allow easier extensibility. On the downside, the mechanisms provided here do not cover the whole slew of possible options of DNS/DNS-SD, but instead only those deemed to be required. Others can be added later.

In support of service discovery, this document also defines name discovery and schemes for reusable elements in GRASP objectives which are designed to be extensible so that future work that identifies elements required across multiple objectives do not need to define a scheme how to do this.
2. Specification (Normative)

2.1. Service and Name Objectives

Unsolicited, flooded announcements (M_FLOOD) in GRASP and solicited flooded discovery (M_DISCOVERY) operate on the unit of GRASP objective-names. Therefore a scheme is required to indicate services via objective-names. Note: future work may want to reuse the encodings related to services (defined below in this document) inside other (multicast or unicast only) objective exchanges, in which case the service names are not impacted.

When an objective is meant to be solely about a service name as defined and registered according to RFC6335, the objective MUST use an objective-name of SRV.<service-name>. This naming scheme allows to avoid creating duplicate and potentially inconsistent registration of names for those objectives vs. registrations done for example for DNS-SD. The primary use case for this naming scheme are therefore service names that are intended to be used in both DNS-SD and GRASP.

When an objective is meant announcement and discovery of a DNS compatible <name> such as "www-internal" in "www-internal.example.com", the objective SHOULD use an objective-name of NAME.<name>. See Section 2.3.3 for more details.

See Section 5 for the detailed IANA asks relating to these definitions.

2.2. Objective Value Reuseable Elements Structure

Because service discovery, as explained in the prior section, needs to utilize different objectives, it requires cross-objective standardized encoding of the elements of services. GRASP did not define standardized message elements for the message body (called "objective-value") of GRASP messages. Therefore, this document introduces such a feature.

[RFC-editor: please remove all occurrences of XXXX in rfcXXXX with the RFC number assigned to this document and remove this edit note.]
If an objective wants to use reusable elements, the objective-value MUST be a CBOR map and the reusable elements are found under the key "@rfcXXXX". Objectives that do not want reusable elements as defined here can use any objective-value format including a CBOR map, but they cannot use the "@rfcXXXX" key if they use a map. This approach was chosen as the hopefully least intrusive mechanism given how by nature all of "objective-value" is meant to be defined by individual objective definitions.

The value of "@rfcXXXX" is a map of reusable elements. Each relement has an IANA registered element-name and codepoint (see Section 5). The element-name is for documentation purposes only, CBOR encodings only use the numeric codepoint for encoding efficiency to minimize the risk for this solution to not be applicable to low-bitrate networks such as in IoT.

Format and semantic of the relement-value is determined by the specification of the reusable element as is the fact whether more than one instances of the same reusable element are permitted.

Reusable elements SHOULD be defined to be extensible. The methods used depend on the complexity of the element and the likely need to extend/modify the element with backward or non-backward compatible information. The following is a set of initial options to choose from:

Element values that are a map MUST permit and reserve key value 0 (numerical) for private extensions of the element defined by the individual objective.

Element values that are a map MUST NOT use bareword key values starting with a "_". These too are for private extensions defined by the individual objective.

Element values SHOULD be defined so that additional keys in maps and additional elements at the end of arrays can be ignored by prior versions of the definition. Whenever a newer definition is made for an element where this rule is violated, the element SHOULD be changed.
in a way for older version recipients to recognize that it is not compatible with it.

One method to indicate compatibility is a traditional version "<mayor>.<minor>". Within the same <mayor> version number, increasing <minor> version numbers must be backward compatible. Different <mayor> version numbers are not expected to be compatible with each other. If they are, then this can be indicated by including multiple version numbers.

A compressed form of version compatibility information is the use of a simple bitmask element where each bit indicates a version that the represented data is compatible with.

2.3. Reuseable Elements

2.3.1. Sender Loop Count

relement-codepoint //= ( &(sender-loop-count:1) => 1..255 )

Sender-loop-count is set by the sender of an objective message to the same value as the loop-count of the message. On receipt, distance = ( sender-loop-count - loop-count ) is the distance of the sender from the receiver in hops. This element can be used for informational purposes in M_FLOOD and M_DISCOVERY messages and may be required to be used in these messages by the specification of other elements (such as the service element described below). This element MUST occur at most once. If a receiver expects to use the distance but sender-loop-count was not announced, then distance SHOULD be assumed to be 255 by the receiver.

2.3.2. Service Element

The srv-element (service element) is a reusable element to request or announce a service instance or to request and list service instance names.
relement-codepoint // = ( &(!srv-element:2) => context-element )

context-element = {
    ?( &(!private:0) => any),
    ?( &(!msg-type:1 => msg-type),
    ?( &(!service:2) => tstr),
    *( &(!instance:3) => tstr),
    ?( &(!domain:4) => tstr),
    ?( &(!priority:5) => 0..65535 ),
    ?( &(!weight:6) => 0..65535 ),
    *( &(!kvpairs:7) => { *(tstr: any) },
    ?( &(!range:8) => 0..255 ),
    *( &(!clocator:9) => clocator),
}

clocator = [ context, locator-option ]
context = cstr
locator-option = ; from GRASP

msg-type = &{ describe: 0, describe-request:1, enumerate:2, enumerate-request:3 }

Service: A service name registered according to RFC6335. If it is not present, then objective-name MUST be SRV.<service-name> where <service-name> is the service-name.

Instance: The <Instance> of a DNS-SD Service Instance Name ( <Instance> . <Service> . <Domain>). It is optional, see Section 3.2.

Domain: The equivalent of the <Domain> field of a DNS-SD Service Instance Name. If domain is not present, this is equivalent to ".local" in DNS (as introduced by mDNS) and implies the unnamed "local" domain, which is the GRASP domain across which the message is transmitted.

Priority, Weight: Service Instance selection criteria as defined in RFC2782. If either one is not present, its value defaults to 0.

Kvpairs: Map of key/value pairs that are service parameters in the same format as the key/value pairs in TXT field(s) of DNS-SD TXT records as defined in RFC6763, section 6.3.

Range: Allows to flexibly combine distance and priority/weight based service selection according to the definition of distance in Section 2.3.1.
If min-distance is the distance of the closest service announcer, and min-range the range announced by it, then the recipient MUST consider the priority/weight of all service announcers that are not further away than (min-distance + min-range). If not included, range defaults to 255.

If range is announced, the sender-loop-count element MUST also be announced.

Clocator: The "contextual locator" allows to indicate zero or more locators for the indicated service instance. The context element indicates in which context the locator-option is to be resolved. The reserved context value of "" (empty string) indicates the GRASP domain used, aka: the "local" context in which the service announcement is made. The reserved context value of "0" indicates the default routing context of the announcing node. This is often called "global table", "VRF 0" or "default VRF" on nodes using the "VRF" abstraction. Any other value is a string specifying a context such as another VRF.

The mechanism by which originator and recipient of the srv-element agree on common naming for contexts is outside the scope of this specification. The context therefore allows to indicate locators both for the context through which the GRASP message distributed the srv-element (GRASP domain) as well as that for other contexts. Assume the GRASP domain is the ACP, then clocators in ACP would have a context of "", clocators in the global routing table (part of the data-plane) a context of "0", and clocators on other VRFs (also part of data-plane) a clocator that is their string name.

If no locators are indicated, then the locator of the service(s) is the optional locator-option of the GRASP message in which the objective is contained meant to be used for the service(s) indicated and the clocator implied is "".

If locator(s) are indicated, the messages location-option must be ignored for the service (but may be necessary to be present for other purposes of the objective).

Msg-type Type (aka: intention) of the srv-element. If not present, it is assumed to be "describe".

Describe: Describes one service instance. At least one clocator is required for a positive response, all other fields are permitted, but optional. "Describe" is used in M_FLOOD for unsolicited announcements of services (flooded), in M_RESPONSE messages for solicited announcements of a service and in M_NEGOTIATE for negotiated announcements (both unicast). If clocator is not
included, then all fields except service and instance (and msg-type and private) must not be included and the srv-element provides a negative reply: No information about this service/service instance. This is only permitted in unicast "describe" messages.

Describe-request: Request for a "describe" reply. It is used in M_DISCOVERY (flooded) for solicited discovery of services or in M_REQ_SYN (unicasted) for negotiated discovery of service instance(s). In "describe-request", only service is mandatory (but can be provided via the objective-name field of the message), and domain is optional. "Instance" is optional. If provided, then the recipient is asked to provide information about the named instance only. All other fields of srv-element are to be ignored by the receiver in this specification, but a semantic for setting them may be introduced in followup work, specifically to filter replies by the indicated fields.

"Describe-request" without instance MAY be answered by "Enumerate" (see below) if the responder has so many instances that it thinks the initiator should rather first select one or fewer instances and ask for their description. The sender of te "Describe-request" MUST be prepared to accept that answer and as necessary follow up with "Describe-request" with the instance names of interest.

Enumerate: Used in the same GRASP messages as "describe", but instead of providing information about one service instance, it is listing service instance names. The purpose of enumerate is the same as browsing a service in DNS-SD. It would be followed by some human or automated selection of one or more instances and then a "describe" M_REQ_SYN request for those instances sent to the source of the "enumerate" to learn about the locators and other parameters of the service instances.

In this specification, all fields other than service, instance and domain (and msg-type and private) must be unset in "enumerate".

Enumerate-request: Requests an "enumerate" reply. It is used in the same way as "Describe-request" except that instance would usually not be set (because in that case it is more useful to send a "Describe-request").

2.3.3. Name Element

The NAME,<name> elements is meant to provide basic name resolution comparable to mDNS name resolution for GRASP domains where this is
desirable and no better name resolution exist - for example in the ACP where there is no requirement for DNS.

Because the GRASP service lookup (unlike) DNS does not mandate that nodes have names (not even service instance names), the use of names is primarily meant to support legacy software. New designs should instead look up only services and service instance names, and nodes should announce their names as service instance names for the services they offer:

For example consider a GRASP (ACP) domain of "example.com". The node providing some "www" service could have a name "www-internal" which means GRASP objective NAME.www-internal, that objective value would include primarily the nodes IP address(es) and the port number for the www service would have to be guessed (80). Better, the node would announce GRASP objective SRV.www and the objective value would include the service instance name www-internal and the (TCP) port information (80 or a non-default port).

relement-codepoint //= ( &(name-element:3) => context-element )

countext-element //= {
  *( &name:10) => tstr),
}

ipv6-address-option = [O_IPv4_ADDRESS, ipv6-address]
ipv4-address-option = [O_IPv6_ADDRESS, ipv6-address]
locator-option /= ipv4-address-option
locator-option /= ipv6-address-option

Name information is carried in the name-element relement. It is a context-element like the one used for srv-element except that it adds the name component and that it does not permit the service and instance components and that it allows only describe and describe-request values in the msg-type. Clocators MUST use the ipv6-address-option or ipv4-address-option in the locator-option component.

TBD: Unclear if/how we should best formalize the differences in the context element permitted information between services and names. The above is quite informal.
Priority, weight, kvpairs, range (and of course private) MAY be used in describe messages to support multiple instances of the same name, as used for name anycast/prioritycast.

Nodes may have multiple names. These can be listed in the name component. If a nodes names have the notion of a primary name and secondary names then the primary name should be the first in the list of names. In DNS-SD, the name pointed to by CNAME RRs can be considered to be the primary name. A describe-request for a non-primary name SHOULD return in the list of names the requested name and the primary name.

Note that there is no reverse lookup defined in this version of the document (no lookup from IP address to name).

3. Explanations (Informative)

3.1. Using GRASP service announcements

TBD: This section contains a range of details that should become normative in later versions.

This section provides a step by step walk-through of how to use GRASP service announcements and compares it to DNS-SD.

The most simple method to use GRASP service discovery is to select (and if still necessary, register) a <service-name> and start one or more agents (e.g.: ASAs) announcing their service instance(s) via GRASP. At minimum, an agent should periodically (default 60 seconds) announce the service instance via GRASP M_FLOOD messages as an objective SRV.<service-name> with a srv-element and a sender-loop-count element (default 255). The ttl of the GRASP message should be 3.5 times the announcement period, e.g.: 210000 msec.

Consumers of the service will use GRASP to learn of the service instances and select one. This approach is most similar to the use of DNS-SD with mDNS except that the scope of the announcement is a whole GRASP domain (such as the ACP) as opposed to a single IP subnet in mDNS and that mDNS primarily relies on request & reply but in its standard not on periodic unsolicited announcements. We describe here the unsolicited flooding option via M_FLOOD first because it is recommended for services with a dense population of service consumers and it is most simple to describe.

On the service announcer, the parameters priority, weight and range of the service instance can be selected from intent or configuration - or left at default. The default range 255 will result in selection of a random target of the service like in DNS-SD. Setting priority/
weight allows to prioritize and weigh the selection as in DNS-SD. Setting range to 0 allows to select the closest target, priority/weight are only compared between targets of the same shortest distance. Distance based options are not available in DNS-SD because it does not expect that network distance is available to arbitrary DNS-SD client. It is available to GRASP clients though. Using 0 < range < 255 allows for a hybrid priority/weight and distance based service selection (e.g.: Select the highest priority instance within a range of 5 hops).

If the service is a non-GRASP service, then the result of the service discovery has to be a transport locator to which the client can open a connection and talk the protocol implied by the service. This transport locator(s) have to be put into the clocator parameter. The context of the clocator would normally be "", aka: the transport locator is in the IP reachability associated with the GRASP domain (e.g.: IPv6 of the ACP for ACP GRASP domain).

If an ACP service is announced via ACP GRASP, then the locator(s) can be O_IPv6_LOCATOR or O_FQDN_LOCATOR. The O_IPv6_LOCATOR is used if the service is defined to be available via some transport layer port (TCP, UDP or other). The determination of the actual transport connection to be used is the same as in DNS-SD: If the transport protocol is not TCP or UDP, it has to be implied by the specification of <service-name> or can be detailed in kvpairs which carries the same information as DNS-TXT TXT RRs of the service. Alternatively, the transport-proto field of the locator can contain any valid IP protocol directly (TBD), which is not possible in DNS-SD.

Like DNS-SD, service discovery via GRASP does not require allocation and use of well-known ports for services. Unlike DNS-SD, there is no need in GRASP to define service instance names or target names. In DNS SD, PTR RRs resolve from a service name to a set of service instance named. SRV and TXT RRs resolve from service instance names to service instance parameters including the target. A target is the DNS host name of the service instance. It gets resolved via A/AAAA RRs to IPv4/IPv6 addresses of the targ. In GRASP service discovery, host names are not used. Service instance names are optional too. Service instance names are useful for human diagnostics and human selection of service instances. In fully automated environments, they can be are less important. For diagnostic purposes, it is recommended to give service instances service instance names in GRASP service announcements.

A locator with O_URI_LOCATOR type can be used in GRASP to indicate a URI for the transport method for a service instance. If the URI includes a host part, care must be taken to use only IP addresses in the host part if the context of the GRASP domain does not support
host name resolution - such as the ACP - or to use the GRASP name resolution mechanisms described elsewhere in this document. And that the addresses indicated are also reachable in the GRASP domain. For example, in service announcements across a DULL GRASP domain, only the IPv6 link-local addresses on that subnet must be used (this applies equally when using the _O.IPv6.LOCATOR).

Instead of using M_FLOOD to periodically announce service instances, M_DISCOVERY can be used to actively query for service instances. The msg-type type must then be "describe-request". Because no periodic flooding is necessary, this solution is more lightweight for the network when the number of requesting clients is small. Note though that the M_DISCOVERY will terminate as soon as a provider of the objective is found, so the service instances found will be based on distance and therefore selection of instance by priority and weight will not work equally well as with M_FLOOD. Consider for example a central service instance in the NOC that should always be used (for example for centralized operational diagnostics) unless the WAN connection is broken, in which case distributed backup service instances should be used. With the current logic of M_DISCOVERY this is not possible.

3.2. Further comparison with DNS-SD

Neither the GRASP SRV.* objective-name, the service name nor any other parameter explicitly indicate the second label "_tcp" or "_udp" of DNS-SD entries. DNS-SD, RFC6763 explains how this is an unnecessary, historic artefact.

This version of the document does not define an equivalent to "_sub" structuring of service enumeration.

This version of the document does not define mechanisms for reverse resolution of arbitrary services: An inquirer may unicast M_SYNC_REC to a node with a series of objectives with specific service names of interest and describe-request, but there is no indication of "ANY" service.

3.3. Open Issues

TBD: Examine limitations mentioned in "in this version of the text/document".

TBD: The GRASP specification does currently only permit TCP and UDP for the transport-proto element. This draft should expand the GRASP definitions to permit any valid IP protocol. We just need to decide whether this should only apply to the locator in the srv element or
also retroactive to the locator-option in GRASP messages (maybe not there?).

TBD: A fitting CBOR representation for a kvpair key without value needs to be specified so that it can be distinguished from an empty value as outlined in RFC6763 section 6.4.

TBD: In this version, every service/service-instance is an element by itself. Future versions of this document may add more encoding options to allow more compact encoding of recurring fields.

TBD: Is there a way in CDDL to formally define the string names of the relement-codepoint’s?

4. Security Considerations

TBD.

5. IANA Considerations

This document requests a new "GRASP Objective Value Standard Elements" table in the GRASP Parameter Registrar. The values in this table are names and a unique numerical value assigned to each name. Future values MUST be assigned using the RFC Required policy defined by [RFC8126]. The numerical value is simply to be assigned sequentially. The following initial values are assigned by this document:

sender-loop-count 1 [defined in rfcXXXX]

srv-element 2 [defined in rfcXXXX]

name-element 3 [defined in rfcXXXX]

This document updates the handling of the "GRASP Objective Names" Table introduced in the GRASP IANA considerations as follows:

Assignments for objective-names of the form "SRV.<text>" and "NAME.<text>" are special.

Assignment of "SRV.<text>" can only be requested if <text> is also a registered service-name according to RFC6335. The specification required for registration of a "GRASP Objective Name" MUST declare that the intended use of the objective name in GRASP is intended to be compatible with the intended use of the registered service name.

Registration of "SRV.<text>" in the "GRASP Objective Name" table is optional, but recommended for all new service-names that are meant to
be used with GRASP. Non-registration can for example happen with DNS-SD <-> GRASP gateways that inject pre-existing service-names into GRASP. Note that according to the GRASP RFC, registration is mandatory, so this exemption for "SRV.<text>" is also an update to that specification.

There MUST NOT be any assignment for objective names of the form "NAME.<text>". These names are simply used by GRASP nodes without registration (just like names in mDNS).

6. Acknowledgements

7. Change log [RFC Editor: Please remove]

7.1. 01 -

Only refreshing, no changes since -00.

7.2. 00 - Initial version

8. References

8.1. Normative References

[I-D.ietf-anima-grasp]


8.2. Informative References

[I-D.ietf-anima-autonomic-control-plane]

Author’s Address
DNS Stateful Operations
draft-ietf-dnsop-session-signal-14

Abstract

This document defines a new DNS OPCODE for DNS Stateful Operations (DSO). DSO messages communicate operations within persistent stateful sessions, using type-length-value (TLV) syntax. Three TLVs are defined that manage session timeouts, termination, and encryption padding, and a framework is defined for extensions to enable new stateful operations. This document updates RFC 1035 by adding a new DNS header opcode and result code which has different message semantics. This document updates RFC 7766 by redefining a session, providing new guidance on connection re-use, and providing a new mechanism for handling session idle timeouts.

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 February 3, 2019.
Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
Provisions Relating to IETF Documents
(http://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction .............................................. 3
2. Requirements Language .................................. 6
3. Terminology ............................................. 6
4. Discussion .............................................. 11
5. Applicability ........................................... 12
6. Protocol Details ....................................... 12
   6.1. DSO Session Establishment ....................... 12
      6.1.1. Connection Sharing .......................... 14
      6.1.2. Zero Round-Trip Operation ................. 15
      6.1.3. Middlebox Considerations .................. 16
   6.2. Message Format .................................. 17
      6.2.1. DNS Header Fields in DSO Messages .......... 18
      6.2.2. DSO Data .................................. 20
      6.2.3. EDNS(0) and TSIG .......................... 25
   6.3. Message Handling ............................... 26
      6.3.1. Error Responses ............................ 27
   6.4. Flow Control Considerations ................... 28
   6.5. Responder-Initiated Operation Cancellation ..... 28
7. DSO Session Lifecycle and Timers ................... 30
   7.1. DSO Session Initiation .......................... 30
   7.2. DSO Session Timeouts ........................... 30
   7.3. Inactive DSO Sessions .......................... 31
   7.4. The Inactivity Timeout .......................... 33
      7.4.1. Closing Inactive DSO Sessions ............. 33
      7.4.2. Values for the Inactivity Timeout .......... 34
   7.5. The Keepalive Interval ........................ 35
      7.5.1. Keepalive Interval Expiry ................ 35
      7.5.2. Values for the Keepalive Interval .......... 35
   7.6. Server-Initiated Session Termination .......... 37
      7.6.1. Server-Initiated Retry Delay Message ..... 38
8. Base TLVs for DNS Stateful Operations ............. 41
1. Introduction

This document specifies a mechanism for managing stateful DNS connections. DNS most commonly operates over a UDP transport, but can also operate over streaming transports; the original DNS RFC specifies DNS over TCP [RFC1035] and a profile for DNS over TLS [RFC7858] has been specified. These 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 (such as DNS Push Notifications [I-D.ietf-dnssd-push]).

The existing EDNS(0) Extension Mechanism for DNS [RFC6891] is explicitly defined to only have "per-message" semantics. While EDNS(0) has been used to signal at least one session-related parameter (edns-tcp-keepalive EDNS0 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. For example, a server cannot arbitrarily instruct a client to close a connection because the server can only send EDNS(0) options in responses to queries that contained EDNS(0) options.

This document defines a new DNS OPCODE, DSO ([TBA1], tentatively 6), for DNS Stateful Operations. DSO messages are used to communicate operations within persistent stateful sessions, expressed using type-
length-value (TLV) syntax. This document defines an initial set of three TLVs, used to manage session timeouts, termination, and encryption padding.

The three TLVs defined here are all mandatory for all implementations of DSO. Further TLVs may be defined in additional specifications.

DSO messages may or may not be acknowledged; this is signaled by providing a non-zero message ID for messages that must be acknowledged and a zero message ID for messages that are not to be acknowledged, and is also part of the definition of a particular message type. Messages are pipelined; answers may appear out of order when more than one answer is pending.

The format for DSO messages (Section 6.2) differs somewhat from the traditional DNS message format used for standard queries and responses. The standard twelve-byte header is used, but the four count fields (QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT) are set to zero and accordingly their corresponding sections are not present.

The actual data pertaining to DNS Stateful Operations (expressed in TLV syntax) is appended to the end of the DNS message header. The stream protocol carrying the DSO message frames it with 16-bit message length, so the length of the DSO data is determined from that length, rather than from any of the DNS header counts.

When displayed using packet analyzer tools that have not been updated to recognize the DSO format, this will result in the DSO data being displayed as unknown additional data after the end of the DNS message.

This new format has distinct advantages over an RR-based format because it is more explicit and more compact. Each TLV definition is specific to its use case, and as a result contains no redundant or overloaded fields. Importantly, it completely avoids conflating DNS Stateful Operations in any way with normal DNS operations or with existing EDNS(0)-based functionality. A goal of this approach is to avoid the operational issues that have befallen EDNS(0), particularly relating to middlebox behaviour (see for example [I-D.ietf-dnsop-no-response-issue] sections 3.2 and 4).

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 pseudo-RR. The specifications for each individual option need to define how each different option is to be acknowledged, if necessary.
In contrast to EDNS(0), with DSO there is no compelling motivation to pack multiple operations into a single message for efficiency reasons, because DSO always operates using a connection-oriented transport protocol. Each DSO operation is communicated in its own separate DNS message, and the transport protocol can take care of packing several DNS messages into a single IP packet if appropriate. For example, TCP can pack multiple small DNS messages into a single TCP segment. This simplification allows for clearer semantics. Each DSO request message communicates just one primary operation, and the RCODE in the corresponding response message indicates the success or failure of that operation.
2. 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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

"DSO" is used to mean DNS Stateful Operation.

The term "connection" means a bidirectional byte (or message) stream, where the bytes (or messages) are delivered reliably and in-order, such as provided by using DNS over TCP [RFC1035] [RFC7766] or DNS over TLS [RFC7858].

The unqualified 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.
- Either end of the connection may initiate messages to the other.

In this document the term "session" is used exclusively as described above. The term has no relationship to the "session layer" of the OSI "seven-layer model".

A "DSO Session" is established between two endpoints that acknowledge persistent DNS state via the exchange of DSO messages over the connection. This is distinct from a DNS-over-TCP session as described in the previous specification for DNS over TCP [RFC7766].

A "DSO Session" is terminated when the underlying connection is closed. The underlying connection can be closed in two ways:

Where this specification says, "close gracefully," that means sending a TLS close_notify (if TLS is in use) followed by a TCP FIN, or the equivalents for other protocols. Where this specification requires a connection to be closed gracefully, the requirement to initiate that graceful close is placed on the client, to place the burden of TCP’s TIME-WAIT state on the client rather than the server.

Where this specification says, "forcibly abort," that means sending a TCP RST, or the equivalent for other protocols. In the BSD Sockets API this is achieved by setting the SO_LINGER option to zero before closing the socket.
The term "server" means the software with a listening socket, awaiting incoming connection requests in the usual DNS sense.

The term "client" means the software which initiates a connection to the server's listening socket in the usual DNS sense.

The terms "initiator" and "responder" correspond respectively to the initial sender and subsequent receiver of a DSO request message or unacknowledged 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 DSO request message or unacknowledged message) or a responder (when sending a DSO response message).

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

In protocol implementation there are generally two kinds of errors that software writers have to deal with. The first is situations that arise due to factors in the environment, such as temporary loss of connectivity. While undesirable, these situations do not indicate a flaw in the software, and they are situations that software should generally be able to recover from. The second is situations that should never happen when communicating with a correctly-implemented peer. If they do happen, they indicate a serious flaw in the protocol implementation, beyond what it is reasonable to expect software to recover from. This document describes this latter form of error condition as a "fatal error" and specifies that an implementation encountering a fatal error condition "MUST forcibly abort the connection immediately". Given that these fatal error conditions signify defective software, and given that defective software is likely to remain defective for some time until it is fixed, after forcibly aborting a connection, a client SHOULD refrain from automatically reconnecting to that same service instance for at least one hour.

This document uses the term "same service instance" as follows:

- In cases where a server is specified or configured using an IP address and TCP port number, two different configurations are referring to the same service instance if they contain the same IP address and TCP port number.

- In cases where a server is specified or configured using a hostname and TCP port number, such as in the content of a DNS SRV record [RFC2782], two different configurations (or DNS SRV
records) are considered to be referring to the same service instance if they contain the same hostname (subject to the usual case insensitive DNS name matching rules [RFC1034] [RFC1035]) and TCP port number. In these cases, configurations with different hostnames are considered to be referring to different service instances, even if those different hostnames happen to be aliases, or happen to resolve to the same IP address(es). Implementations SHOULD NOT resolve hostnames and then perform matching of IP address(es) in order to evaluate whether two entities should be determined to be the "same service instance".

When an anycast service is configured on a particular IP address and port, it must be the case that although there is more than one physical server responding on that IP address, each such server can be treated as equivalent. What we mean by "equivalent" here is that both servers can provide the same service and, where appropriate, the same authentication information, such as PKI certificates, when establishing connections.

In principle, anycast servers could maintain sufficient state that they can both handle packets in the same TCP connection. In order for this to work with DSO, they would need to also share DSO state. It is unlikely that this can be done successfully, however, so we recommend that each anycast server instance maintain its own session state.

If a change in network topology causes packets in a particular TCP connection to be sent to an anycast server instance that does not know about the connection, the new server will automatically terminate the connection with a TCP reset, since it will have no record of the connection, and then the client can reconnect or stop using the connection, as appropriate.

If after the connection is re-established, the client’s assumption that it is connected to the same service is violated in some way, that would be considered to be incorrect behavior in this context. It is however out of the possible scope for this specification to make specific recommendations in this regard; that would be up to follow-on documents that describe specific uses of DNS stateful operations.

The term "long-lived operations" refers to operations such as Push Notification subscriptions [I-D.ietf-dnssd-push], Discovery Relay interface subscriptions [I-D.ietf-dnssd-mdns-relay], and other future long-lived DNS operations that choose to use DSO as their basis. These operations establish state that persists beyond the lifetime of a traditional brief request/response transaction. This document, the base specification for DNS Stateful Operations, defines a framework...
for supporting long-lived operations, but does not itself define any
long-lived operations. Nonetheless, to appreciate the design
rationale behind DNS Stateful Operations, it is helpful to understand
the kind of long-lived operations that it is intended to support.

DNS Stateful Operations uses three kinds of message: "DSO request
messages", "DSO response messages", and "DSO unacknowledged
messages". A DSO request message elicits a DSO response message.
DSO unacknowledged messages are unidirectional messages and do not
induce a DNS response.

Both DSO request messages and DSO unacknowledged messages are
formatted as DNS request messages (the header QR bit is set to zero,
as described in Section 6.2). One difference is that in DSO request
messages the MESSAGE ID field is nonzero; in DSO unacknowledged
messages it is zero.

The content of DSO messages is expressed using type-length-value
(TLV) syntax.

In a DSO request message or DSO unacknowledged message the first TLV
is referred to as the "Primary TLV" and determines the nature of the
operation being performed, including whether it is an acknowledged or
unacknowledged operation; any other TLVs in a DSO request message or
unacknowledged message are referred to as "Additional TLVs" and serve
additional non-primary purposes, which may be related to the primary
purpose, or not, as in the case of the encryption padding TLV.

A DSO response message may contain no TLVs, or it may contain one or
more TLVs as appropriate to the information being communicated. In
the context of DSO response messages, one or more TLVs with the same
DSO-TYPE as the Primary TLV in the corresponding DSO request message
are referred to as "Response Primary TLVs". Any other TLVs with
different DSO-TYPEs are referred to as "Response Additional TLVs".
The Response Primary TLV(s), if present, MUST occur first in the
response message, before any Response Additional TLVs.

Two timers (elapsed time since an event) are defined in this
document:

- an inactivity timer (see Section 7.4 and Section 8.1)
- a keepalive timer (see Section 7.5 and Section 8.1)

The timeouts associated with these timers are called the inactivity
timeout and the keepalive interval, respectively. The term "Session
Timeouts" is used to refer to this pair of timeout values.
Resetting a timer means resetting the timer value to zero and starting the timer again. Clearing a timer means resetting the timer value to zero but NOT starting the timer again.
4. Discussion

There are several use cases for DNS Stateful operations that can be described here.

Firstly, establishing session parameters such as server-defined timeouts is of great use in the general management of persistent connections. For example, using DSO sessions for stub-to-recursive DNS-over-TLS [RFC7858] is more flexible for both the client and the server than attempting to manage sessions using just the edns-tcp-keepalive EDNS0 Option [RFC7828]. The simple set of TLVs defined in this document is sufficient to greatly enhance connection management for this use case.

Secondly, DNS-SD [RFC6763] has evolved into a naturally session-based mechanism where, for example, long-lived subscriptions lend themselves to ‘push’ mechanisms as opposed to polling. Long-lived stateful connections and server-initiated messages align with this use case [I-D.ietf-dnssd-push].

A general use case is that DNS traffic is often bursty but session establishment can be expensive. One challenge with long-lived connections is to maintain sufficient traffic to maintain NAT and firewall state. To mitigate this issue this document introduces a new concept for the DNS, that is DSO "Keepalive traffic". This traffic carries no DNS data and is not considered ‘activity’ in the classic DNS sense, but serves to maintain state in middleboxes, and to assure client and server that they still have connectivity to each other.
5. Applicability

DNS Stateful Operations are applicable in cases where it is useful to maintain an open session between a DNS client and server, where the transport allows such a session to be maintained, and where the transport guarantees in-order delivery of messages, on which DSO depends. Examples of transports that can support session signaling are DNS-over-TCP [RFC1035] [RFC7766] and DNS-over-TLS [RFC7858].

Note that in the case of DNS over TLS, there is no mechanism for upgrading from DNS-over-TCP to DNS-over-TLS (see [RFC7858] section 7).

DNS Stateful Operations are not applicable for transports that cannot support clean session semantics, or that do not guarantee in-order delivery. While in principle such a transport could be constructed over UDP, the current DNS specification over UDP transport [RFC1035] does not provide in-order delivery or session semantics, and hence cannot be used. Similarly, DNS-over-HTTP [I-D.ietf-doh-dns-over-https] cannot be used because HTTP has its own mechanism for managing sessions, and this is incompatible with the mechanism specified here.

No other transports are currently defined for use with DNS Stateful Operations. Such transports can be added in the future, if they meet the requirements set out in the first paragraph of this section.

6. Protocol Details

6.1. DSO Session Establishment

In order for a session to be established between a client and a server, the client must first establish a connection to the server, using an applicable transport (see Section 5).

In some environments it may be known in advance by external means that both client and server support DSO, and in these cases either client or server may initiate DSO messages at any time. In this case, the session is established as soon as the connection is established; this is referred to as implicit session establishment.

However, in the typical case a server will not know in advance whether a client supports DSO, so in general, unless it is known in advance by other means that a client does support DSO, a server MUST NOT initiate DSO request messages or DSO unacknowledged messages until a DSO Session has been mutually established by at least one successful DSO request/response exchange initiated by the client, as
described below. This is referred to as explicit session establishment.

Until a DSO session has been implicitly or explicitly established, a client MUST NOT initiate DSO unacknowledged messages.

A DSO Session is established over a connection by the client sending a DSO request message, such as a DSO Keepalive request message (Section 8.1), and receiving a response, with matching MESSAGE ID, and RCODE set to NOERROR (0), indicating that the DSO request was successful.

If the RCODE in the response is set to DSOTYPENI ("DSO-TYPE Not Implemented", [TBA2] tentatively RCODE 11) this indicates that the server does support DSO, but does not implement the DSO-TYPE of the primary TLV in this DSO request message. A server implementing DSO MUST NOT return DSOTYPENI for a DSO Keepalive request message, because the Keepalive TLV is mandatory to implement. But in the future, if a client attempts to establish a DSO Session using a response-requiring DSO request message using some newly-defined DSO-TYPE that the server does not understand, that would result in a DSOTYPENI response. If the server returns DSOTYPENI then a DSO Session is not considered established, but the client is permitted to continue sending DNS messages on the connection, including other DSO messages such as the DSO Keepalive, which may result in a successful NOERROR response, yielding the establishment of a DSO Session.

If the RCODE is set to any value other than NOERROR (0) or DSOTYPENI ([TBA2] tentatively 11), then the client MUST assume that the server does not implement DSO at all. In this case the client is permitted to continue sending DNS messages on that connection, but the client MUST NOT issue further DSO messages on that connection.

Two other possibilities exist: the server might drop the connection, or the server might send no response to the DSO message. In the first case, the client SHOULD mark the server as not supporting DSO, and not attempt a DSO connection for some period of time (at least an hour) after the failed attempt. The client MAY reconnect but not use DSO, if appropriate.

In the second case, the client SHOULD set a reasonable timeout, after which time the server will be assumed not to support DSO. At this point the client MUST drop the connection to the server, since the server's behavior is out of spec, and hence its state is undefined. The client MAY reconnect, but not use DSO, if appropriate.
When the server receives a DSO request message from a client, and transmits a successful NOERROR response to that request, the server considers the DSO Session established.

When the client receives the server’s NOERROR response to its DSO request message, the client considers the DSO Session established.

Once a DSO Session has been established, either end may unilaterally send appropriate DSO messages at any time, and therefore either client or server may be the initiator of a message.

Once a DSO Session has been established, clients and servers should behave as described in this specification with regard to inactivity timeouts and session termination, not as previously prescribed in the earlier specification for DNS over TCP [RFC7766].

Because the Keepalive TLV can’t fail (that is, can’t return an RCODE other than NOERROR), it is an ideal candidate for use in establishing a DSO session. Any other option that can only succeed MAY also be used to establish a DSO session. For clients that implement only the DSO-TYPEs defined in this base specification, sending a Keepalive TLV is the only DSO request message they have available to initiate a DSO Session. Even for clients that do implement other future DSO-TYPEs, for simplicity they MAY elect to always send an initial DSO Keepalive request message as their way of initiating a DSO Session. A future definition of a new response-requiring DSO-TYPE gives implementers the option of using that new DSO-TYPE if they wish, but does not change the fact that sending a Keepalive TLV remains a valid way of initiating a DSO Session.

6.1.1. Connection Sharing

As previously specified for DNS over TCP [RFC7766]:

To mitigate the risk of unintentional server overload, DNS clients MUST take care to minimize the number of concurrent TCP connections made to any individual server. It is RECOMMENDED that for any given client/server interaction there SHOULD be no more than one connection for regular queries, one for zone transfers, and one for each protocol that is being used on top of TCP (for example, if the resolver was using TLS). However, it is noted that certain primary/secondary configurations with many busy zones might need to use more than one TCP connection for zone transfers for operational reasons (for example, to support concurrent transfers of multiple zones).

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 DSO 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 requirement has two benefits. First, it reduces unnecessary connection load on the DNS server. Second, it avoids paying the TCP slow start penalty when making subsequent connections to the same server.

However, server implementers and operators should be aware that connection sharing may not be possible in all cases. A single host 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.

6.1.2. Zero Round-Trip Operation

DSO permits zero round-trip operation using TCP Fast Open [RFC7413] and TLS 1.3 [I-D.ietf-tls-tls13] to reduce or eliminate round trips in session establishment.

A client MAY send multiple response-requiring DSO messages using TCP fast open or TLS 1.3 early data, without having to wait for a response to the first request message to confirm successful establishment of a DSO session.

However, a client MUST NOT send non-response-requiring DSO request messages until after a DSO Session has been mutually established.

Similarly, a server MUST NOT send DSO request messages until it has received a response-requiring DSO request message from a client and transmitted a successful NOERROR response for that request.

Caution must be taken to ensure that DSO messages sent before the first round-trip is completed are idempotent, or are otherwise immune to any problems that could be result from the inadvertent replay that can occur with zero round-trip operation.
6.1.3. Middlebox Considerations

Where an application-layer middlebox (e.g., a DNS proxy, forwarder, or session multiplexer) is in the path, care must be taken to avoid inappropriately passing session signaling through the middlebox.

In cases where a DSO session is terminated on one side of a middlebox, and then some session is opened on the other side of the middlebox in order to satisfy requests sent over the first DSO session, any such session MUST be treated as a separate session. If the middlebox does implement DSO sessions, it MUST handle unrecognized TLVs in the same way as any other DSO implementation as described below in Section 6.2.2.4.

This does not preclude the use of DSO messages in the presence of an IP-layer middlebox, such as a NAT that rewrites IP-layer and/or transport-layer headers but otherwise preserves the effect of a single session between the client and the server. And of course it does not apply to middleboxes that do not implement DNS Stateless Operations.

These restrictions do not apply to such middleboxes: since they have no way to understand a DSO message, a pass-through middlebox like the one described in the previous paragraph will pass DSO messages unchanged or drop them (or possibly drop the connection). A middlebox that is not doing a strict pass-through will have no way to know on which connection to forward a DSO message, and therefore will not be able to behave incorrectly.

To illustrate the above, consider a network where a middlebox terminates one or more TCP connections from clients and multiplexes the queries therein over a single TCP connection to an upstream server. The DSO messages and any associated state are specific to the individual TCP connections. A DSO-aware middlebox MAY in some circumstances be able to retain associated state and pass it between the client and server (or vice versa) but this would be highly TLV-specific. For example, the middlebox may be able to maintain a list of which clients have made Push Notification subscriptions [I-D.ietf-dnssd-push] and make its own subscription(s) on their behalf, relaying any subsequent notifications to the client (or clients) that have subscribed to that particular notification.
6.2. Message Format

A DSO message begins with the standard twelve-byte DNS message header [RFC1035] with the OPCODE field set to the DSO OPCODE ([TBA1] tentatively 6). However, unlike standard DNS messages, the question section, answer section, authority records section and additional records sections are not present. The corresponding count fields (QDCOUNT, ANCOUNT, NSCOUNT, ARCOUNT) MUST be set to zero on transmission.

If a DSO message is received where any of the count fields are not zero, then a FORMERR MUST be returned.
6.2.1. DNS Header Fields in DSO Messages

In an unacknowledged message the MESSAGE ID field MUST be set to zero. In an acknowledged request message the MESSAGE ID field MUST be set to a unique nonzero value, that the initiator is not currently using for any other active operation on this connection. For the purposes here, a MESSAGE ID is in use in this DSO Session if the initiator has used it in a request for which it is still awaiting a response, or if the client has used it to set up a long-lived operation that has not yet been cancelled. For example, a long-lived operation could be a Push Notification subscription [I-D.ietf-dnssd-push] or a Discovery Relay interface subscription [I-D.ietf-dnssd-mdns-relay].

Whether a message is acknowledged or unacknowledged is determined only by the specification for the Primary TLV. An acknowledgment cannot be requested by including a nonzero message ID in a message the primary TLV of which is specified to be unacknowledged, nor can an acknowledgment be prevented by sending a message ID of zero in a message with a primary TLV that is specified to be acknowledged. A responder that receives either such malformed message MUST treat it as a fatal error and forcibly abort the connection immediately.

In a request or unacknowledged message the DNS Header QR bit MUST be zero (QR=0). If the QR bit is not zero the message is not a request or unacknowledged message.

In a response message the DNS Header QR bit MUST be one (QR=1). If the QR bit is not one the message is not a response message.

In a response message (QR=1) the MESSAGE ID field MUST contain a copy of the value of the MESSAGE ID field in the request message being responded to. In a response message (QR=1) the MESSAGE ID field MUST NOT be zero. If a response message (QR=1) is received where the MESSAGE ID is zero this is a fatal error and the recipient MUST forcibly abort the connection immediately.

The DNS Header OPCODE field holds the DSO OPCODE value ([TBA1] tentatively 6).

The Z bits are currently unused in DSO messages, and in both DSO requests and DSO responses the Z bits MUST be set to zero (0) on transmission and MUST be silently ignored on reception.
In a DNS request message (QR=0) the RCODE is set according to the definition of the request. For example, in a Retry Delay message (Section 7.6.1) the RCODE indicates the reason for termination. However, in most cases, except where clearly specified otherwise, in a DNS request message (QR=0) the RCODE is set to zero on transmission, and silently ignored on reception.

The RCODE value in a response message (QR=1) may be one of the following values:

<table>
<thead>
<tr>
<th>Code</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOERROR</td>
<td>Operation processed successfully</td>
</tr>
<tr>
<td>1</td>
<td>FORMERR</td>
<td>Format error</td>
</tr>
<tr>
<td>2</td>
<td>SERVFAIL</td>
<td>Server failed to process request due to a problem with the server</td>
</tr>
<tr>
<td>3</td>
<td>NXDOMAIN</td>
<td>Name Error -- Named entity does not exist (TLV-dependent)</td>
</tr>
<tr>
<td>4</td>
<td>NOTIMP</td>
<td>DSO not supported</td>
</tr>
<tr>
<td>5</td>
<td>REFUSED</td>
<td>Operation declined for policy reasons</td>
</tr>
<tr>
<td>9</td>
<td>NOTAUTH</td>
<td>Not Authoritative (TLV-dependent)</td>
</tr>
<tr>
<td>[TBA2]</td>
<td>DSOTYPENI</td>
<td>Primary TLV’s DSO-Type is not implemented</td>
</tr>
</tbody>
</table>

Use of the above RCODEs is likely to be common in DSO but does not preclude the definition and use of other codes in future documents that make use of DSO.

If a document defining a new DSO-TYPE makes use of NXDOMAIN (Name Error) or NOTAUTH (Not Authoritative) then that document MUST specify the specific interpretation of these RCODE values in the context of that new DSO TLV.
6.2.2. DSO Data

The standard twelve-byte DNS message header with its zero-valued count fields is followed by the DSO Data, expressed using TLV syntax, as described below Section 6.2.2.1.

A DSO request message or DSO unacknowledged message MUST contain at least one TLV. The first TLV in a DSO request message or DSO unacknowledged message is referred to as the "Primary TLV" and determines the nature of the operation being performed, including whether it is an acknowledged or unacknowledged operation. In some cases it may be appropriate to include other TLVs in a request message or unacknowledged message, such as the Encryption Padding TLV (Section 8.3), and these extra TLVs are referred to as the "Additional TLVs" and are not limited to what is defined in this document. New "Additional TLVs" may be defined in the future and those definitions will describe when their use is appropriate.

A DSO response message may contain no TLVs, or it may be specified to contain one or more TLVs appropriate to the information being communicated. This includes "Primary TLVs" and "Additional TLVs" defined in this document as well as in future TLV definitions. It may be permissible for an additional TLV to appear in a response to a primary TLV even though the specification of that primary TLV does not specify it explicitly. See Section 9.2 for more information.

A DSO response message may contain one or more TLVs with DSO-TYPE the same as the Primary TLV from the corresponding DSO request message, in which case those TLV(s) are referred to as "Response Primary TLVs". A DSO response message is not required to carry Response Primary TLVs. The MESSAGE ID field in the DNS message header is sufficient to identify the DSO request message to which this response message relates.

A DSO response message may contain one or more TLVs with DSO-TYPEs different from the Primary TLV from the corresponding DSO request message, in which case those TLV(s) are referred to as "Response Additional TLVs".

Response Primary TLV(s), if present, MUST occur first in the response message, before any Response Additional TLVs.

It is anticipated that most DSO operations will be specified to use request messages, which generate corresponding responses. In some specialized high-traffic use cases, it may be appropriate to specify unacknowledged messages. Unacknowledged messages can be more efficient on the network, because they don’t generate a stream of corresponding reply messages. Using unacknowledged messages can also
simplify software in some cases, by removing need for an initiator to maintain state while it waits to receive replies it doesn’t care about. When the specification for a particular TLV states that, when used as a Primary TLV (i.e., first) in an outgoing DNS request message (i.e., QR=0), that message is to be unacknowledged, the MESSAGE ID field MUST be set to zero and the receiver MUST NOT generate any response message corresponding to this unacknowledged message.

The previous point, that the receiver MUST NOT generate responses to unacknowledged messages, applies even in the case of errors. When a DSO message is received where both the QR bit and the MESSAGE ID field are zero, the receiver MUST NOT generate any response. For example, if the DSO-TYPE in the Primary TLV is unrecognized, then a DSOTYPENI error MUST NOT be returned; instead the receiver MUST forcibly abort the connection immediately.

Unacknowledged messages MUST NOT be used "speculatively" in cases where the sender doesn’t know if the receiver supports the Primary TLV in the message, because there is no way to receive any response to indicate success or failure. Unacknowledged messages are only appropriate in cases where the sender already knows that the receiver supports, and wishes to receive, these messages.

For example, after a client has subscribed for Push Notifications [I-D.ietf-dnssd-push], the subsequent event notifications are then sent as unacknowledged messages, and this is appropriate because the client initiated the message stream by virtue of its Push Notification subscription, thereby indicating its support of Push Notifications, and its desire to receive those notifications.

Similarly, after a Discovery Relay client has subscribed to receive inbound mDNS (multicast DNS, [RFC6762]) traffic from a Discovery Relay, the subsequent stream of received packets is then sent using unacknowledged messages, and this is appropriate because the client initiated the message stream by virtue of its Discovery Relay link subscription, thereby indicating its support of Discovery Relay, and its desire to receive inbound mDNS packets over that DSO session [I-D.ietf-dnssd-mdns-relay].
6.2.2.1. TLV Syntax

All TLVs, whether used as "Primary", "Additional", "Response Primary", or "Response Additional", use the same encoding syntax.

Specifications that define new TLVs must specify whether the DSO-TYPE can be used as the Primary TLV, used as an Additional TLV, or used in either context, both in the case of requests and of responses. The specification for a TLV must also state whether, when used as the Primary (i.e., first) TLV in a DNS request message (i.e., QR=0), that DSO message is to be acknowledged. If the DSO message is to be acknowledged, the specification must also state which TLVs, if any, are to be included in the response. The Primary TLV may or may not be contained in the response, depending on what is specified for that TLV.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          DSO-TYPE                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          DSO-LENGTH                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
/                           DSO-DATA                            /
/                                                               /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DSO-TYPE: A 16-bit unsigned integer, in network (big endian) byte order, giving the DSO-TYPE of the current DSO TLV per the IANA DSO Type Code Registry.

DSO-LENGTH: A 16-bit unsigned integer, in network (big endian) byte order, giving the size in bytes of the DSO-DATA.

DSO-DATA: Type-code specific format. The generic DSO machinery treats the DSO-DATA as an opaque "blob" without attempting to interpret it. Interpretation of the meaning of the DSO-DATA for a particular DSO-TYPE is the responsibility of the software that implements that DSO-TYPE.
6.2.2.2. Request TLVs

The first TLV in a DSO request message or unacknowledged message is the "Primary TLV" and indicates the operation to be performed. A DSO request message or unacknowledged message MUST contain at least one TLV, the Primary TLV.

Immediately following the Primary TLV, a DSO request message or unacknowledged message MAY contain one or more "Additional TLVs", which specify additional parameters relating to the operation.

6.2.2.3. Response TLVs

Depending on the operation, a DSO response message MAY contain no TLVs, 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. Or it may contain a single response TLV, with the same DSO-TYPE as the Primary TLV in the request message. Alternatively it may contain one or more TLVs of other types, or a combination of the above, as appropriate for the information that needs to be communicated. The specification for each DSO TLV determines what TLVs are required in a response to a request using that TLV.

If a DSO response is received for an operation where the specification requires that the response carry a particular TLV or TLVs, and the required TLV(s) are not present, then this is a fatal error and the recipient of the defective response message MUST forcibly abort the connection immediately.
6.2.2.4. Unrecognized TLVs

If DSO request message is received containing an unrecognized Primary TLV, with a nonzero MESSAGE ID (indicating that a response is expected), then the receiver MUST send an error response with matching MESSAGE ID, and RCODE DSOTYPENI (TBA2 tentatively 11). The error response MUST NOT contain a copy of the unrecognized Primary TLV.

If DSO unacknowledged message is received containing an unrecognized Primary TLV, with a zero MESSAGE ID (indicating that no response is expected), then this is a fatal error and the recipient MUST forcibly abort the connection immediately.

If a DSO request message or unacknowledged message is received where the Primary TLV is recognized, containing one or more unrecognized Additional TLVs, the unrecognized Additional TLVs MUST be silently ignored, and the remainder of the message is interpreted and handled as if the unrecognized parts were not present.

Similarly, if a DSO response message is received containing one or more unrecognized TLVs, the unrecognized TLVs MUST be silently ignored, and the remainder of the message is interpreted and handled as if the unrecognized parts were not present.
6.2.3. EDNS(0) and TSIG

Since the ARCOUNT field MUST be zero, a DSO message can’t contain a valid EDNS(0) option in the additional records section. If functionality provided by current or future EDNS(0) options is desired for DSO messages, one or more new DSO TLVs need to be defined to carry the necessary information.

For example, the EDNS(0) Padding Option [RFC7830] used for security purposes is not permitted in a DSO message, so if message padding is desired for DSO messages then the Encryption Padding TLV described in Section 8.3 MUST be used.

Similarly, a DSO 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 DSO data appears *after* the additional records section, it would not be included in the signature calculation. If use of signatures with DSO messages becomes necessary in the future, a new DSO TLV needs to be defined to perform this function.

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

Although messages may contain other EDNS(0) options as appropriate, this specification explicitly prohibits use of the edns-tcp-keepalive EDNS0 Option [RFC7828] in *any* messages sent on a DSO Session (because it is obsoleted by the functionality provided by the DSO Keepalive operation). If any message sent on a DSO Session contains an edns-tcp-keepalive EDNS0 Option this is a fatal error and the recipient of the defective message MUST forcibly abort the connection immediately.
6.3. Message Handling

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).

As described above in Section 6.2.1 whether an outgoing message with QR=0 is unacknowledged or acknowledged is determined by the specification for the Primary TLV, which in turn determines whether the MESSAGE ID field in that outgoing message will be zero or nonzero.

A DSO unacknowledged message has both the QR bit and the MESSAGE ID field set to zero, and MUST NOT elicit a response.

Every DSO request message (QR=0) with a nonzero MESSAGE ID field is an acknowledged DSO request, and MUST elicit a corresponding response (QR=1), which MUST have the same MESSAGE ID in the DNS message header as in the corresponding request.

Valid DSO request messages sent by the client with a nonzero MESSAGE ID field elicit a response from the server, and Valid DSO request messages sent by the server with a nonzero MESSAGE ID field elicit a response from the client.

The namespaces of 16-bit MESSAGE IDs are independent in each direction. This means it is *not* an error for both client and server to send request messages at the same time as each other, using the same MESSAGE ID, in different directions. This simplification is necessary in order for the protocol to be implementable. It would be infeasible to require the client and server to coordinate with each other regarding allocation of new unique MESSAGE IDs. It is also not necessary to require the client and server to coordinate with each other regarding allocation of new unique MESSAGE IDs. The value of the 16-bit MESSAGE ID combined with the identity of the initiator (client or server) is sufficient to unambiguously identify the operation in question. This can be thought of as a 17-bit message identifier space, using message identifiers 0x00001-0xFFFF for client-to-server DSO request messages, and message identifiers 0x10001-0x1FFFF for server-to-client DSO request messages. The least-significant 16 bits are stored explicitly in the MESSAGE ID field of the DSO message, and the most-significant bit is implicit from the direction of the message.

As described above in Section 6.2.1, an initiator MUST NOT reuse a MESSAGE ID that it already has in use for an outstanding request (unless specified otherwise by the relevant specification for the DSO-TYPE in question). At the very least, this means that a MESSAGE ID can’t be reused in a particular direction on a particular DSO.
Session while the initiator is waiting for a response to a previous request using that MESSAGE ID on that DSO Session (unless specified otherwise by the relevant specification for the DSO-TYPE in question), and for a long-lived operation the MESSAGE ID for the operation can’t be reused while that operation remains active.

If a client or server receives a response (QR=1) where the MESSAGE ID is zero, or is any other value that does not match the MESSAGE ID of any of its outstanding operations, this is a fatal error and the recipient MUST forcibly abort the connection immediately.

If a responder receives a request (QR=0) where the MESSAGE ID is not zero, and the responder tracks query MESSAGE IDs, and the MESSAGE ID matches the MESSAGE ID of a query it received for which a response has not yet been sent, it MUST forcibly abort the connection immediately. This behavior is required to prevent a hypothetical attack that takes advantage of undefined behavior in this case. However, if the server does not track MESSAGE IDs in this way, no such risk exists, so tracking MESSAGE IDs just to implement this sanity check is not required.

6.3.1. Error Responses

When an unacknowledged DSO message type is received (MESSAGE ID field is zero), the receiver SHOULD already be expecting this DSO message type. Section 6.2.2.4 describes the handling of unknown DSO message types. Parsing errors MUST also result in the receiver aborting the connection. When an unacknowledged DSO message of an unexpected type is received, the receiver should abort the connection. Other internal errors processing the unacknowledged DSO message are implementation dependent as to whether the connection should be aborted according to the severity of the error.

When an acknowledged DSO request message is unsuccessful for some reason, the responder returns an error code to the initiator.

In the case of a server returning an error code to a client in response to an unsuccessful DSO request message, the server MAY choose to end the DSO Session, or MAY choose to allow the DSO 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 DSO 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 end the DSO Session by sending a Retry Delay message, as described in Section 7.6.1.
Upon receiving an error response from the server, a client SHOULD NOT automatically close the DSO Session. An error relating to one particular operation on a DSO Session does not necessarily imply that all other operations on that DSO 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 end the DSO 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 end the DSO Session by sending the client a Retry Delay message (Section 7.6.1), then the client SHOULD continue to use that DSO Session for subsequent operations.

6.4. Flow Control Considerations

Because unacknowledged DSO messages do not generate an immediate response from the responder, if there is no other traffic flowing from the responder to the initiator, this can result in a 200ms delay before the TCP acknowledgment is sent to the initiator [NagleDA]. If the initiator has another message pending, but has not yet filled its output buffer, this can delay the delivery of that message by more than 200ms. In many cases, this will make no difference. However, implementors should be aware of this issue. Some operating systems offer ways to disable the 200ms TCP acknowledgment delay; this may be useful for relatively low-traffic sessions, or sessions with bursty traffic flows.

6.5. Responder-Initiated Operation Cancellation

This document, the base specification for DNS Stateful Operations, does not itself define any long-lived operations, but it defines a framework for supporting long-lived operations, such as Push Notification subscriptions [I-D.ietf-dnssd-push] and Discovery Relay interface subscriptions [I-D.ietf-dnssd-mdns-relay].

Generally speaking, a long-lived operation is initiated by the initiator, and, if successful, remains active until the initiator terminates the operation.

However, it is possible that a long-lived operation may be valid at the time it was initiated, but then a later change of circumstances may render that previously valid operation invalid.

For example, a long-lived client operation may pertain to a name that the server is authoritative for, but then the server configuration is changed such that it is no longer authoritative for that name.
In such cases, instead of terminating the entire session it may be desirable for the responder to be able to cancel selectively only those operations that have become invalid.

The responder performs this selective cancellation by sending a new response message, with the MESSAGE ID field containing the MESSAGE ID of the long-lived operation that is to be terminated (that it had previously acknowledged with a NOERROR RCODE), and the RCODE field of the new response message giving the reason for cancellation.

After a response message with nonzero RCODE has been sent, that operation has been terminated from the responder’s point of view, and the responder sends no more messages relating to that operation.

After a response message with nonzero RCODE has been received by the initiator, that operation has been terminated from the initiator’s point of view, and the cancelled operation’s MESSAGE ID is now free for reuse.
7. DSO Session Lifecycle and Timers

7.1. DSO Session Initiation

A DSO Session begins as described in Section 6.1.

The client may perform as many DNS operations as it wishes using the newly created DSO Session. When the client has multiple messages to send, it SHOULD NOT wait for each response before sending the next message. This prevents TCP's delayed acknowledgement algorithm from forcing the client into a slow lock-step. The server MUST act on messages in the order they are transmitted, but SHOULD NOT delay sending responses to those messages as they become available in order to return them in the order the requests were received. [RFC7766] section 3.3 specifies this in more detail.

7.2. DSO Session Timeouts

Two timeout values are associated with a DSO Session: the inactivity timeout, and the keepalive interval. Both values are communicated in the same TLV, the Keepalive TLV (Section 8.1).

The first timeout value, the inactivity timeout, is the maximum time for which a client may speculatively keep a DSO Session open with no operations pending (e.g., an outstanding DNS Push request) in the expectation that it may have future requests to send to that server.

The second timeout value, the keepalive interval, is the maximum permitted interval between messages if the client wishes to keep the DSO Session alive.

The two timeout values are independent. The inactivity timeout may be lower, the same, or higher than the keepalive interval, though in most cases the inactivity timeout is expected to be shorter than the keepalive interval.

A shorter inactivity timeout with a longer keepalive interval signals to the client that it should not speculatively keep an inactive DSO Session open for very long without reason, but when it does have an active reason to keep a DSO Session open, it doesn't need to be sending an aggressive level of DSO keepalive traffic to maintain that session. An example of this would be a client that has subscribed to DNS Push notifications: in this case, the client is not sending any traffic to the server, but the session is not inactive, because there is a pending request to the server to receive push notifications.

A longer inactivity timeout with a shorter keepalive interval signals to the client that it may speculatively keep an inactive DSO Session...
open for a long time, but to maintain that inactive DSO Session it
should be sending a lot of DSO keepalive traffic. This configuration
is expected to be less common.

In the usual case where the inactivity timeout is shorter than the
keepalive interval, it is only when a client has a very long-lived,
low-traffic, operation that the keepalive interval comes into play,
to ensure that a sufficient residual amount of traffic is generated
to maintain NAT and firewall state and to assure client and server
that they still have connectivity to each other.

On a new DSO Session, if no explicit DSO Keepalive message exchange
has taken place, the default value for both timeouts is 15 seconds.

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

7.3. Inactive DSO Sessions

At both servers and clients, the generation or reception of any
complete DNS message, including DNS requests, responses, updates, or
DSO messages, resets both timers for that DSO Session, with the
exception that a DSO Keepalive message resets only the keepalive
timer, not the inactivity timeout timer.

In addition, for as long as the client has an outstanding operation
in progress, the inactivity timer remains cleared, and an inactivity
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 inactivity timer is cleared upon
transmission of a request and remains cleared until reception of the
corresponding response. At the server, the inactivity timer is
cleared upon reception of a request and remains cleared until
transmission of the corresponding response.

For long-lived DNS Stateful operations (such as a Push Notification
subscription [I-D.ietf-dnssd-push] or a Discovery Relay interface
subscription [I-D.ietf-dnssd-mdns-relay]), an operation is considered
in progress for as long as the operation is active, until it is
cancelled. This means that a DSO Session can exist, with active
operations, with no messages flowing in either direction, for far
longer than the inactivity timeout, and this is not an error. This
is why there are two separate timers: the inactivity timeout, and the
keepalive interval. Just because a DSO Session has no traffic for an
extended period of time does not automatically make that DSO Session "inactive", if it has an active operation that is awaiting events.
7.4. The Inactivity Timeout

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

7.4.1. Closing Inactive DSO Sessions

When a connection’s inactivity timeout is reached the client MUST begin closing the idle connection, but a client is not required to keep an idle connection open until the inactivity timeout is reached. A client MAY close a DSO Session at any time, at the client’s discretion. If a client determines that it has no current or reasonably anticipated future need for a currently inactive DSO Session, then the client SHOULD gracefully close that connection.

If, at any time during the life of the DSO Session, the inactivity timeout value (i.e., 15 seconds by default) elapses without there being any operation active on the DSO Session, the client MUST close the connection gracefully.

If, at any time during the life of the DSO Session, twice the inactivity timeout value (i.e., 30 seconds by default), or five seconds, if twice the inactivity timeout value is less than five seconds, elapses without there being any operation active on the DSO Session, the server MUST consider the client delinquent, and MUST forcibly abort the DSO Session.

In this context, an operation being active on a DSO Session includes a query waiting for a response, an update waiting for a response, or an active long-lived operation, but not a DSO Keepalive message exchange itself. A DSO Keepalive message exchange resets only the keepalive interval timer, not the inactivity timeout timer.

If the client wishes to keep an inactive DSO Session open for longer than the default duration then it uses the DSO Keepalive message to request longer timeout values, as described in Section 8.1.
7.4.2. Values for the Inactivity Timeout

For the inactivity timeout value, lower values result in more frequent DSO Session teardown and re-establishment. Higher values result in lower traffic and lower CPU load on the server, but higher memory burden to maintain state for inactive DSO Sessions.

A server may dictate any value it chooses for the inactivity timeout (either in a response to a client-initiated request, or in a server-initiated message) including values under one second, or even zero.

An inactivity timeout of zero informs the client that it should not speculatively maintain idle connections at all, and as soon as the client has completed the operation or operations relating to this server, the client should immediately begin closing this session.

A server will abort an idle client session after twice the inactivity timeout value, or five seconds, whichever is greater. In the case of a zero inactivity timeout value, this means that if a client fails to close an idle client session then the server will forcibly abort the idle session after five seconds.

An inactivity timeout of 0xFFFFFFFF represents "infinity" and informs the client that it may keep an idle connection open as long as it wishes. Note that after granting an unlimited inactivity timeout in this way, at any point the server may revise that inactivity timeout by sending a new DSO Keepalive message dictating new Session Timeout values to the client.

The largest *finite* inactivity timeout supported by the current Keepalive TLV is 0xFFFFFFFF (2^32-2 milliseconds, approximately 49.7 days).
7.5. The Keepalive Interval

The purpose of the keepalive interval is to manage the generation of sufficient messages to maintain state in middleboxes (such as 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 to establish a new session if appropriate.

7.5.1. Keepalive Interval Expiry

If, at any time during the life of the DSO Session, the keepalive interval value (i.e., 15 seconds by default) elapses without any DNS messages being sent or received on a DSO Session, the client MUST take action to keep the DSO Session alive, by sending a DSO Keepalive message (Section 8.1). A DSO Keepalive message exchange resets only the keepalive timer, not the inactivity timer.

If a client disconnects from the network abruptly, without cleanly closing its DSO Session, perhaps leaving a long-lived operation uncanceled, the server learns of this after failing to receive the required DSO keepalive traffic from that client. If, at any time during the life of the DSO Session, twice the keepalive interval value (i.e., 30 seconds by default) elapses without any DNS messages being sent or received on a DSO Session, the server SHOULD consider the client delinquent, and SHOULD forcibly abort the DSO Session.

7.5.2. Values for the Keepalive Interval

For the keepalive interval value, lower values result in a higher volume of DSO 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 DSO Session open for the same length of time (determined by the inactivity timeout) regardless of the level of DSO keepalive traffic required.

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 DSO 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 DSO 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 outside that 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.

By default it is RECOMMENDED that clients request, and servers grant, a keepalive interval of 60 minutes. This keepalive interval provides for reasonably timely detection if a client abruptly disconnects without cleanly closing the session, and is sufficient to maintain state in firewalls and NAT gateways that follow the IETF recommended Best Current Practice that the "established connection idle-timeout" used by middleboxes be at least 2 hours 4 minutes [RFC5382] [RFC7857].

Note that the lower the keepalive interval value, the higher the load on client and server. For example, a hypothetical keepalive interval value of 100ms would result in a continuous stream of at least ten messages per second, in both directions, to keep the DSO 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 DSO Keepalive message (either a response to a client-initiated request, or a server-initiated message) with a keepalive interval value less than ten seconds. If a client receives a DSO Keepalive message specifying a keepalive interval value less than ten seconds this is a fatal error and the client MUST forcibly abort the connection immediately.

A keepalive interval value of 0xFFFFFFFF represents "infinity" and informs the client that it should generate no DSO keepalive traffic. Note that after signaling that the client should generate no DSO keepalive traffic in this way, at any point the server may revise that DSO keepalive traffic requirement by sending a new DSO Keepalive message dictating new Session Timeout values to the client.

The largest *finite* keepalive interval supported by the current Keepalive TLV is 0xFFFFFFFF (2^32-2 milliseconds, approximately 49.7 days).
7.6. Server-Initiated Session Termination

In addition to cancelling individual long-lived operations selectively (Section 6.5) there are also occasions where a server may need to terminate one or more entire sessions. An entire session may need to be terminated if the client is defective in some way, or departs from the network without closing its session. Sessions may also need to be terminated if the server becomes overloaded, or if the server is reconfigured and lacks the ability to be selective about which operations need to be cancelled.

This section discusses various reasons a session may be terminated, and the mechanisms for doing so.

In normal operation, closing a DSO Session is the client’s responsibility. The client makes the determination of when to close a DSO Session based on an evaluation of both its own needs, and the inactivity timeout value dictated by the server. A server only causes a DSO Session to be ended in the exceptional circumstances outlined below.

Some of the exceptional situations in which a server may terminate a DSO Session include:

- The server application software or underlying operating system is shutting down or restarting.
- The server application software terminates unexpectedly (perhaps due to a bug that makes it crash).
- The server is undergoing a reconfiguration or maintenance procedure, that, due to the way the server software is implemented, requires clients to be disconnected. For example, some software is implemented such that it reads a configuration file at startup, and changing the server’s configuration entails modifying the configuration file and then killing and restarting the server software, which generally entails a loss of network connections.
- The client fails to meet its obligation to generate the required DSO keepalive traffic, or to close an inactive session by the prescribed time (twice the time interval dictated by the server, or five seconds, whichever is greater, as described in Section 7.2).
- The client sends a grossly invalid or malformed request that is indicative of a seriously defective client implementation.
The server is over capacity and needs to shed some load.

7.6.1. Server-Initiated Retry Delay Message

In the cases described above where a server elects to terminate a DSO Session, it could do so simply by forcibly aborting the connection. However, if it did this the likely behavior of the client might be simply to treat this as a network failure and reconnect immediately, putting more burden on the server.

Therefore, to avoid this reconnection implosion, a server SHOULD instead choose to shed client load by sending a Retry Delay message, with an appropriate RCODE value informing the client of the reason the DSO Session needs to be terminated. The format of the Retry Delay TLV, and the interpretations of the various RCODE values, are described in Section 8.2. After sending a Retry Delay message, the server MUST NOT send any further messages on that DSO Session.

The server MAY randomize retry delays in situations where many retry delays are sent in quick succession, so as to avoid all the clients attempting to reconnect at once. In general, implementations should avoid using the Retry Delay message in a way that would result in many clients reconnecting at the same time, if every client attempts to reconnect at the exact time specified.

Upon receipt of a Retry Delay message from the server, the client MUST make note of the reconnect delay for this server, and then immediately close the connection gracefully.

After sending a Retry Delay message 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 forcibly abort the connection.

A Retry Delay message MUST NOT be initiated by a client. If a server receives a Retry Delay message this is a fatal error and the server MUST forcibly abort the connection immediately.
7.6.1.1. Outstanding Operations

At the instant a server chooses to initiate a Retry Delay message there may be DNS requests already in flight from client to server on this DSO Session, which will arrive at the server after its Retry Delay 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 message from the server arrives at the client, the client will determine that any DNS requests it previously sent on this DSO 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.

In the case where some, but not all, of the existing operations on a DSO 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 affected DSO Sessions en masse by sending them all a Retry Delay message, 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, depending on the nature of the reconfiguration.

In the case where a server is terminating a large number of DSO 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).
7.6.1.2. Client Reconnection

After a DSO Session is ended by the server (either by sending the client a Retry Delay message, or by forcibly aborting the underlying transport connection) the client SHOULD try to reconnect, to that service instance, or to another suitable service instance, if more than one is available. If reconnecting to the same service instance, the client MUST respect the indicated delay, if available, before attempting to reconnect. Clients should not attempt to randomize the delay; the server will randomly jitter the retry delay values it sends to each client if this behavior is desired.

If the service instance will only be out of service for a short maintenance period, it should use a value a little longer that the expected maintenance window. It should not default to a very large delay value, or clients may not attempt to reconnect after it resumes service.

If a particular service instance does not want a client to reconnect ever (perhaps the service instance is being de-commissioned), it SHOULD set the retry delay to the maximum value 0xFFFFFFFF (2^32-1 milliseconds, approximately 49.7 days). It is not possible to instruct a client to stay away for longer than 49.7 days. If, after 49.7 days, the DNS or other configuration information still indicates that this is the valid service instance for a particular service, then clients MAY attempt to reconnect. In reality, if a client is rebooted or otherwise lose state, it may well attempt to reconnect before 49.7 days elapses, for as long as the DNS or other configuration information continues to indicate that this is the service instance the client should use.
8. Base TLVs for DNS Stateful Operations

This section describes the three base TLVs for DNS Stateful Operations: Keepalive, Retry Delay, and Encryption Padding.

8.1. Keepalive TLV

The Keepalive TLV (DSO-TYPE=1) performs two functions: to reset the keepalive timer for the DSO Session, and to establish the values for the Session Timeouts. The client will request the desired session timeout values and the server will acknowledge with the response values that it requires the client to use.

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

```
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     INACTIVITY TIMEOUT (32 bits)                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          KEEPALIVE INTERVAL (32 bits)                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

INACTIVITY TIMEOUT: The inactivity timeout for the current DSO Session, specified as a 32-bit unsigned integer, in network (big endian) byte order, in units of milliseconds. This is the timeout at which the client MUST begin closing an inactive DSO Session. The inactivity timeout can be any value of the server’s choosing. If the client does not gracefully close an inactive DSO Session, then after twice this interval, or five seconds, whichever is greater, the server will forcibly abort the connection.

KEEPALIVE INTERVAL: The keepalive interval for the current DSO Session, specified as a 32-bit unsigned integer, in network (big endian) byte order, in units of milliseconds. This is the interval at which a client MUST generate DSO keepalive traffic to maintain connection state. The keepalive interval MUST NOT be less than ten seconds. If the client does not generate the mandated DSO keepalive traffic, then after twice this interval the server will forcibly abort the connection. Since the minimum allowed keepalive interval is ten seconds, the minimum time at which a server will forcibly disconnect a client for failing to generate the mandated DSO keepalive traffic is twenty seconds.

The transmission or reception of DSO Keepalive messages (i.e., messages where the Keepalive TLV is the first TLV) reset only the keepalive timer, not the inactivity timer. The reason for this is that periodic DSO Keepalive messages are sent for the sole purpose of
keeping a DSO Session alive, when that DSO Session has current or recent non-maintenance activity that warrants keeping that DSO Session alive. Sending DSO keepalive traffic itself is not considered a client activity; it is considered a maintenance activity that is performed in service of other client activities. If DSO keepalive traffic were to reset the inactivity timer, then that would create a circular livelock where keepalive traffic would be sent indefinitely to keep a DSO Session alive, where the only activity on that DSO Session would be the keepalive traffic keeping the DSO Session alive so that further keepalive traffic can be sent. For a DSO Session to be considered active, it must be carrying something more than just keepalive traffic. This is why merely sending or receiving a DSO Keepalive message does not reset the inactivity timer.

When sent by a client, the DSO Keepalive request message MUST be sent as an acknowledged request, with a nonzero MESSAGE ID. If a server receives a DSO Keepalive message with a zero MESSAGE ID then this is a fatal error and the server MUST forcibly abort the connection immediately. The DSO Keepalive request message resets a DSO Session’s keepalive timer, and at the same time communicates to the server the client’s requested Session Timeout values. In a server response to a client-initiated DSO Keepalive request message, the Session Timouts contain the server’s chosen values from this point forward in the DSO Session, 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.

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

Once a DSO Session is in progress (Section 6.1) a DSO Keepalive message MAY be initiated by a server. When sent by a server, the DSO Keepalive message MUST be sent as an unacknowledged message, with the MESSAGE ID set to zero. The client MUST NOT generate a response to a server-initiated DSO Keepalive message. If a client receives a DSO Keepalive request message with a nonzero MESSAGE ID then this is a fatal error and the client MUST forcibly abort the connection immediately. The unacknowledged DSO Keepalive message from the server resets a DSO Session’s keepalive timer, and at the same time unilaterally informs the client of the new Session Timeout values to use from this point forward in this DSO Session. No client DSO
response message to this unilateral declaration is required or allowed.

In DSO Keepalive response messages, the Keepalive TLV is REQUIRED and is used only as a Response Primary TLV sent as a reply to a DSO Keepalive request message from the client. A Keepalive TLV MUST NOT be added to other responses as a Response Additional TLV. If the server wishes to update a client's Session Timeout values other than in response to a DSO Keepalive request message from the client, then it does so by sending an unacknowledged DSO Keepalive message of its own, as described above.

It is not required that the Keepalive TLV be used in every DSO Session. While many DNS Stateful operations will be used in conjunction with a long-lived session state, not all DNS Stateful operations require long-lived session state, and in some cases the default 15-second value for both the inactivity timeout and keepalive interval may be perfectly appropriate. However, note that for clients that implement only the DSO-TYPEs defined in this document, a Keepalive request message is the only way for a client to initiate a DSO Session.

8.1.1. Client handling of received Session Timeout values

When a client receives a response to its client-initiated DSO Keepalive message, or receives a server-initiated DSO Keepalive message, the client has then received Session Timeout values dictated by the server. The two timeout values contained in the Keepalive TLV from the server may each be higher, lower, or the same as the respective Session Timeout values the client previously had for this DSO Session.

In the case of the keepalive timer, the handling of the received value is straightforward. When a client receives a server-initiated message with the Keepalive TLV as its primary TLV, it resets the keepalive timer. Whenever it receives a Keepalive TLV from the server, either in a server-initiated message or a reply to its own client-initiated Keepalive message, it updates the keepalive interval for the DSO Session. The new keepalive interval indicates the maximum time that may elapse before another message must be sent or received on this DSO Session, if the DSO Session is to remain alive. If the client receives a response to a keepalive message that specifies a keepalive interval shorter than the current keepalive timer, the client MUST immediately send a Keepalive message. However, this should not normally happen in practice: it would require that Keepalive interval the server be shorter than the round-trip time of the connection.
In the case of the inactivity timeout, the handling of the received value is a little more subtle, though the meaning of the inactivity timeout remains as specified -- it still indicates the maximum permissible time allowed without useful activity on a DSO Session. The act of receiving the message containing the Keepalive TLV does not itself reset the inactivity timer. The time elapsed since the last useful activity on this DSO Session is unaffected by exchange of DSO Keepalive messages. The new inactivity timeout value in the Keepalive TLV in the received message does update the timeout associated with the running inactivity timer; that becomes the new maximum permissible time without activity on a DSO Session.

- If the current inactivity timer value is less than the new inactivity timeout, then the DSO Session may remain open for now. When the inactivity timer value reaches the new inactivity timeout, the client MUST then begin closing the DSO Session, as described above.

- If the current inactivity timer value is equal to the new inactivity timeout, then this DSO Session has been inactive for exactly as long as the server will permit, and now the client MUST immediately begin closing this DSO Session.

- If the current inactivity timer value is already greater than the new inactivity timeout, then this DSO Session has already been inactive for longer than the server permits, and the client MUST immediately begin closing this DSO Session.

- If the current inactivity timer value is already more than twice the new inactivity timeout, then the client is immediately considered delinquent (this DSO Session is immediately eligible to be forcibly terminated by the server) and the client MUST immediately begin closing this DSO Session. However if a server abruptly reduces the inactivity timeout in this way, then, to give the client time to close the connection gracefully before the server resorts to forcibly aborting it, the server SHOULD give the client an additional grace period of one quarter of the new inactivity timeout, or five seconds, whichever is greater.
8.1.2. Relation to edns-tcp-keepalive EDNS0 Option

The inactivity timeout value in the Keepalive TLV (DSO-TYPE=1) has similar intent to the edns-tcp-keepalive EDNS0 Option [RFC7828]. A client/server pair that supports DSO MUST NOT use the edns-tcp-keepalive EDNS0 Option within any message after a DSO Session has been established. A client that has sent a DSO message to establish a session MUST NOT send an edns-tcp-keepalive EDNS0 Option from this point on. Once a DSO Session has been established, if either client or server receives a DNS message over the DSO Session that contains an edns-tcp-keepalive EDNS0 Option, this is a fatal error and the receiver of the edns-tcp-keepalive EDNS0 Option MUST forcibly abort the connection immediately.
8.2. Retry Delay TLV

The Retry Delay TLV (DSO-TYPE=2) can be used as a Primary TLV (unacknowledged) in a server-to-client message, or as a Response Additional TLV in either direction.

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

```
1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     RETRY DELAY (32 bits)                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

RETRY DELAY: A time value, specified as a 32-bit unsigned integer, in network (big endian) byte order, in units of milliseconds, within which the initiator MUST NOT retry this operation, or retry connecting to this server. Recommendations for the RETRY DELAY value are given in Section 7.6.1.

8.2.1. Retry Delay TLV used as a Primary TLV

When sent from server to client, the Retry Delay TLV is used as the Primary TLV in an unacknowledged message. It is used by a server to instruct a client to close the DSO Session and underlying connection, and not to reconnect for the indicated time interval.

In this case it applies to the DSO Session as a whole, and the client MUST begin closing the DSO Session, as described in Section 7.6.1. The RCODE in the message header SHOULD indicate the principal reason for the termination:

- NOERROR indicates a routine shutdown or restart.
- FORMERR indicates that the client requests are too badly malformed for the session to continue.
- SERVFAIL indicates that the server is overloaded due to resource exhaustion and needs to shed load.
- REFUSED indicates that the server has been reconfigured, and at this time it is now unable to perform one or more of the long-lived client operations that were previously being performed on this DSO Session.
- NOTAUTH indicates that the server has been reconfigured and at this time it is now unable to perform one or more of the long-lived client operations that were previously being performed on
this DSO Session because it does not have authority over the names in question (for example, a DNS Push Notification server could be reconfigured such that it is no longer accepting DNS Push Notification requests for one or more of the currently subscribed names).

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

In some cases, when a server sends a Retry Delay message to a client, there may be more than one reason for the server wanting to end the session. Possibly the configuration could have been changed such that some long-lived client operations can no longer be continued due to policy (REFUSED), and other long-lived client operations can no longer be performed due to the server no longer being authoritative for those names (NOTAUTH). In such cases the server MAY use any of the applicable RCODE values, or RCODE=NOERROR (routine shutdown or restart).

Note that the selection of RCODE value in a Retry Delay message is not critical, since the RCODE value is generally used only for information purposes, such as writing to a log file for future human analysis regarding the nature of the disconnection. Generally clients do not modify their behavior depending on the RCODE value. The RETRY DELAY in the message tells the client how long it should wait before attempting a new connection to this service instance.

For clients that do in some way modify their behavior depending on the RCODE value, they should treat unknown RCODE values the same as RCODE=NOERROR (routine shutdown or restart).

A Retry Delay message from server to client is an unacknowledged message; the MESSAGE ID MUST be set to zero in the outgoing message and the client MUST NOT send a response.

A client MUST NOT send a Retry Delay DSO message to a server. If a server receives a DSO message where the Primary TLV is the Retry Delay TLV, this is a fatal error and the server MUST forcibly abort the connection immediately.
8.2.2. Retry Delay TLV used as a Response Additional TLV

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

The indicated time interval during which the initiator SHOULD NOT retry applies only to the failed operation, not to the DSO Session as a whole.

8.3. Encryption Padding TLV

The Encryption Padding TLV (DSO-TYPE=3) can only be used as an Additional or Response Additional TLV. It is only applicable when the DSO Transport layer uses encryption such as TLS.

The DSO-DATA for the the Padding TLV is optional and is a variable length field containing non-specified values. A DSO-LENGTH of 0 essentially provides for 4 bytes of padding (the minimum amount).

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                                                               /
/                   VARIABLE NUMBER OF BYTES                    /
/                                                               /
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

As specified for the EDNS(0) Padding Option [RFC7830] the PADDING bytes SHOULD be set to 0x00. Other values MAY be used, for example, in cases where there is a concern that the padded message could be subject to compression before encryption. PADDING bytes of any value MUST be accepted in the messages received.

The Encryption Padding TLV may be included in either a DSO request, response, or both. As specified for the EDNS(0) Padding Option [RFC7830] if a request is received with an Encryption Padding TLV, then the response MUST also include an Encryption Padding TLV.

The length of padding is intentionally not specified in this document and is a function of current best practices with respect to the type and length of data in the preceding TLVs [I-D.ietf-dprive-padding-policy].
9. Summary Highlights

This section summarizes some noteworthy highlights about various components of the DSO protocol.

9.1. QR bit and MESSAGE ID

In DSO Request Messages the QR bit is 0 and the MESSAGE ID is nonzero.

In DSO Response Messages the QR bit is 1 and the MESSAGE ID is nonzero.

In DSO Unacknowledged Messages the QR bit is 0 and the MESSAGE ID is zero.

The table below illustrates which combinations are legal and how they are interpreted:

<table>
<thead>
<tr>
<th>QR=0</th>
<th>MESSAGE ID zero</th>
<th>MESSAGE ID nonzero</th>
</tr>
</thead>
<tbody>
<tr>
<td>QR=1</td>
<td>Unacknowledged Message</td>
<td>Request Message</td>
</tr>
<tr>
<td>QR=0</td>
<td>Invalid - Fatal Error</td>
<td>Response Message</td>
</tr>
<tr>
<td>QR=1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9.2. TLV Usage

The table below indicates, for each of the three TLVs defined in this document, whether they are valid in each of ten different contexts.

The first five contexts are requests or unacknowledged messages from client to server, and the corresponding responses from server back to client:

- **C-P** - Primary TLV, sent in DSO Request message, from client to server, with nonzero MESSAGE ID indicating that this request MUST generate response message.

- **C-U** - Primary TLV, sent in DSO Unacknowledged message, from client to server, with zero MESSAGE ID indicating that this request MUST NOT generate response message.

- **C-A** - Additional TLV, optionally added to request message or unacknowledged message from client to server.

- **CRP** - Response Primary TLV, included in response message sent back to the client (in response to a client "C-P" request with nonzero MESSAGE ID indicating that a response is required) where the DSO-TYPE of the Response TLV matches the DSO-TYPE of the Primary TLV in the request.

- **CRA** - Response Additional TLV, included in response message sent back to the client (in response to a client "C-P" request with nonzero MESSAGE ID indicating that a response is required) where the DSO-TYPE of the Response TLV does not match the DSO-TYPE of the Primary TLV in the request.

The second five contexts are their counterparts in the opposite direction: requests or unacknowledged messages from server to client, and the corresponding responses from client back to server:

- **S-P** - Primary TLV, sent in DSO Request message, from server to client, with nonzero MESSAGE ID indicating that this request MUST generate response message.

- **S-U** - Primary TLV, sent in DSO Unacknowledged message, from server to client, with zero MESSAGE ID indicating that this request MUST NOT generate response message.

- **S-A** - Additional TLV, optionally added to request message or unacknowledged message from server to client.
o SRP - Response Primary TLV, included in response message sent back to the server (in response to a server "S-P" request with nonzero MESSAGE ID indicating that a response is required) where the DSO-TYPE of the Response TLV matches the DSO-TYPE of the Primary TLV in the request.

o SRA - Response Additional TLV, included in response message sent back to the server (in response to a server "S-P" request with nonzero MESSAGE ID indicating that a response is required) where the DSO-TYPE of the Response TLV does not match the DSO-TYPE of the Primary TLV in the request.

+-------------------------+-------------------------+
<table>
<thead>
<tr>
<th>C-P C-U C-A CRP CRA</th>
<th>S-P S-U S-A SRP SRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeepAlive</td>
<td>X X</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>RetryDelay</td>
<td>X X</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Padding</td>
<td>X X X X</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
</tbody>
</table>

Note that some of the columns in this table are currently empty. The table provides a template for future TLV definitions to follow. It is recommended that definitions of future TLVs include a similar table summarizing the contexts where the new TLV is valid.
10. IANA Considerations

10.1. DSO OPCODE Registration

The IANA is requested to record the value ([TBA1] tentatively) 6 for the DSO OPCODE in the DNS OPCODE Registry. DSO stands for DNS Stateful Operations.

10.2. DSO RCODE Registration

The IANA is requested to record the value ([TBA2] tentatively) 11 for the DSOTYPENI error code in the DNS RCODE Registry. The DSOTYPENI error code ("DSO-TYPE Not Implemented") indicates that the receiver does implement DNS Stateful Operations, but does not implement the specific DSO-TYPE of the primary TLV in the DSO request message.

10.3. DSO Type Code Registry

The IANA is requested to create the 16-bit DSO Type Code Registry, with initial (hexadecimal) values as shown below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Status</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
<td>Standard</td>
<td>RFC-TBD</td>
</tr>
<tr>
<td>0001</td>
<td>KeepAlive</td>
<td>Standard</td>
<td>RFC-TBD</td>
</tr>
<tr>
<td>0002</td>
<td>RetryDelay</td>
<td>Standard</td>
<td>RFC-TBD</td>
</tr>
<tr>
<td>0003</td>
<td>EncryptionPadding</td>
<td>Standard</td>
<td>RFC-TBD</td>
</tr>
<tr>
<td>0004-003F</td>
<td>Unassigned, reserved for DSO session-management TLVs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040-F7FF</td>
<td>Unassigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F800-FBFF</td>
<td>Reserved for experimental/local use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC00-FFFF</td>
<td>Reserved for future expansion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DSO Type Code zero is reserved and is not currently intended for allocation.

Registrations of new DSO Type Codes in the "Reserved for DSO session-management" range 0004-003F and the "Reserved for future expansion"
Requests to register additional new DSO Type Codes in the "Unassigned" range 0040-F7FF are to be recorded by IANA after Expert Review [RFC8126]. The expert review should validate that the requested type code is specified in a way that conforms to this specification, and that the intended use for the code would not be addressed with an experimental/local assignment.

DSO Type Codes in the "experimental/local" range F800-FBFF may be used as Experimental Use or Private Use values [RFC8126] and may be used freely for development purposes, or for other purposes within a single site. No attempt is made to prevent multiple sites from using the same value in different (and incompatible) ways. There is no need for IANA to review such assignments (since IANA does not record them) and assignments are not generally useful for broad interoperability. It is the responsibility of the sites making use of "experimental/local" values to ensure that no conflicts occur within the intended scope of use.

11. 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.

The data field of the "Encryption Padding" TLV could be used as a covert channel.

When designing new DSO TLVs, the potential for data in the TLV to be used as a tracking identifier should be taken into consideration, and should be avoided when not required.

When used without TLS or similar cryptographic protection, a malicious entity maybe able to inject a malicious Retry Delay Unacknowledged Message into the data stream, specifying an unreasonably large RETRY DELAY, causing a denial-of-service attack against the client.

The establishment of DSO sessions has an increasing impact on the number of open TCP connections on a DNS server. Additional resources may be used on the server as a result. However, because the server can limit the number of DSO sessions established and can also close existing DSO sessions as needed, denial of service or resource exhaustion should not be a concern.
11.1. TCP Fast Open Considerations

It would be possible to add a TLV that requires the server to do some significant work, and send that to the server as initial data in a TCP SYN packet. A flood of such packets could be used as a DoS attack on the server. None of the TLVs defined here have this property. If a new TLV is specified that does have this property, the specification should require that some kind of exchange be done with the server before work is done. That is, the TLV that requires work could not be processed without a round-trip from the server to the client to verify that the source address of the packet is reachable.

One way to accomplish this would be to have the client send a TLV indicating that it wishes to have the server do work of this sort; this TLV would not actually result in work being done, but would request a nonce from the server. The client could then use that nonce to request that work be done.

Alternatively, the server could simply disable TCP fast open. This same problem would exist for DNS-over-TLS with TLS early data; the same remedies would apply.

12. Acknowledgements

Thanks to Stephane Bortzmeyer, Tim Chown, Ralph Droms, Paul Hoffman, Jan Komissar, Edward Lewis, Allison Mankin, Rui Paulo, David Schinazi, Manju Shankar Rao, and Bernie Volz for their helpful contributions to this document.
13. References

13.1. Normative References


13.2. Informative References

[I-D.ietf-dnsop-no-response-issue]

[I-D.ietf-dnssd-mdns-relay]

[I-D.ietf-dnssd-push]

[I-D.ietf-doh-dns-over-https]

[I-D.ietf-dprive-padding-policy]

[I-D.ietf-tls-tls13]


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
Abstract

This document specifies a network proxy 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 19, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
Table of Contents

1. Introduction ........................................ 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 ................................... 30
10. IANA Considerations .................................... 31
11. Acknowledgments ....................................... 31
12. References ............................................ 32
  12.1. Normative References ............................... 32
  12.2. Informative References .............................. 33
 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] [Roadmap].

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 at lower data rates, and is not acknowledged. Increasing 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. At the opposite end of the spectrum, 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 large multicast domains, but does require 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.arpa" [I-D.ietf-homenet-dot].

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 is one option. This option has been used for many years at IETF meetings to advertise the IETF Terminal Room printer. Details of this example are given in Appendix A of the Roadmap document [Roadmap]. However, this manual DNS configuration 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 is another option [RegProt] [DNS-UL]. However, this 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 "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 document [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 discussed 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 with this is is that Multicast DNS devices do not routinely announce their records on the network. Generally they remain silent until queried. This means that the complete set of Multicast DNS records in use on a link can only be discovered by active querying, not by passive listening. Because of this, a proxy can only know what names exist on a 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, there is no reasonable way for a proxy to programatically learn all the answers it would need to push up to the remote DNS server using DNS Update. Even if such a mechanism were possible, it would risk generating high load on the network continuously, even when there are 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].

How a client discovers what domain name(s) to use for its service discovery queries, (and consequently what Discovery Proxy or Proxies to use) is described in Section 5.2.

The diagram below illustrates a network topology using a Discovery Proxy to provide discovery service to a remote client.

```
         +--------+    Unicast     +-----------+  +---------+  +---------+
         | Remote |  Communication | Discovery |  | Network |  | Network |
         +--------+                +-----------+  +---------+  +---------+
             |            |            |
--------------------------------------------
Multicast-capable LAN segment (e.g., Ethernet)
```
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 is 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, is 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", when, and only when, they appear in all capitals, as shown here [RFC2119] [RFC8174].

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

- when any host from that set sends a packet to any other host 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.

An overview of the larger collection of related Service Discovery technologies, and how Discovery Proxy relates to those, is given in the Service Discovery Road Map document [Roadmap].
5. Discovery Proxy Operation

In a typical configuration, a Discovery Proxy is configured to be authoritative [RFC1034] [RFC1035] for four or more 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.

One or more DNS subdomains for IPv4 Reverse Mapping records. These subdomains will have names that end in "in-addr.arpa."

One or more DNS subdomains for IPv6 Reverse Mapping records. These subdomains will have names that end in "ip6.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 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 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.

<table>
<thead>
<tr>
<th><em>example.com</em></th>
<th>Building 1</th>
<th>1st Floor</th>
<th>Alice’s printer</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>example.com</em></td>
<td>Building 2</td>
<td><em>2nd Floor</em></td>
<td>Bob’s printer</td>
</tr>
<tr>
<td><em>example.com</em></td>
<td>Building 3</td>
<td>3rd Floor</td>
<td>Charlie’s printer</td>
</tr>
<tr>
<td>Building 4</td>
<td>Building 5</td>
<td>4th Floor</td>
<td></td>
</tr>
<tr>
<td>Building 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. As in the rest of DNS-Based Service Discovery, all names are represented as-is using plain UTF-8 encoding, and, as described in Section 5.5.4, no text encoding translations are performed.

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.

As an alternative to using VLAN technology, using a Multicast DNS Discovery Relay [Relay] is another way that a Discovery Proxy can have a ‘virtual’ presence on a remote link.
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 5.6.

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 IPv4 subnet address(es) and IPv6 prefix(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. Two varieties of such Domain Enumeration PTR records exist; those with names derived from the domain name communicated to the clients via DHCP, and those with names derived from IPv4 subnet address(es) and IPv6 prefix(es) in use by the clients. Below is an example showing the name-based variety:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>PTR</th>
<th>Host Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>b._dns-sd._udp.example.com.</td>
<td>PTR</td>
<td>Building 1.example.com.</td>
</tr>
<tr>
<td>c._dns-sd._udp.example.com.</td>
<td>PTR</td>
<td>Building 2.example.com.</td>
</tr>
<tr>
<td>e._dns-sd._udp.example.com.</td>
<td>PTR</td>
<td>Building 4.example.com.</td>
</tr>
<tr>
<td>db._dns-sd._udp.example.com.</td>
<td>PTR</td>
<td>Building 1.example.com.</td>
</tr>
<tr>
<td>lb._dns-sd._udp.example.com.</td>
<td>PTR</td>
<td>Building 1.example.com.</td>
</tr>
</tbody>
</table>
```

The meaning of these records is defined in the DNS Service Discovery specification [RFC6763] but for convenience is repeated here. The "b" ("browse") records tell the client device the list of browsing domains to display for the user to select from. The "db" ("default browse") record tells the client device which domain in that list should be selected by default. The "db" domain MUST be one of the domains in the "b" list; if not then no domain is 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, at the time of writing). 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.

Note that in the example above, for clarity, space characters in names are shown as actual spaces. If this data is manually entered
into a textual zone file for authoritative server software such as BIND, care must be taken because the space character is used as a field separator, and other characters like dot (’.’), semicolon (’;’), dollar (’$’), backslash (’\’), etc., also have special meaning. These characters have to be escaped when entered into a textual zone file, following the rules in Section 5.1 of the DNS specification [RFC1035]. For example, a literal space in a name is represented in the textual zone file using ‘\032’, so "Building 1.example.com." is entered as "Building\0321.example.com."

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

In the case where Discovery Proxy functionality is widely deployed within an enterprise (either by having a Discovery Proxy on each link, or by having a Discovery Proxy with a remote ‘virtual’ presence on each link using VLANs or Multicast DNS Discovery Relays [Relay]) this 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 do 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 accommodate 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). The restricted names may be any valid name consisting of only letters, digits, and hyphens, including Punycode-encoded names [RFC3492].
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:

```
prnt.bldg1.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 LDR 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.

As described above in Section 5.2.1, for clarity, space characters in names are shown as actual spaces. If this data were to be manually entered into a textual zone file (which it isn’t) then spaces would need to be represented using '\032', so "My Printer._ipp._tcp.Building 1.example.com." would become "My\032Printer._ipp._tcp.Building\0321.example.com."

Note that the '\032' representation does not appear in the network packets sent over the air. In the wire format of DNS messages, spaces are sent as spaces, not as '\032', and likewise, in a graphical user interface at the client device, spaces are shown as spaces, not as '\032'.

Cheshire Expires September 19, 2018 [Page 15]
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.

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.

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.

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 "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 IPv4 or IPv6 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:

\[
2.113.0.203.in-addr.arpa.\text{PTR} \text{prnt.local}.
\]

into this Unicast DNS response:

\[
2.113.0.203.in-addr.arpa.\text{PTR} \text{prnt.bldg1.example.com}.
\]

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.

Note that this design assumes that all addresses on a given IPv4 subnet or IPv6 prefix are mapped to hostnames using the Discovery Proxy mechanism. It would be possible to implement a Discovery Proxy that can be configured so that some address-to-name mappings are performed using Multicast DNS on the local link, while other address-to-name mappings within the same IPv4 subnet or IPv6 prefix are configured manually.
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 into 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 neighboring 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 distant 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 [DNS-LLQ] or DNS Push Notification subscriptions [Push]) the DNS TTLs reported in the resulting Unicast DNS response MUST 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 [DNS-LLQ] or DNS Push Notifications [Push], 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 A and AAAA records for IPv4 link-local addresses [RFC3927] and IPv6 link-local addresses [RFC4862] 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 SHOULD 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

Multicast DNS devices do not routinely announce their records on the network. Generally they remain silent until queried. This means that the complete set of Multicast DNS records in use on a link can only be discovered by active querying, not by passive listening. Because of this, a Discovery Proxy can only know what names exist on a 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 programmatically generate the traditional NSEC [RFC4034] and NSEC3 [RFC5155] 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 with a Type Bit Map signifying which record types do and do not exist for just the specific name queried, and no other names.

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 encoding [RFC3492] and UTF-8 encoding [RFC3629], 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.

To avoid the need for application-specific knowledge about the semantics of particular TXT record keys, protocol designers are advised to avoid placing link-local names or link-local IP addresses in TXT record keys, if translation of those names or addresses would be required for off-link operation. 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) [DNS-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 Notifications, 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 Notifications (used for long-lived service browsing PTR queries) or not (used for one-shot operations like SRV or address record queries). Note that queries using LLQ or DNS Push Notifications are received directly from the client. Queries not using LLQ or DNS Push Notifications 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 Notifications; 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 MUST be capped to at most ten seconds. (Reasoning: Queries not using LLQ or Push Notifications are generally queries that that expect an answer from only one device, so the first response is also the only response.)
- Not using LLQ or Push Notifications; at least one answer in cache:
  Send response right away to minimise delay.
  DNS TTLs in responses MUST be capped to at most ten seconds.
  No local mDNS queries are performed.
  (Reasoning: Queries not using LLQ or Push Notifications are generally queries that that expect an answer from only one device. 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 Notifications; 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 Notifications).
  (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 Notification state, and if mDNS answers are received later, LLQ or Push Notification messages are sent.
  DNS TTLs in responses are returned unmodified.

- Using LLQ or Push Notifications; 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 Notification state, and results in transmission of mDNS queries, with appropriate Known Answer lists, to determine if further answers are available. If additional mDNS answers are received later, LLQ or Push Notification 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 [DNS-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 relevant zones. They do not exist in the corresponding ".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 is not applicable to Discovery Proxy.

7.2. NSEC and NSEC3 Records

In DNSSEC NSEC [RFC4034] and NSEC3 [RFC5155] 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 of 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 IPv4-only host and an IPv6-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 IPv4 and one for IPv6.
Since for practical purposes, a group of IPv4-only hosts and a group
of IPv6-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 IPv4-only and the other IPv6-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 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 privacy issue is particularly serious for the IPv4 and IPv6 reverse zones. If the public delegation of the reverse zones points to the Discovery Proxy, and the Discovery Proxy is reachable globally, then it could leak a significant amount of information. Attackers could discover hosts that otherwise might not be easy to identify, and learn their hostnames. Attackers could also discover the existence of links where hosts frequently come and go.

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, resulting in denial of service to other legitimate 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. IANA Considerations

This document has no IANA Considerations.

11. 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, Ben Campbell, Tim Chown, Alissa Cooper, Spencer Dawkins, Ralph Droms, Joel Halpern, Ray Hunter, Joel Jaeggli, Warren Kumari, Ted Lemon, Alexey Melnikov, Kathleen Moriarty, Tom Pusateri, Eric Rescorla, Adam Roach, David Schinazi, Markus Stenberg, Dave Thaler, and Andrew Yourtchenko for their comments.
12. References

12.1. Normative References


12.2. Informative References


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


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.

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 andDeployed

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 walk-through giving the details of this particular specific example is given in Appendix A of the Roadmap document [Roadmap].

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

Implementations of the translating/filtering Discovery Proxy specified in this document 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.

Author’s Address

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

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

This document proposes a device pairing mechanism that establishes a relation 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 https://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 October 22, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents
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 obtain information that could be used to identify the host. The necessary relation 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. This pairing system can provide the pairing secrets used in DNSSD Privacy Extensions [I-D.ietf-dnssd-privacy].
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 authentication involving an SAS (short authentication string) to proof the authenticity of the pairing data.

The design of this protocol is based on the analysis of pairing protocols issues presented in [I-D.ietf-dnssd-pairing-info] and in [K17].

Many pairing scenarios involve cell phones equipped with cameras capable of reading a QR code. In these scenarios, scanning QR codes might be more user friendly than selecting names or reading short authentication strings from on screen menus. An optional use of QR codes in pairing protocols is presented in Section 3.

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.

The original version of this document was organized in two parts. The first part presented the pairing need, the list of requirements that shall be met. This first part was informational in nature. The second part composed 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.

The DNS-SD working group approved this split during its meeting in Prague in July 2017. This version of the document implements the split, only retaining the specification part.

2. Protocol Specification

In the proposed pairing protocol, we will consider the device that initiates the pairing as the "client" and the device that responds as the "server". The server will publish a "pairing service". The client will discover the service instance during the discovery phase, as explained in Section 2.1. The pairing service itself is specified in Section 2.3.

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

2.1. 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.

When the pairing service starts, the server will advertise the pairing service according to DNS-SD [RFC6763] over mDNS [RFC6762]. In conformance with DNS-SD, the service is described by an SRV record and by an empty TXT record. These records will be organized as follows:

1. The pairing service is identified in DNS-SD as "_pairing._tcp".

2. The instance name will be a text chosen by the server. It MAY be a random string if the server does not want to advertise its identity in the local environment, or the user friendly name of the server in other cases.

3. The priority and weight fields of the SRV record SHOULD be set according to [RFC6763].

4. The host name MUST be set to the host name advertised by the server in mDNS. The server MAY use a randomized host name as
explained in [I-D.ietf-dnssd-privacy], provided that this name is
properly published in mDNS.

5. The port number MUST be set to the number at which the server is
listening for the pairing service. This port number SHOULD be
randomly picked by the server.

The discovery proceeds as follows:

1. The server advertises an instance of the above described pairing
service and displays its instance name on the server’s screen.

2. The client discovers all the instances of the pairing service
available on the local network. This may result in the discovery
of several instance names.

3. Among these available instance names, 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 record of the
selected instance, retrieves the corresponding server’s A (or
AAAA) record, and establishes the connection.

2.2. Agreement on a Shared Secret

Once the server has been selected at the end of the discovery phase,
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 TLS and a key exporter.

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).
The secret "S_p" will be authenticated in the authentication part of the protocol.

2.3.  Authentication

The pairing protocol implemented on top of TLS allows the users to authenticate the shared secret established in the "Agreement" phase, and to minimize the risk of interference by a third party like a "man-in-the-middle". 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;
```

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 MUST be the same algorithm as negotiated for building the PRF in the TLS connection. 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:

```c
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:

```c
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:

$$SAS = \text{first 20 bits of } \text{HMAC}_\text{hash}(S_p, R_c \ || \ R_s)$$

Where "$\text{HMAC}_\text{hash}$" 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 7 digit decimal integer, including leading zeroes, and ask the user to compare the values. 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".

If the values do match and both users agree, the protocol continues with the exchange of names, both server and client announcing their own preferred name in a Success message.

```c
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.

3. Optional Use of QR Codes

When 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. The agreement process can also be streamlined by the scanning of a second QR code.

3.1. Discovery Using QR Codes

If QR code scanning is available as out-of-band channel, the discovery data is directly transmitted via QR codes instead of DNS-SD over mDNS. Leveraging QR codes, the discovery proceeds as follows:

1. The server displays a QR code containing the connection data otherwise found in the SRV and A or AAAA records: IPv4 or IPv6 address, port number, and optionally host name.

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

[[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 fixed name as instance name.]

3.2. Agreement with QR Codes

When QR codes are available, the agreement on a shared secret proceeds exactly as in the general case.

3.3. Authentication with QR Codes

The availability of QR codes does not change the required network messages or the computation of the SAS, which will performed exactly as specified in Section 2.3, but when QR codes are supported, the SAS may also be represented as QR code.

In the general case, 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.

Once user agreement has been obtained, the protocol continues as in the general case presented in Section 2.3.

4. 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 2.1 and the authentication procedures specified in Section 2.3 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.
5. IANA Considerations

This draft does not require any IANA action.

6. Acknowledgments

We would like to thank Steve Kent and Ted Lemon for their detailed reviews of this document, and for their advice on how to improve it.

7. References

7.1. Normative References


7.2. Informative References

[I-D.ietf-dnssd-pairing-info]

[I-D.ietf-dnssd-privacy]

Authors' Addresses

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

Email: huitema@huitema.net

Daniel Kaiser
Esch-sur-Alzette  4360
Luxembourg

Email: daniel@kais3r.de
Device Pairing Design Issues

draft-ietf-dnssd-pairing-info-01

Abstract

This document discusses issues and problems occurring in the design of device pairing mechanism. It presents experience with existing pairing systems and general user interaction requirements to make the case for "short authentication strings". It then reviews the design of cryptographic algorithms designed to maximise the robustness of the short authentication string mechanisms, as well as implementation considerations such as integration with TLS.

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 https://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 October 22, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must
Table of Contents

1. Introduction ................................................. 2
   1.1. Document Organization ................................. 3
2. Protocol Independent Secure Pairing ............................ 3
3. Identity Assurance ........................................... 4
4. Manual Authentication ......................................... 4
   4.1. Short PIN Proved Inadequate ........................... 4
   4.2. Push Buttons Just Work, But Are Insecure .............. 5
   4.3. Short Range Communication ............................. 5
   4.4. Short Authentication Strings .......................... 6
   4.5. Revisiting the PIN versus SAS discussion ............. 7
5. Resist Cryptographic Attacks ................................... 8
6. Privacy Requirements ......................................... 10
7. Using TLS ................................................... 11
8. QR codes .................................................. 12
9. Intra User Pairing and Transitive Pairing .................... 14
10. Security Considerations .................................... 14
11. IANA Considerations ......................................... 14
12. Acknowledgments ............................................ 15
13. Informative References ..................................... 15
Authors’ Addresses ............................................... 16

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.

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 will guide the design of a pairing protocol.

This document does not specify an actual pairing protocol, but it served as the basis for the design of the pairing protocol developed for DNS-SD privacy [I-D.ietf-dnssd-pairing].
1.1. Document Organization

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

This document results from a split of an earlier pairing draft that contained two parts. The first part, presented the pairing need, and the list of requirements that shall be met. The second part presented the design 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.

The working group approved this split.

2. Protocol Independent Secure Pairing

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.

In this document we provide background and discuss the design of a manually authenticated pairing protocol that is independent of the underlying network protocol stack. We discuss (1) means allowing the two parties engaged in the pairing to discover each other in an existing unsecured network -- e.g. means for learning about the network parameters of the respective other device -- which allows them to establish a connection; (2) agreeing on a shared secret via this connection; and (3) manually authenticating this secret. For our discussion and our secure pairing protocol specification [I-D.ietf-dnssd-pairing], we assume an IP based unsecured network. With little adaption, this pairing mechanism can be used on other protocol stacks as well.

We limit the goal of the protocol to the establishment of a shared secret between two parties. Once that secret has been established, it can trivially be used to secure the exchange of other informations, such as for example public keys and certificates.
3. 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.

4. Manual Authentication

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. Because of this, the protocol requires specific features to detect Man-in-the-Middle attacks, and if possible resist them.

This section discusses existing techniques that are used in practice for manually authenticating a Diffie-Hellman key exchange, and Section 5 provides a layman description of the MiTM problem and countermeasures. A more in depth exploration of manually authenticated pairing protocols may be found in [NR11] and [K17].

4.1. 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 read that PIN and provided it to the other device. The PIN was then 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 had 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.

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.

4.2. 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.

4.3. Short Range Communication

Many pairing protocols that use out-of-band channels have been defined. 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:

- QR codes, displayed on the screen of one device, and read by the camera of the other device.
Near Field Communication (NFC) systems, which provides wireless communication with a very short range.

Sound systems, in which one system 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.

The pairing protocols should not rely on the secrecy of the out-of-band channels; most of these out-of-band channels do not provide confidentiality. QR codes could be read by third parties. Powerful radio antennas might be able to interfere with NFC. Sensitive microphones might pick the sounds. However, a property that all of these channels share is authenticity, i.e. an assurance that the data obtained over the out-of-band channel actually comes from the other party. This is because these out-of-band channels involve the user transmitting information from one device to the other. We will discuss the specific case of QR codes in Section 8.

4.4. Short Authentication Strings

The evolving pairing protocols seem to converge towards using Short Authentication Strings and verifying them via the "compare and confirm" method. This is in line with academic studies, such as [KFR09] or [USK11], and, from the users’ perspective, results in a very simple interaction:

1. Alice and Bob compare displayed strings that represent a fingerprint of the afore exchanged pairing key.

2. 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 number. 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".

4.5. Revisiting the PIN versus SAS discussion

In Section 4.1 we presented the drawbacks of using short pins. One could object that many of the technical issues could be overcome by use of better PAKE algorithms, or by supporting longer PIN. And one could also argue that if PIN based pairing algorithms suffer from failure modes such as static PIN configuration, SAS based protocols are vulnerable to SAS bypass.

The SAS bypass argument is rooted in the psychology of users. In practice, pairing processes can be stressful. The user has to discover on each device the proper combination of key entries that brings up the required pairing UI, will be anxious and eager to complete the procedure, and may well be predisposed to click "OK" in the final stage of the algorithm without actually verifying the SAS. Some users may bypass the required comparison step, because they just want to be done with the pairing.

An advantage of PIN based processes is that they cannot be bypassed. The user must enter the PIN before continuing. Also, once the PIN is entered, everything is automatic. The user does not need to input more data, or press any additional button. PIN based protocols would be a great fit for the QR-code based interaction. One device would display a QR code that contains the PIN. Once the QR code is scanned by the other device, the process is automated.

QR based PIN entry may be user friendly, but one of the arguments developed in Section 4.1 still holds. Let’s assume that an adversary somehow obtains the PIN, maybe by scanning the QR code at a distance. That adversary could mount MITM or impersonation attacks, and compromise the pairing process. It is thus very important to ensure that the PIN is only readable by the user doing the pairing.

We could also argue that the SAS bypass failure mode may be mitigated by specific user designs. For example, instead of just clicking OK, the user could be required to enter the SAS displayed by the other device. This requires about the same interactions as a PIN based process, and it would be slightly safer because the SAS does not have to be kept secret once the keys have been exchanged.

If we summarize the debate, we see that both SAS and PIN based solutions have failure modes depending on implementations. In the SAS mode, the failure happens when the UI does not force the user to copy the PIN and relies on a simple "OK to continue" dialog. In the PIN mode, the failure happens when the device fails to generate a
random PIN for each session, and comes pre-programmed with a simple static PIN of "0000" or "0001".

5. 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:

Alice                       Bob
\[ g^x_A \rightarrow \quad \leftarrow g^x_B \]
\[ n_A \rightarrow \quad \leftarrow n_B \]

Computes              Computes
\[ s = g^x_{AxB} \quad s = g^x_{AxB} \]
\[ h = \text{hash}(s|n_A|n_B) \quad h = \text{hash}(s|n_A|n_B) \]

Displays short Displays short
version of h version of h

If the two short hashes match, Alice and Bob are supposedly assured that they have computed the same secret, but there is a problem. Let’s redraw the same message flow, this time involving the attacker Eve:

Alice                Eve                Bob
\[ g^x_A \rightarrow \quad g^x_{A'}\rightarrow \quad \leftarrow g^x_B \]
\[ n_A \rightarrow \quad n_A \rightarrow \quad \leftarrow n_B \]
Picks n_B’ smartly

Computes              Computes
\[ s' = g^x_{AxB'} \quad s'' = g^x_{A'xB} \]
\[ h' = \text{hash}(s'|n_A|n_B') \quad h'' = \text{hash}(s''|n_A|n_B) \]

Displays short Displays short
version of h’ version of h’

In order to pick a nonce n_B’ that circumvents this naive security measure, Eve runs the following algorithm:
\[
s' = g^{xA\times B'} \\
s'' = g^{xA'\times B} \\
\text{repeat} \\
\quad \text{pick a new version of } nB' \\
\quad h' = \text{hash}(s'|nA|nB') \\
\quad h'' = \text{hash}(s''|nA|nB) \\
\quad \text{until the short version of } h' \\
\quad \text{matches the short version of } h'' \\
\]

Running this algorithm will take \( O(2^b) \) iterations on average (assuming a uniform distribution), where \( b \) is the bit length of the SAS. Since hash algorithms are fast, 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.

Eve could also utilize the fact that she may freely choose the whole input for the hash function and thus choose \( g^{xA'} \) and \( g^{xB'} \) so that an arbitrary collision (birthday attack) instead of a second preimage is sufficient for fooling Alice and Bob.

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
\[ g^{xA} \rightarrow \quad \leftarrow g^{xB} \]

Computes               Computes
\[ s = g^{xA\times B} \quad s = g^{xA\times B} \]
\[ h_a = \text{hash}(s|nA) \rightarrow \quad \leftarrow nB \]
\[ nA \rightarrow \quad \text{verifies } h_a == \text{hash}(s|nA) \]
Computes               Computes
\[ h = \text{hash}(s|nA|nB) \quad h = \text{hash}(s|nA|nB) \]
Displays short        Displays short
version of \( h \)        version of \( h \)

Alice will only disclose \( nA \) after having confirmation from Bob that \( \text{hash}(nA) \) 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 than n one-shot attacks.

There is still a theoretical problem, if Eve has somehow managed to "crack" the hash function. We can build "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. The simplest design uses a 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 likelihood of such padding.

6. 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 user-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 4, 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 the 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.

7. 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 comparing a "short authentication string" (SAS). As explained in Section 5, the secure comparison requires a "commit before disclose" mechanism.

We have three possible designs: (1) create a pairing algorithm from scratch, specifying our own cryptographic protocol; (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.

8. QR codes

In Section 4.3, 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 provides 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:

```
mmmmmmmm m m mmmmmm
 # mmm # # "m" # mmm #
 # *** # m" " # *** #
#mnnmnn# # m m #mnmnmn#
 mm m mm"## m mnm mm
 " #"mm m" # #"m" #
 #"mnm m" m" "m" #m
mnmnmnm #mnm#mnm# m
 # mmm # "mm" # "
 # ### # " m # ### #
 #mnmnm# # "m"m m m
```

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

```
mmmmmmmm m mm mmmmmm
 # mmm # "##m" # mmm #
 # ### # " # # ### #
#mnmnm# # m"m #mnmnm#
 mnmnm mnm"# m m mm m
 #"m mmm"##"##m m#m
" #mnmnm"m###"#m # m
mnmnmnm # "m"m "m"#m
 # mmm # #mmm m " #
 # ### # #mm"## m#
 #mnmnm# #nm#""m m
```

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.

9. Intra User Pairing and Transitive Pairing

There are two usage modes for pairing: inter-user, 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.

10. Security Considerations

This document lists a set of security issues that have to be met by pairing protocols, but does not specify any protocol.

11. IANA Considerations

This draft does not require any IANA action.
12. Acknowledgments

We would like to thank Steve Kent for a detailed early review of an early draft of this document. Both him and Ted Lemon were influential in the decision to separate the analysis of pairing requirements from the specification of pairing protocol in [I-D.ietf-dnssd-pairing]

13. Informative References


[I-D.ietf-dnssd-pairing]

[I-D.ietf-dnssd-privacy]

[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.


Authors' Addresses

Daniel Kaiser
Esch-sur-Alzette 4360
Luxembourg

Email: daniel@kais3r.de
Internet-Draft            Device Pairing Issues               April 2018

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

Email: huitema@huitema.net
Privacy Extensions for DNS-SD

draft-ietf-dnssd-privacy-04

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

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 https://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 October 20, 2018.
Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 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 .......... 7

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 ........................... 9
3.2.2. Using a Predictable Nonce ............................ 9
3.2.3. Using a Short Proof .................................. 10
3.2.4. Direct Queries ....................................... 12
3.3. Private Discovery Service .............................. 12

3.3.1. A Note on Private DNS Services .................... 13
3.4. Randomized Host Names .................................. 14
3.5. Timing of Obfuscation and Randomization ............... 14

4. Private Discovery Service Specification .................... 14

4.1. Host Name Randomization ................................ 15
4.2. Device Pairing .......................................... 15
4.3. Private Discovery Server ............................... 15

4.3.1. Establishing TLS Connections ....................... 15
4.4. Publishing Private Discovery Service Instances ....... 16
4.5. Discovering Private Discovery Service Instances ..... 17
4.6. Direct Discovery of Private Discovery Service Instances . 18
4.7. Using the Private Discovery Service .................. 19

5. Security Considerations ..................................... 19

5.1. Attacks Against the Pairing System .................... 19
5.2. Denial of Discovery of the Private Discovery Service .. 19
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. Parts of the published information can seriously breach the user’s privacy. 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 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 listening 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 both the device, and, by extension, the device’s owner.

2.3. Privacy Implication of Publishing Service Attributes

The TXT record’s attribute-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.

A sometimes heard argument is that devices providing services can be identified by observing the local traffic, and that trying to hide the presence of the service is futile. This argument, however, does not carry much weight because

1. proving privacy at the discovery layer is of the essence for enabling automatically configured privacy-preserving network applications. Application layer protocols are not forced to leverage the offered privacy, but if device tracking is not prevented at the deeper layers, including the service discovery layer, obfuscating a certain service’s protocol at the application layer is futile.

2. Further, even if the application layer does not protect privacy, it is hard to record and analyse the unicast traffic (which most
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] and [K17], 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 utilize [I-D.ietf-dnssd-pairing] as the 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 browsed using regular DNS-SD procedures, and then filtered so that only instances offered by 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.

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 initially proposed a value of about 5 minutes, which would work well for the mDNS variant of DNS-SD. However, this may cause an excessive number of updates for the DNS server based version of DNS-SD. We propose to set a value of about 30 minutes, which seems to be a reasonable compromise.
Receivers can pre-calculate all the $M$ relevant proofs once per time interval and then establish a mapping from the corresponding instance names to the pairing data in form of a hash table. These $M$ relevant proofs are the proofs resulting from hashing a host’s $M$ pairing keys alongside the current nonce. Each time they receive an instance name, they can test in $O(1)$ time if the received service information is relevant or not.

Unix defines a 32 bit time stamp as the number of seconds elapsed since January 1st, 1970 not counting leap seconds. The most significant 20 bits of this 32 bit number represent the number of 2048 seconds intervals since the epoch. 2048 seconds correspond to 34 minutes and 8 seconds, which is close enough to our design goal of 30 minutes. We will thus use this 20 bit number as nonce, which for simplicity will be padded zeroes to 24 bits and encoded in 3 octets.

For coping with time skew, receivers pre-calculate proofs for the respective next time interval and store hash tables for the last, the current, and the next time interval. When receiving a service instance name, receivers first check whether the nonce corresponds to the current, the last or the next time interval, and if so, check whether the instance name is in the corresponding hash table. For (approximately) meeting our design goal of 5 min validity, the last time interval may only be considered if the current one is less than half way over and the next time interval may only be considered if the current time interval is more than half way over.

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 BASE32. 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.

<table>
<thead>
<tr>
<th>Proof</th>
<th>Collisions</th>
<th>BASE64</th>
<th>BASE32</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>5.96046%</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>0.02328%</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>40</td>
<td>0.00009%</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>48</td>
<td>3.6E-09</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>56</td>
<td>1.4E-11</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

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 plain DNS over TLS is the need for an authentication based on pre-shared keys.
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:

```c
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 further published an UDP variant, DNS over DTLS [RFC8094], which would also be a candidate.

DPRIVE and Private Discovery, however, solve two somewhat different problems. While DPRIVE is concerned with the confidentiality of DNS transactions addressing the problems outlined in [RFC7626], 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 host names in the SRV records, nodes could publish randomized host names. 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.

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 the PDS to all paired peers.

Each instance name is composed as follows:

pick a 24 bit nonce, set to the 20 most significant bits of the 32 bit Unix GMT time padded with 4 zeroes.

For example, on August 22, 2017 at 20h 4 min and 54 seconds international time, the Unix 32 bit time had the hexadecimal value 0x599C8E68. The corresponding nonce would be set to the 24 bits: 0x599C80.

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_name = 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. When using the Direct Discovery procedure over mDNS, nodes SHOULD always set the QU-bit (unicast response requested, see [RFC6762] Section 5.4) because responses related to a "_pds._tcp" instance are only relevant for the querying node itself.

When nodes send a DNS-SD discovery request, they will receive in response a series of PTR records, each providing the name of one of the instances present in the scope.

For each time interval, the querier SHOULD pre-calculate a hash table mapping instance names to pairings according to the following conceptual algorithm:

nonce = 20 bit rounded time stamp of the \ respective next time interval padded to \ 24 bits with four zeroes
for each available pairing
  retrieve the key Xj of pairing number j
  compute F = first 48 bits of hash(nonce, Xj)
  construct the binary instance_name as described \ in the previous section
  instance_names[nonce][instance_name] = Xj;

The querier SHOULD store the hash tables for the previous, the current, and the next time interval.

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 72 bits,
      discard the instance.

nonce = first 24 bits of binary.

Check that the 4 least significant bits of the nonce have the value 0, and that the 20 most significant bits of the nonce match the first 20 bits of the current time, or the previous interval (20 bit number minus 1) if the current interval is less than half over, or the next interval (20 bit number plus 1) if the current interval is more than half over. If the nonce does not match an acceptable value, discard the instance.

if ((Xj = instance_names[nonce][instance_name]) != null)
   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 15 minutes.

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 proof, 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 name 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 for protecting the privacy of nodes that offer and query for services. 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.

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.

5.6. Replay attacks and clock synchronization

The mitigation of replay attacks relies on verification of the time encoded in the nonce. This verification assumes that the hosts engaged in discovery have a reasonably accurate sense of the current time.

5.7. Fingerprinting the number of published instances

Adversaries could monitor the number of instances published by a particular device, which in the absence of mitigations will reflect the number of pairings established by that device. This number will probably vary between 1 and maybe 100, providing the adversary with maybe 6 or 7 bits of input in a fingerprinting algorithm.

Devices MAY protect against this fingerprinting by publishing a number of "fake" instances in addition to the real ones. The fake instance identifiers will contain the same nonce as the genuine instance identifiers, and random bits instead of the proof. Peers should be able to quickly discard these fake instances, as the proof will not match any of the values that they expect. One plausible padding strategy is to ensure that the total number of published instances, either fake or genuine, matches one of a few values such as 16, 32, 64, or higher powers of 2.

6. IANA Considerations

This draft does not require any IANA action.
7. Acknowledgments

This draft results from initial discussions with Dave Thaler, and encouragements from the DNS-SD working group members. We would like to thank Stephane Bortzmeyer and Ted Lemon for their detailed reviews of the working draft.

8. References

8.1. Normative References


8.2.  Informative References

[I-D.ietf-dnssd-pairing]

[I-D.ietf-dnssd-push]

[I-D.ietf-tls-tls13]


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
DNS Push Notifications
draft-ietf-dnssd-push-14

Abstract

The Domain Name System (DNS) was designed to return matching records efficiently for queries for data that are relatively static. When those records change frequently, DNS is still efficient at returning the updated results when polled, as long as the polling rate is not too high. 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 https://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 19, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
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 .................... 13
      6.2.1.  SUBSCRIBE Request .............................. 13
      6.2.2.  SUBSCRIBE Response ............................ 16
   6.3.  DNS Push Notification Updates ....................... 19
      6.3.1.  PUSH Message .................................. 19
   6.4.  DNS Push Notification UNSUBSCRIBE .................. 22
      6.4.1.  UNSUBSCRIBE Request ......................... 22
   6.5.  DNS Push Notification RECONFIRM ..................... 24
      6.5.1.  RECONFIRM Request ........................... 24
      6.5.2.  RECONFIRM Response ........................... 26
   6.6.  Client-Initiated Termination ......................... 28
7.  Security Considerations .................................. 29
   7.1.  Security Services .................................. 29
   7.2.  TLS Name Authentication ............................ 29
   7.3.  TLS Compression .................................... 30
   7.4.  TLS Session Resumption ............................. 30
8.  IANA Considerations ...................................... 30
9.  Acknowledgements ........................................ 31
10. References ................................................ 31
    10.1.  Normative References ............................. 31
    10.2.  Informative References ........................... 33
Authors’ Addresses .......................................... 35
1. Introduction

Domain Name System (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", when, and only when, they appear in all capitals, as shown here [RFC2119] [RFC8174].
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 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 deliver 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 record changes are sent to that multicast group address 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) mechanism [LLQ] is an existing deployed solution to provide asynchronous change notifications, used by Apple’s Back to My Mac Service [RFC6281] introduced in Mac OS X 10.5 Leopard in 2007. Back to My Mac was designed in an era when the data center operations staff asserted that it was impossible for a server to handle large numbers of mostly-idle TCP connections, so LLQ was 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 DNS Stateful Operations (DSO) [DSO] running over TLS over 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. Instead of inventing a new vocabulary of messages to communicate DNS zone changes as LLQ did, this specification borrows the established 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 receiving further 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 [RFC7719]. 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. If a query is received with the RD bit set, matching records for names falling within the server’s zones should be returned with the RA (Recursion Available) bit clear. If the query is for a name not in the server’s zone, an error with RCODE NOTAUTH (Not Authoritative) should be returned.

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 unilaterally 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". If messages are received for a class other than "IN", and that class is not supported, an error with RCODE NOTIMPL (Not Implemented) should be returned.

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. Specifically:

(a) 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.

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

4. Transport

Other DNS operations like 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 DNS Stateful Operations (DSO) [DSO] running over TLS over TCP [RFC7858].

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 [SYN] [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, and 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.
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 session 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 sessions to alternate DNS servers that support DNS Push Notifications for the zone and distribute subscriptions at the client’s 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 Stateful Operations (DSO) [DSO].

For details of the DSO message format refer to the DNS Stateful Operations specification [DSO]. Those details are not repeated here.

DNS Push Notification clients and servers MUST support DSO, but (as stated in the DSO specification [DSO]) the server SHOULD NOT issue any DSO messages until after the client has first initiated an acknowledged DSO message of its own. A single server can support DNS Queries, DNS Updates, and DNS Push Notifications (using DSO) on the same TCP port, and until the client has sent at least one DSO message, the server does not know what kind of client has connected to it. Once the client has indicated willingness to use DSO by sending one of its own, either side of the session may then initiate further DSO messages 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 DSO 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 session first, and only issue a DSO Keepalive operation later if it determines that to be necessary.

Once the session 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.
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 begins by opening a DSO Session to its normal configured DNS recursive resolver and requesting a Push Notification subscription. If this is successful, then the recursive resolver will make appropriate Push Notification subscriptions on the client’s behalf, and the client will receive appropriate results. If the recursive resolver does not support Push Notification subscriptions, then it will return an error code, and the client should proceed to discover the appropriate server for direct communication. 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].

The algorithm described here is an iterative algorithm, which starts with the full name of the record to which the client wishes to subscribe. Successive SOA queries are then issued, trimming one label each time, until the closest enclosing authoritative server is discovered. There is also an optimization to enable the client to take a "short cut" directly to the SOA record of the closest enclosing authoritative server in many cases.

1. The client begins the discovery by sending a DNS query to its local resolver, with record type SOA [RFC1035] for the record name to which it wishes to subscribe. As an example, suppose the client wishes to subscribe to PTR records with the name _ipp._tcp.foo.example.com (to discover Internet Printing Protocol (IPP) printers [RFC8010] [RFC8011] being advertised at "foo.example.com"). The client begins by sending an SOA query for _ipp._tcp.foo.example.com to the local recursive resolver. The goal is to determine the server authoritative for the name _ipp._tcp.foo.example.com. The DNS zone containing the name _ipp._tcp.foo.example.com could be example.com, or foo.example.com, or _tcp.foo.example.com, or even _ipp._tcp.foo.example.com. The client does not know in advance where the closest enclosing zone cut occurs, which is why it uses the procedure described here to discover this information.

2. If the requested SOA record exists, it will be returned in the Answer section with a NOERROR response code, and the client has succeeded in discovering the information it needs. (This text is not placing any new requirements on DNS recursive resolvers. It
is merely describing the existing operation of the DNS protocol
[RFC1034] [RFC1035].)

3. If the requested SOA record does not exist, the client will get
back a NOERROR/NODATA response or an NXDOMAIN/Name Error
response. In either case, the local resolver SHOULD include the
SOA record for the zone of the requested name in the Authority
Section. If the SOA record is received in the Authority Section,
then the client has succeeded in discovering the information it
needs. (This text is not placing any new requirements on DNS
recursive resolvers. It is merely describing the existing
operation of the DNS protocol regarding negative responses
[RFC2308].)

4. If the client receives a response containing no SOA record, then
it proceeds with the iterative approach. The client strips the
leading label from the current query name and 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, repeating the
iterative search until either an SOA is received, or the query
name is empty. In the case of an empty name, this is a network
configuration error which should not happen 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.

5. 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.

6. 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. (The "_dns-push-tls._tcp" service type is
allocated by IANA for this purpose, and, like any allocated IANA
service type, MUST NOT be used for other services. Other
services that require an IANA service type should use a unique
service type allocated by IANA for that service [RFC6335][ST].)

7. If the zone in question is set up to offer DNS Push Notifications
then this SRV record MUST exist. (If this SRV record does not
exist then the zone is not correctly configured for DNS Push
Notifications as specified in this document.) 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
8. 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 willing to accept a subscription request, or is not reachable within a reasonable time, as determined by the client, then a subsequent server is to be contacted.

Each time a client makes a new DNS Push Notification subscription session, it SHOULD repeat the discovery process in order to determine the preferred DNS server for subscriptions at that time. However, 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 DSO session to the server. A SUBSCRIBE request is encoded in a DSO [DSO] message. This specification defines a DSO TLV for DNS Push Notification SUBSCRIBE Requests/Responses (tentatively DSO Type Code 0x40).

The entity that initiates a SUBSCRIBE request is by definition the client. A server MUST NOT send a SUBSCRIBE request over an existing session from a client. If a server does send a SUBSCRIBE request over a DSO session initiated by a client, this is a fatal error and the client should immediately abort the connection with a TCP RST (or equivalent for other protocols).

6.2.1. SUBSCRIBE Request

A SUBSCRIBE request begins with the standard DSO 12-byte header [DSO], followed by the SUBSCRIBE TLV. A SUBSCRIBE request message is illustrated in 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 session. For the purposes here, a MESSAGE ID is in use on this session if the client has used it in a request for which it has not yet received a response, or 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.

The other header fields MUST be set as described in the DSO specification [DSO]. The DNS Opcode is the DSO Opcode (tentatively 6). The four count fields MUST be zero, and the corresponding four sections MUST be empty (i.e., absent).

The DSO-TYPE is SUBSCRIBE (tentatively 0x40). The DSO-LENGTH is the length of the DSO-DATA that follows, which specifies the name, type, and class of the record(s) being sought.
The DSO-DATA for a SUBSCRIBE request MUST contain exactly one question. The DSO-DATA for a SUBSCRIBE request 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 DSO session between the client and the server is closed.

SUBSCRIBE requests on a given session 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 DSO session. For the purpose of this matching, the established DNS case-insensitivity for US-ASCII letters applies (e.g., "example.com" and "Example.com" are the same). If a server receives such a duplicate SUBSCRIBE message this is an error and the server MUST 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, where 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 DSO 12-byte header [DSO], possibly followed by one or more optional TLVs, such as a Retry Delay 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 include a SUBSCRIBE TLV. If a client receives a SUBSCRIBE response message containing a SUBSCRIBE TLV then the response message is processed but the SUBSCRIBE TLV MUST be silently ignored.

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOERROR</td>
<td>0</td>
<td>SUBSCRIBE successful.</td>
</tr>
<tr>
<td>FORMERR</td>
<td>1</td>
<td>Server failed to process request due to a malformed request.</td>
</tr>
<tr>
<td>SERVFAIL</td>
<td>2</td>
<td>Server failed to process request due to a problem with the server.</td>
</tr>
<tr>
<td>NXDOMAIN</td>
<td>3</td>
<td>NOT APPLICABLE. DNS Push Notification servers MUST NOT return NXDOMAIN errors in response to SUBSCRIBE requests.</td>
</tr>
<tr>
<td>NOTIMP</td>
<td>4</td>
<td>Server does not implement DSO.</td>
</tr>
<tr>
<td>REFUSED</td>
<td>5</td>
<td>Server refuses to process request for policy or security reasons.</td>
</tr>
<tr>
<td>NOTAUTH</td>
<td>9</td>
<td>Server is not authoritative for the requested name.</td>
</tr>
<tr>
<td>DSOTYPENI</td>
<td>11</td>
<td>SUBSCRIBE operation not supported.</td>
</tr>
</tbody>
</table>

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, that means (a) the client is (at least partially) misconfigured, (b) the server resources are exhausted, or (c) there is some other unknown failure on the server. In any case, the client shouldn’t retry the subscription right away. Either end can terminate the session, 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 TLV [DSO] 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. These recommended values apply both to the default values a server should place in the Retry Delay TLV, and the default values a client should assume if the server provides no Retry Delay TLV.

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) 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. If a more serious server failure occurs, the delay may be longer in accordance with the specific problem encountered.

For RCODE = 4 (NOTIMP), which occurs on a server that doesn’t implement DSO [DSO], it is unlikely that the server will begin supporting DSO in the next few minutes, so the retry delay SHOULD be one hour. Note that in such a case, a server that doesn’t implement DSO is unlikely to place a Retry Delay TLV in its response, so this recommended value in particular applies to what a client should assume by default.

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 session to
remain open, or MAY send a DNS Push Notification Retry Delay
Operation TLV instructing the client to close the session, as
described in the DSO specification [DSO]. 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 DSO 12-byte header [DSO],
followed by the PUSH TLV. A PUSH message is illustrated in Figure 2.

The MESSAGE ID field MUST be zero. There is no client response to a
PUSH message.

The other header fields MUST be set as described in the DSO
specification [DSO]. The DNS Opcode is the DSO Opcode (tentatively
6). The four count fields MUST be zero, and the corresponding four
sections MUST be empty (i.e., absent).

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

The DSO-DATA contains one or more Update records. 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 Update records are
formatted in the customary way for Resource Records in DNS messages.
Update records in a PUSH Message are interpreted according to the
same rules as for DNS Update [RFC2136] messages, namely:

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.
Figure 2: PUSH Message

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 session. 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 session 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 DSO session 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 session-level mechanism, not a subscription-level mechanism.

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 DSO session being closed) record aging resumes and records are removed from the local cache when their TTL reaches zero.
6.4. DNS Push Notification UNSUBSCRIBE

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

A server MUST NOT initiate an UNSUBSCRIBE request. If a server does send an UNSUBSCRIBE request over a DSO session initiated by a client, this is a fatal error and the client should immediately abort the connection with a TCP RST (or equivalent for other protocols).

6.4.1. UNSUBSCRIBE Request

An UNSUBSCRIBE request begins with the standard DSO 12-byte header [DSO], followed by the UNSUBSCRIBE TLV. An UNSUBSCRIBE request message is illustrated in Figure 3.

The MESSAGE ID field MUST be zero. There is no server response to a UNSUBSCRIBE message.

The other header fields MUST be set as described in the DSO specification [DSO]. The DNS Opcode is the DSO Opcode (tentatively 6). The four count fields MUST be zero, and the corresponding four sections MUST be empty (i.e., absent).

In the UNSUBSCRIBE TLV the DSO-TYPE is UNSUBSCRIBE (tentatively 0x42). The DSO-LENGTH is 2 octets.

The DSO-DATA contains the MESSAGE ID field of 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.

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.
Figure 3: UNSUBSCRIBE Request
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 begins with the standard DSO 12-byte header [DSO], followed by the RECONFIRM TLV. A RECONFIRM request message is illustrated in Figure 4.

The MESSAGE ID field MUST be set to a unique value, that the client is not using for any other active operation on this DSO session. For the purposes here, a MESSAGE ID is in use on this session if the client has used it in a request for which it has not yet received a response, or 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.

The other header fields MUST be set as described in the DSO specification [DSO]. The DNS Opcode is the DSO Opcode (tentatively 6). The four count fields MUST be zero, and the corresponding four sections MUST be empty (i.e., absent).

The DSO-TYPE is RECONFIRM (tentatively 0x43). The DSO-LENGTH is the length of the data that follows, which specifies the name, type, class, and content of the record being disputed.
The DSO-DATA for a RECONFIRM request MUST contain exactly one record. The DSO-DATA for a RECONFIRM request 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 DSO 12-byte header [DSO], possibly followed by one or more optional TLVs, such as a Retry Delay TLV. For suggested values for the Retry Delay TLV, see Section 6.2.2.

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 include a DSO RECONFIRM TLV. If a client receives a RECONFIRM response message containing a RECONFIRM TLV then the response message is processed but the RECONFIRM TLV MUST be silently ignored.

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOERROR</td>
<td>0</td>
<td>RECONFIRM accepted.</td>
</tr>
<tr>
<td>FORMERR</td>
<td>1</td>
<td>Server failed to process request due to a malformed request.</td>
</tr>
<tr>
<td>SERVFAIL</td>
<td>2</td>
<td>Server failed to process request due to a problem with the server.</td>
</tr>
<tr>
<td>NXDOMAIN</td>
<td>3</td>
<td>NOT APPLICABLE. DNS Push Notification servers MUST NOT return NXDOMAIN errors in response to RECONFIRM requests.</td>
</tr>
<tr>
<td>NOTIMP</td>
<td>4</td>
<td>Server does not implement DSO.</td>
</tr>
<tr>
<td>REFUSED</td>
<td>5</td>
<td>Server refuses to process request for policy or security reasons.</td>
</tr>
<tr>
<td>NOTAUTH</td>
<td>9</td>
<td>Server is not authoritative for the requested name.</td>
</tr>
<tr>
<td>DSOTYPENI</td>
<td>11</td>
<td>RECONFIRM operation not supported.</td>
</tr>
</tbody>
</table>

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 has exhausted its resources or other serious problem occurred.

RCODE values NOTIMP indicates that the server does not support DSO, and DSO 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 DSOTYPENI indicates that the server does not support RECONFIRM requests.

Nonzero RCODE values SERVFAIL, REFUSED and DSOTYPENI 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 DSO Retry Delay TLV to end the DSO session, as described in the DSO specification [DSO].
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 DSO session. When a client terminates an individual subscription (via UNSUBSCRIBE) or all subscriptions on that DSO session (by ending the session) 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 session via UNSUBSCRIBE, a client MAY close the session immediately, or it may keep it open if it anticipates performing further operations on that session in the future. If a client wishes to keep an idle session open, it MUST respect the maximum idle time required by the server [DSO].

If a client plans to terminate one or more subscriptions on a session and doesn’t intend to keep that session open, then as an efficiency optimization it MAY instead choose to simply close the session, which implicitly terminates all subscriptions on that session. 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 session has been cancelled.

When closing a session, 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 session that it would not want lost (like DNS updates) then the client SHOULD do an orderly disconnect, sending a TLS close_notify followed by 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

The Strict Privacy Usage Profile for DNS over TLS is strongly recommended for DNS Push Notifications as defined in "Authentication and (D)TLS Profile for DNS-over-(D)TLS" [I-D.ietf-dprive-dtls-and-tls-profiles]. The Opportunistic Privacy Usage Profile is permissible as a way to support incremental deployment of security capabilities. Cleartext connections for DNS Push Notifications are not permissible.

DNSSEC is RECOMMENDED for the authentication of DNS Push Notification servers. 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 Server 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 usable, 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.

See Authentication and (D)TLS Profile for DNS-over-(D)TLS [I-D.ietf-dprive-dtls-and-tls-profiles] for more information on authenticating domain names. Also note that a DNS Push server is an authoritative server and a DNS Push client is a standard DNS client. While the terminology in Authentication and (D)TLS Profile for DNS-over-(D)TLS [I-D.ietf-dprive-dtls-and-tls-profiles] explicitly states it does not apply to authoritative servers, it does in this case apply to DNS Push Notification clients and servers.

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 DSO session 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 DSO session. 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 Registry Service Types [RFC6335][ST].

This document defines four DNS Stateful Operations 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, Soraia Zlatkovic, Sara
Dickinson, and Andrew Sullivan.

10. References

10.1. Normative References

[DSO] Bellis, R., Cheshire, S., Dickinson, J., Dickinson, S.,
Mankin, A., and T. Pusateri, "DNS Stateful Operations",
draft-ietf-dnsop-session-signal-05 (work in progress),
January 2018.

Version 1.3", draft-ietf-tls-tls13-26 (work in progress),
March 2018.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768,
DOI 10.17487/RFC0768, August 1980,
<https://www.rfc-editor.org/info/rfc768>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7,
RFC 793, DOI 10.17487/RFC0793, September 1981,
<https://www.rfc-editor.org/info/rfc793>.

STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987,

[RFC1035] Mockapetris, P., "Domain names - implementation and
specification", STD 13, RFC 1035, DOI 10.17487/RFC1035,

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts -
Application and Support", STD 3, RFC 1123,
DOI 10.17487/RFC1123, October 1989,
10.2. Informative References


[I-D.dukkipati-tcpm-tcp-loss-probe]

[I-D.ietf-dprive-dtls-and-tls-profiles]

[LLQ]

[obs]


Authors' Addresses

Tom Pusateri
Unaffiliated
Raleigh, NC  27608
USA

Phone: +1 919 867 1330
Email: pusateri@bangj.com

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

Phone: +1 408 974 3207
Email: cheshire@apple.com
Simple Homenet Naming and Service Discovery Architecture
draft-ietf-homenet-simple-naming-02

Abstract

This document describes how names are published and resolved on homenets, and how hosts are configured to use these names to discover services on homenets. It presents the complete architecture, and describes a simple subset of that architecture that can be used in low-cost homenet routers.

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 https://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction ................................................. 2
2. Requirements ................................................ 3
   2.1. Managed LAN versus Homenet ............................ 4
   2.2. Homenet-specific considerations ...................... 4
3. Terminology .................................................. 5
4. Name ......................................................... 5
5. Authority .................................................... 6
6. Resolution ................................................... 7
7. Publication ................................................... 7
   7.1. DNS Service Discovery Registration Protocol .......... 7
   7.2. Configuring Service Discovery ......................... 8
8. Host Configuration .......................................... 10
9. Globally Unique Name ........................................ 10
10. DNSSEC Validation ........................................... 10
11. Support for Multiple Provisioning Domains ................ 11
12. Using the Local Namespace While Away From Home ........ 11
13. Management Considerations ................................ 11
14. Privacy Considerations ..................................... 12
15. Security Considerations .................................... 12
16. IANA considerations ....................................... 12
17. Normative References ...................................... 12
Appendix A. Existing solutions ............................... 14
Authors’ Addresses ............................................. 15

1. Introduction

This document is a homenet architecture document. The term ‘homenet’
refers to a set of technologies that allow home network users to have
a local-area network (LAN) with more than one physical link and,
on Optionally, more than one internet service provider. Home network
users are assumed not to be knowledgable in network operations, so
homenets automatically configure themselves, providing connectivity
and service discovery within the home with no operator intervention.
This document describes the aspect of homenet automatic configuration
that has to do with service discovery and name resolution.

The homenet naming architecture consists of two parts: the simple
naming architecture, and the advanced naming architecture. The
advanced architecture provides approximate parity of features with a
managed network, including the ability to publish services on the
internet. The simple architecture provides a minimal set of features
required to enable seamless service discovery on a multi-link home

network, but does not attempt to provide feature parity with a managed LAN.

This document begins by presenting a motivational list of requirements and considerations, which should give the reader a clear idea of the scope of the problem being solved. It then explains how each requirement is addressed, and provides references for relevant standards documents describing the details of the implementation. Some requirements are not satisfied by the simple architecture; these are discussed in this document, but explained in more detail in the Advanced Homenet Naming Architecture document, which is to follow.

2. Requirements

Name service on a local area network (LAN) requires the following:

- Name: a forward domain under which information about local services will be published
- Authority: a name server that is authoritative for at least one forward domain and one or two reverse domains that are applicable to that network and is capable of signing and publishing the zones using DNSSEC
- Resolution: a full-service caching DNS resolver that fully supports EDNS(0) and queries with the DO bit set
- Publication: a mechanism that
  - allows services on the LAN to publish information about the services they provide
  - allows services to publish information on how to reach them
  - manages the lifetime of such information, so that it persists long enough to prevent spoofing, but protects end users from seeing stale information
- Host configuration: one or more automatic mechanisms (e.g. DHCP or RA) that provide:
  - caching resolver information to hosts on the LAN
  - information about how services on the LAN can publish information
- Trust: some basis for trusting the information that is provided by the service discovery system
2.1. Managed LAN versus Homenet

A managed network is one that has a (human) manager, or operator. The operator has authority over the network, and the authority to publish names in a forward DNS tree, and reverse names in the reverse tree. The operator has the authority to sign the respective trees with DNSSEC, and acquire TLS certificates for hosts/servers within the network.

On a managed LAN, many of these services can be provided by operators. When a new printer is added to the network, it can be added to the service discovery system (the authoritative server) manually. When a printer is taken out of service, it can be removed. In this scenario, the role of "publisher" is filled by the network operator.

In many managed LANs, establishment of trust for service discovery is simply on the basis of a belief that the local resolver will give a correct answer. Once the service has been discovered and chosen, there may be some security (e.g., TLS) that protects the connection to the service, but the trust model is often just "you’re connected to a network you trust, so you can trust the printer that you discovered on this network."

A homenet does not have an operator, so functions that would normally be performed by the operator have to happen automatically. This has implications for trust establishment—since there is no operator controlling what services are published locally, some other mechanism is required for basic trust establishment. Additionally, whereas in a managed LAN with multiple links to the Internet, the network operator can configure the network so that multihoming is handled seamlessly, in a homenet, multihoming must be handled using multiple provisioning domains [RFC7556].

2.2. Homenet-specific considerations

A naming architecture for homenets therefore adds the following considerations:

- All of the operations mentioned here must reliably function automatically, without any user intervention or debugging.
- Because user intervention cannot be required, naming conflicts must be resolved automatically, and, to the extent possible, transparently.
- 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.

- Homenets must address the problem of multiple provisioning domains, in the sense that the DNS may give a different answer depending on whether caching resolvers at one ISP or another are queried.

An additional requirement from the Homenet Architecture [9] is that hosts are not required to implement any homenet-specific capabilities in order to discover and access services on the homenet. This architecture may define optional homenet-specific features, but hosts that do not implement these features must work on homenets.

3. Terminology

This document uses the following terms and abbreviations:

- HNR Homenet Router
- SHNR Homenet Router implementing simple homenet naming architecture
- AHNR Homenet Router implementing advanced homenet naming architecture
- ISP Internet Service Provider

4. Name

In order for names to be published on a homenet, it is necessary that there be a set of domain names under which such names are published. These domain names, together, are referred to as the "local domains." By default, homenets use the reserved domain 'home.arpa.' for publishing names for forward lookups. So a host called 'example' that published its name on the homenet would publish its records on the domain name 'example.home.arpa.'. Because 'home.arpa.' is used by all homenets, it has no global meaning, and names published under the domain 'home.arpa' cannot be used outside of the homenet on which they are published.

Homenet routers that implement advanced homenet naming may also be configured with a global domain. How such a domain is configured is out of scope for this document, and is described in the Advanced Homenet Naming Architecture document [advanced].

In addition to the name, which defaults to 'home.arpa.', names are needed for reverse lookups. These names are dependent on the IP addressing used on the homenet. If the homenet is addressed with
IPv4, a reverse domain corresponding to the IPv4 subnet [1] section 5.2.1 should be constructed. For example, if the homenet is allocating local IP addresses out of net 10 [3], a domain, ‘10.in-addr.arpa’ would be required. Like ‘home.arpa.’, ‘10.in-addr.arpa’ is a locally-served zone, and has no validity outside of the homenet.

If the homenet is addressed with IPv6, it is expected to have a unique local address prefix; subsets of this prefix will be advertised on every link on the homenet. Every service on the homenet that supports IPv6 is expected to be reachable at an address that is configured using the ULA prefix. Therefore there is no need for any IPv6 reverse zone to be populated other than the ULA zone. So for example if the homenet’s ULA prefix is fd00:2001:db8::/48, then the reverse domain name for the homenet would end in ‘8.b.d.0.1.0.0.2.0.0.d.f.ip6.arpa’.

5. Authority

The authority role is provided by a name server that is authoritative for each of the local domains. SHNRs provide authoritative service for the homenet using DNSSD Discovery Broker [17]. SHNRs also provide Discovery Relay service [12]. On a homenet that has only SHNRs, each SHNR individually provides authoritative service for the whole homenet by using Discovery relays to discover services off the local link.

The Discovery Proxy model relies on each link having its own name. However, homnets 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 without some tweaks. In order to address this, homnets will use Discovery Brokers [17]. The discovery broker will be configured so that a single query for a particular service will be successful in providing the information required to access that service, regardless of the link it is on.

Artificial link names will be generated using HNCP. These should only be visible to the user in graphical user interfaces in the event that the same name is claimed by a service on two links. Services that are expected to be accessed by users who type in names should use [13] if it is available.

It is possible that local services may offer services available on IP addresses in public as well as ULA prefixes. Homnet hybrid proxies MUST filter out global IP addresses, providing only ULA addresses, similar to the process described in section 5.5.2 of [11].

This filtering applies to queries within the homenet; it is appropriate for non-ULA addresses to be used for offering services,
because in some cases end users may want such services to be reachable outside of the homenet. Configuring this is however out of scope for this document.

6. Resolution

Name resolution is provided by a local DNS cache or proxy on the homenet, henceforth the "local resolver." All host queries are sent to this local resolver. The local resolver may either act as a full-service caching resolver, or as a DNS proxy. Its responsibility with respect to queries on the homenet is to notice queries for names for which the local authoritative server is authoritative. Queries for such names are handled through the local authoritative server. Queries for all other names are resolved either by forwarding them to an ISP-provided full service resolver, or by providing the full service resolver function locally.

7. Publication

7.1. DNS Service Discovery Registration Protocol

The DNSSD Service Registration protocol [13] requires that DNS updates be validated on the basis that they are received on the local link. To ensure that such registrations are actually received on local links in the homenet, updates are sent to the local relay proxy ([12]) (XXX how?).

The relay proxy encapsulates the update and sends it to whatever Discovery Proxy is listening on the link; the Discovery proxy then either consumes the update directly, or forwards it to the authoritative resolver for the local service discovery zone. If the registration protocol is not supported on the homenet, the Discovery Proxy rejects the update with a ??? RCODE.

Homenets are not required to support Service Registration. Service registration requires a stateful authoritative DNS server; this may be beyond the capability of the minimal Homenet router. However, more capable Homenet routers should provide this capability. In order to make this work, minimal Homenet routers MUST implement the split hybrid proxy [12]. This enables a Homenet with one or more Homenet routers that provide a stateful registration cache to allow those routers to take over service, using Discovery Relays to service links that are connected using Homenet routers with more limited functionality.
7.2. Configuring Service Discovery

Clients discovering services using DNS-SD [7] follow a two-step process. The first step is for the client device to determine in which domain(s) to attempt to discover services. The second step is for the client device to then seek desired service(s) in those domain(s). For an example of the second step, given the desired service type "IPP Printing", and the domains "local" and "meeting.ietf.org", the client device forms the queries 

"_ipp._tcp.local. PTR ?" (resolved using Multicast DNS) and
"_ipp._tcp.meeting.ietf.org PTR. ?" (resolved using Unicast DNS) and then presents the combined list of results to the user.

The first step, determining in which domain(s) to attempt to discover services, is performed in a variety of ways, as described in Section 11 of the DNS-Based Service Discovery specification [7].

The domain "local" is generally always in the set of domains in which the client devices attempt to discover services, and other domains for service discovery may be configured manually by the user.

The device also learns additional domains automatically from its network environment. For this automatic configuration discovery, special DNS queries are formulated. To learn additional domain(s) in which to attempt to discover services, the query string

"lb._dns_sd._udp" is prepended onto three different kinds of "bootstrap domain" to form DNS queries that allow the device to learn the configuration information.

One of these bootstrap domains is the fixed string "local". The device issues the query "lb._dns_sd._udp.local. PTR ?" (resolved using Multicast DNS), and if any answers are received, then they are added to the set of domains in which the client devices attempt to discover services.

Another kind of these bootstrap domains is name-based, derived from the DHCPv4 "domain name" option (code 15) [4] (for IPv4) or the DNS Search List (DNSSL) Router Advertisement option [10] (for IPv6). If a domain in the DNSSL is "example.com", then the device issues the query "lb._dns_sd._udp.example.com. PTR ?" (resolved using Unicast DNS), and if any answers are received, then they are likewise added to the set of domains in which the client devices attempt to discover services.

Finally, the third kind of bootstrap domain is address-based, derived from the device’s IP address(es) themselves. If the device has IP address 192.168.1.100/24, then the device issues the query

"lb._dns_sd._udp.0.1.168.192.in-addr.arpa. PTR ?" (resolved using
Unicast DNS), and if any answers are received, then they are also added to the set of domains in which the client devices attempt to discover services.

Since there is an HNR on every link of a homenet, automatic configuration could be performed by having HNRs answer the "lb._dns_sd._udp.local. PTR ?" (Multicast DNS) queries. However, because multicast is slow and unreliable on many modern network technologies like Wi-Fi, we prefer to avoid using it. Instead we require that a homenet be configured to answer the name-based bootstrap queries. By default the domain in the DNSSL communicated to the client devices will be "home.arpa", and the homenet will be configured to correctly answer queries such as "lb._dns_sd._udp.example.com. PTR ?", though client devices must not assume that the name will always be "home.arpa". A client could be configured with any valid DNSSL, and should construct the appropriate bootstrap queries derived from the name(s) in their configured DNS Search List.

HNRs will answer domain enumeration queries 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._udp.<domain>" subdomain. By default, the ‘home.arpa’ domain is advertised. Similarly, ‘home.arpa’ is advertised as the default browsing and service registration domain under "db._dns_sd._udp.<domain>", "r._dns_sd._udp.<domain>", "dr._dns_sd._udp.<domain>" and "lb._dns_sd._udp.<domain>".

In order for this discovery process to work, the homenet must provide authoritative answers for each of the domains that might be queried. To do this, it provides authoritative name service for the ‘ip6.arpa’ and ‘in-addr.arpa’ subdomains corresponding to each of the prefixes advertised on the homenet. For example, consider a homenet with the 192.168.1.0/24, 2001:db8:1234:5600::/56 and fc01:2345:6789:1000::/56 prefixes. This homenet will have to provide a name server that claims to be authoritative for 1.168.192.in-addr.arpa, 6.5.4.3.2.1.8.b.d.0.1.0.0.2.ip6.arpa and 0.0.9.8.7.6.5.4.3.2.1.0.c.f.ip6.arpa.

An IPv6-only homenet would not have an authoritative server for a subdomain of in-addr.arpa. These public authoritative zones are required for the public prefixes even if the prefixes are not
delegated. However, they need not be accessible outside of the homenet.

It is out of the scope of this document to specify ISP behavior, but we note that ISPs have the option of securely delegating the zone, or providing an unsigned delegation, or providing no delegation. Any delegation tree that does not include an unsigned delegation at or above the zone cut for the ip6.arpa reverse zone for the assigned prefix will fail to validate.

Ideally, an ISP should provide a secure delegation using a zone-signing key provided by the homenet. However, that too is out of scope for this document. Therefore, an ISP that wishes to support users of the simple homenet naming architecture will have to provide an unsigned delegation. We do not wish, however, to discourage provisioning of signed delegations when that is possible.

8. Host Configuration

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 Recursive DNS Server Option ([10], Section 5.1) 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.

9. Globally Unique Name

Automatic configuration of a globally unique name for the homenet is out of scope for this document. However, homenet servers MUST allow the user to configure a globally unique name in place of the default name, ‘home.arpa.’ By default, even if configured with a global name, homenet routers MUST NOT answer queries from outside of the homenet for subdomains of that name.

10. DNSSEC Validation

DNSSEC Validation for the ‘home.arpa’ 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.
Homenets that have been configured with a globally unique domain MUST support DNSSEC signing of local names, and must provide a way to generate a KSK that can be used in the secure delegation of the globally unique domain assigned to the homenet.

11. Support for Multiple Provisioning Domains

Homenets must support the Multiple Provisioning Domain Architecture [9]. Hosts connected to the homenet may or may not support multiple provisioning domains. For hosts that do not support multiple provisioning domains, the homenet provides one or more resolvers that will answer queries for any provisioning domain. Such hosts may receive answers to queries that either do not work as well if the host chooses a source address from a different provisioning domain, or does not work at all. However, the default source address selection policy, longest-match [CITE], will result in the correct source address being chosen as long as the destination address has a close match to the prefix assigned by the ISP.

Hosts that support multiple provisioning domains will be provisioned with one or more resolvers per provisioning domain. Such hosts can use the IP address of the resolver to determine which provisioning domain is applicable for a particular answer.

Each ISP has its own provisioning domain. Because ISPs connections cannot be assumed to be persistent, the homenet has its own separate provisioning domain.

Configuration from the IPv4 DHCP server are treated as being part of the homenet provisioning domain. The case where a homenet advertises IPv4 addresses from one or more public prefixes is out of scope for this document. Such a configuration is NOT RECOMMENDED for homenets.

Configuration for IPv6 provisioning domains is done using the Multiple Provisioning Domain RA option [CITE].

12. 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.

13. 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.
14. 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 homenet 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 [15] to mitigate this risk.

15. 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.

16. IANA considerations

No new actions are required by IANA for this document.

Note however that this document is relying on the allocation of ‘home.arpa’ described in Special Use Top Level Domain ‘.home.arpa’ [16]. This document therefore can’t proceed until that allocation is done. [RFC EDITOR PLEASE REMOVE THIS PARAGRAPH PRIOR TO PUBLICATION].

17. Normative References


Appendix A. 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. Unfortunately, these solutions do not fully address the requirements of homenets.

For example, Multicast DNS [6] can provide naming and service discovery [7], 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 [5], and a mechanism for discovering services [7]. Unfortunately, DNS provides no mechanism for automatically provisioning new namespaces, and secure updates to namespaces require that the host submitting the update have a public or symmetric key that is known to the network and authorized for updates. In an unmanaged network, the publication and authorization of these keys is an unsolved problem.

Some managed networks get around this problem by having the DHCP server do DNS updates. However, this doesn’t really work, because DHCP doesn’t provide a mechanism for updating service discovery records: it only supports publishing A and AAAA records.

This partially solves the trust problem: DHCP can validate that a device is at least connected to a network link that is actually part of the managed network. This prevents an off-network attacker from registering a name, but provides no mechanism for actually validating the identity of the host registering the name. For example, it would be easy for an attacker on the network to steal a registered name.

Hybrid Multicast DNS [11] proposes a mechanism for extending multicast DNS beyond a single multicast domain. However, in order to use this as a solution, some shortcomings need to be considered.
Most obviously, it requires that every multicast domain have a separate name. This then requires that the homenet generate names for every multicast domain. These names would then be revealed to the end user. But since they would be generated automatically and arbitrarily, they would likely cause confusion rather than clarity, and in degenerate cases 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.

At present, the approach we intend to take with respect to disambiguation is that this will not be solved at a protocol level for devices that do not implement the registration protocol.

Authors’ Addresses

Ted Lemon
Nibbhaya Consulting
P.O. Box 958
Brattleboro, Vermont 05301
United States of America

Email: mellon@fugue.com

Daniel Migault
Ericsson
8400 boulevard Decarie
Montreal, QC H4P 2N2
Canada

Email: daniel.migault@ericsson.com

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

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

DNS-Based Service Discovery allows clients to discover available services using unicast DNS queries. In simple configurations these unicast DNS queries go directly to the appropriate authoritative server(s). In large networks that have complicated topology, or many client devices, or both, it can be advantageous to have an intermediary between the clients and authoritative servers. This intermediary, called a Discovery Broker, serves several purposes. A Discovery Broker can reduce load on both the servers and the clients, and gives the option of presenting clients with service discovery organized around logical, rather than physical, topology.
1. Introduction

DNS-Based Service Discovery (DNS-SD) [RFC6763] is a component of Zero
Configuration Networking [RFC6760] [ZC] [Roadmap].

DNS-SD operates on a single network link (broadcast domain) using
Multicast DNS [RFC6762]. DNS-SD can span multiple links using
unicast DNS.

In the DNS-SD specification [RFC6763] section 11, "Discovery of
Browsing and Registration Domains (Domain Enumeration)", describes
how client devices are automatically configured with the appropriate
unicast DNS domains in which to perform their service discovery
queries. When used in conjunction with a Discovery Proxy [DisProx]
this allows clients to discover services on remote links, even when
the devices providing those services support only the basic Multicast
DNS form of DNS-Based Service Discovery. A Discovery Broker is a
companion technology that operates in conjunction with existing
authoritative DNS servers (such as a Discovery Proxy [DisProx]) and
existing clients performing service discovery using unicast DNS
queries.
2. Problem Statement

The following description of how a Discovery Broker works is illustrated using the example of a long rectangular office building. The building is large enough to have hundreds or even thousands of employees working there, the network is large enough that it would be impractical to operate it as a single link (a single broadcast domain, with a single IPv4 subnet or IPv6 network prefix).

Suppose, for this example, that the network is divided into twelve separate links, connected by routers. Each link has its own IPv6 network prefix. The division of the network into twelve sections of roughly equal size is somewhat arbitrary, and does not necessarily follow any physical boundaries in the building that are readily apparent to its inhabitants. Two people in adjacent offices on the same corridor may have Ethernet ports connected to different links. Indeed, two devices in the same office, connected to the company network using secure Wi-Fi, may inadvertently associate with different access points, which happen to be connected to different wired links with different IPv6 network prefixes.

If this network were operated the way most networks have historically been operated, it would use only Multicast DNS Service Discovery, and adjacent devices that happen to connect to different underlying links would be unable to discover each other. And this would not be a rare occurrence. Since this example building contains eleven invisible boundaries between the twelve different links, anyone close to one of those invisible boundaries will have a population of nearby devices that are not discoverable on the network, because they’re on a different link. For example, a shared printer in a corridor outside one person’s office may not be discoverable by the person in the very next office.

One path to solving this problem is as follows:

1. Install a Discovery Proxy [DisProx] on each of the twelve links.
2. Create twelve named subdomains, such as, "services1.example.com", "services2.example.com", "services3.example.com", and so on.
3. Delegate each named subdomain to the corresponding Discovery Proxy on that link.
4. Create entries in the `ip6.arpa` reverse mapping zone directing clients on each link to perform service discovery queries in the appropriate named subdomains, as documented in section 11 of the DNS-SD specification [RFC6763].
In step 4 above, it might be tempting to add only a single record in each reverse mapping domain referencing the corresponding services subdomain. This would work, but it would only facilitate each client discovering the same services it could already discover using Multicast DNS [RFC6762]. In some cases even this is useful, such as when using Wi-Fi Access Points with multicast disabled for efficiency. In such cases this configuration would allow wireless clients to discover services on the wired network segment without having to use costly Wi-Fi multicast.

But for this example we want to achieve more than just equivalency with Multicast DNS.

In this example, each reverse mapping domain is populated with the name of its own services subdomain, plus its neighbors. The reverse mapping domain for the first link has two "lb._dns-sd._udp" PTR records, referencing "services1.example.com" and "services2.example.com". The second link references services1, services2, and services3. The third link references services2, services3, and services4. This continues along the building, until the last link, which references services11 and services12.

In this way a "sliding window" is created, where devices on each link are directed to look for services both on that link and on its two immediate neighbors. Depending on the physical and logical topologies of the building and its network, it may be appropriate to direct clients to query in more than three services subdomains. If the building were a ring instead of a linear rectangle, then the network topology would "wrap around", so that links 1 and 12 would be neighbors.

This solves the problem of being unable to discover a nearby device because it happens to be just the other side of one of the twelve arbitrary invisible network link boundaries.

For many cases this solution is adequate, but there is an issue to consider. In the example above, a client device on link 5 has TCP connections to three Discovery Proxies, on links 4, 5 and 6. In a more complex setup each client could have many more TCP connections to different Discovery Proxies.

Similarly, if there are a many clients, each Discovery Proxy could be required to handle thousands of simultaneous TCP connections from clients.

The solution to these two problems is the Discovery Broker.
3. Discovery Broker Operation

The Discovery Broker is an intermediary between the client devices and the Discovery Proxies. It is a kind of multiplexing crossbar switch. It shields the clients from having to connect to multiple Discovery Proxies, and it shields the Discovery Proxies from having to accept connections from thousands of clients.

Each client needs only a single TCP connection to a single Discovery Broker, rather than multiple TCP connections directly to multiple Discovery Proxies. This eases the load on client devices, which may be mobile and battery-powered.

Each Discovery Proxy needs to support connections to at most a small number of Discovery Brokers. The burden of supporting thousands of clients is taken by the Discovery Broker, which can be a powerful server in a data center. This eases the load on the Discovery Proxy, which may be implemented in a device with limited RAM and CPU resources, like a Wi-Fi access point or IP router.

Recall that a Discovery Proxy [DisProx] is a special kind of authoritative DNS server [RFC1034] [RFC1035]. Externally it behaves like a traditional authoritative DNS server, except that instead of getting its zone data from a manually-administered zone file, it learns its zone data dynamically as a result of performing Multicast DNS queries on its local link.

A Discovery Broker is a similar concept, except that it learns its zone data dynamically as a result of performing *unicast* DNS queries. For example, a Discovery Broker could be configured so that the answer for "<something>.discovery5.example.com" is obtained by performing corresponding unicast DNS queries:

```
<something>.services4.example.com
<something>.services5.example.com
<something>.services6.example.com
```

and then returning the union of the results as the answer. The rdata of the returned answers is not rewritten or modified in any way by the Discovery Broker.
4. Protocol Transparency

From the point of view of an authoritative DNS server such as a Discovery Proxy, the protocol a Discovery Broker uses to make requests of it is the exact same DNS protocol that any other client would use to make requests of it (which may be traditional one-shot DNS queries [RFC1034] [RFC1035] or long-lived DNS Push Notifications [Push]).

A Discovery Broker making requests is no different from any other client making requests. The fact that the Discovery Broker may be making a single request on behalf of thousands of clients making the same request, thereby shielding the Discovery Proxy from excessive traffic burden, is invisible to the Discovery Proxy.

This means that an authoritative DNS server such as a Discovery Proxy does not have to be aware that it is being queried by a Discovery Broker. In some scenarios a Discovery Proxy may be deployed with clients talking to it directly; in other scenarios the same Discovery Proxy product may be deployed with clients talking via a Discovery Broker. The Discovery Proxy simply answers queries as usual in both cases.

Similarly, from the point of view of a client, the protocol it uses to talk to a Discovery Broker is the exact same DNS protocol it uses to talk to a Discovery Proxy or any other authoritative DNS server.

This means that the client does not have to be aware that it is using a Discovery Broker. The client simply sends service discovery queries as usual, according to configuration it received from the network or otherwise, and receives answers as usual. A Discovery Broker may be employed to shield a Discovery Proxy from excessive traffic burden, but this is transparent to a client.

Another benefit for the client is that by having the Discovery Broker query multiple subdomains and aggregate the results, it saves the client from having to do multiple separate queries of its own.
5. Logical vs. Physical Topology

In the example so far, we have focussed on facilitating discovery of devices and services that are physically nearby.

Another application of the Discovery Broker is to facilitate discovery of devices and services according to other logical relationships.

For example, it might be considered desirable for the company’s two file servers to be discoverable company-wide, but for its many printers to only be discovered (by default) by devices on nearby network links.

As another example, company policy may block access to certain resources from Wi-Fi; in such cases it would make sense to implement consistent policies at the service discovery layer, to avoid the user frustration of services being discoverable on Wi-Fi that are not usable from Wi-Fi.

Such policies, and countless variations thereon, may be implemented in a Discovery Broker, limited only by the imagination of the vendor creating the Discovery Broker implementation.
6. Recursive Application

Due to the Protocol Transparency property described above, multiple Discovery Brokers may be "stacked" in whatever combinations are useful. A Discovery Broker makes queries in exactly the same way a client would, and a Discovery Broker accepts queries in exactly the same way a Discovery Proxy (or other authoritative DNS server) would. This means that a Discovery Broker talking to another Discovery Broker is no different from client-to-broker or broker-to-proxy communication, or indeed, direct client-to-proxy communication. The arrows in the chart below are all instances of the same communication protocol.

\[
\begin{array}{c}
\text{client} \rightarrow \text{proxy} \\
\text{client} \rightarrow \text{broker} \rightarrow \text{proxy} \\
\text{client} \rightarrow \text{broker} \rightarrow \text{broker} \rightarrow \text{proxy}
\end{array}
\]

This makes it possible to combine Discovery Brokers with different functionality. A Discovery Broker performing physical aggregation could be used in conjunction with a Discovery Broker performing policy-based filtering, as illustrated below:

\[
\begin{array}{c|c|c|c}
\text{Ethernet} & \rightarrow & \text{Aggregating} & \rightarrow \\
\text{Client} & & \text{Broker} & \text{Proxy} \\
\hline
\text{Wi-Fi} & \rightarrow & \text{Filtering} & \rightarrow \\
\text{Client} & & \text{Broker} & \text{Proxy} \\
\end{array}
\]
7. Security Considerations

Discovery (or non-discovery) of services is not a substitute for suitable access control. Servers listening on open ports are generally discoverable via a brute-force port scan anyway; DNS-Based Service Discovery makes access to these services easier for legitimate users.

8. Informative References


Authors’ Addresses

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

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

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
Abstract

This document extends the specification of the Discovery Proxy for Multicast DNS-Based Service Discovery. It describes a lightweight relay mechanism, a Discovery Relay, which allows Discovery Proxies to provide service on multicast links to which they are not directly attached.

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 https://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 19, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of
1. Introduction

The Discovery Proxy for Multicast DNS-Based Service Discovery
[I-D.ietf-dnssd-hybrid] is a mechanism for discovering services on a
subnetted network through the use of Discovery Proxies, which issue
Multicast DNS (mDNS) requests [RFC6762] on various multicast links in
the network on behalf of a remote host performing DNS-Based Service
Discovery [RFC6763].

In the original Discovery Proxy specification, it is imagined that
for every multicast link on which services will be discovered, a host
will be present running a full Discovery Proxy. This document
introduces a lightweight Discovery Relay which can be used to provide
discovery services on a multicast link without requiring a full
Discovery Proxy on every multicast link.

The Discovery Relay operates by listening for TCP connections from
Discovery Proxies. When a Discovery Proxy connects, the connection
is authenticated and secured using TLS. The Discovery Proxy can then
specify one or more multicast links from which it wishes to receive
mDNS traffic. The Discovery Proxy can also send messages to be
transmitted on its behalf on one or more of those multicast links.
DNS Stateful Operations (DSO) [I-D.ietf-dnsop-session-signal] is used
as a framework for conveying interface and IP header information
associated with each message.

The Discovery Relay functions essentially as a set of one or more
remote virtual interfaces for the Discovery Proxy, one on each
multicast link to which the Discovery Relay is connected. In a
complex network, it is possible that more than one Discovery Relay
will be connected to the same multicast link; in this case, the
Discovery Proxy ideally should only be using one such Relay Proxy per
multicast link, since using more than one will generate duplicate
traffic.

How such duplication is detected and avoided is out of scope for this
document; in principle it could be detected using HNCP [RFC7788] or
configured using some sort of orchestration software in conjunction
with NETCONF [RFC6241] or CPE WAN Management Protocol [TR-069].

Since the primary purpose of a Discovery Relay is providing remote
virtual interface functionality to Discovery Proxies, this document
is written with that usage in mind, and this document talks about
Discovery Relays receiving requests from Discovery Proxies. However,
in principle, a Discovery Relay could be used by any properly
authorized client, so it should be understood that in this document
the term, "Discovery Proxy," potentially means, "any properly
authorized client."
2. Terminology

The following definitions may be of use:

mDNS Agent  A host which sends and/or responds to mDNS queries.

Discovery Proxy  A network service which receives well-formed questions using the DNS protocol, performs multicast DNS queries to answer those questions, and responds with those answers using the DNS protocol.

Discovery Relay  A network service which relays received mDNS messages to a Discovery Proxy, and can transmit mDNS messages on behalf of that Discovery Proxy.

multicast link  A maximal set of network connection points, such that any host connected to any connection point in the set may send a packet with a link-local multicast destination address (specifically the mDNS link-local multicast destination address [RFC6762]) that will be received by all hosts connected to all other connection points in the set. Note that it is becoming increasingly common for a multicast link to be smaller than its corresponding unicast link. For example it is becoming common to have multiple Wi-Fi Access Points on a shared Ethernet backbone, where the multiple Wi-Fi Access Points and their shared Ethernet backbone form a single unicast link (a single IPv4 subnet, or single IPv6 prefix) but not a single multicast link. Unicast packets between two hosts on that IPv4 subnet or IPv6 prefix are correctly delivered, but multicast packets are not forwarded between the various Wi-Fi Access Points. Given the slowness of Wi-Fi multicast, the decision to not forward multicast packets between Wi-Fi Access Points is reasonable, and that further supports the need for technologies like Discovery Proxy and Discovery Relay to facilitate discovery on these networks.

whitelist  A list of one or more IP addresses from which a Discovery Relay may accept connections.

silently discard  When a message that is not supported or not permitted is received, and the required response to that message is to "silently discard" it, that means that no response is sent by the service that is discarding the message to the service that sent it. The service receiving the message may log the event, and may also count such events: "silently" does not preclude such behavior.
3. Protocol Overview

This document describes a way for Discovery Proxies to communicate with mDNS agents on remote multicast links to which they are not directly connected, using a Discovery Relay. As such, there are two parts to the protocol: connections between Discovery Proxies and Discovery Relays, and communications between Discovery Relays and mDNS agents.

3.1. Connections between Proxies and Relays (overview)

Discovery Relays listen for incoming connection requests. Connections between Discovery Proxies and Discovery Relays are established by Discovery Proxies. Connections are authenticated and encrypted using TLS, with both client and server certificates. Connections are long-lived: a Discovery Proxy is expected to send many queries over a single connection, and Discovery Relays will forward all mDNS traffic from subscribed interfaces over the connection.

The stream encapsulated in TLS will carry DNS frames as in the DNS TCP protocol [RFC1035] Section 4.2.2. However, all messages will be DSO messages [I-D.ietf-dnsop-session-signal]. There will be three types of such messages between Discovery Proxy and Discovery Relay:

- Control messages from Proxy to Relay
- mDNS messages from Proxy to Relay
- mDNS messages from Relay to Proxy

Subscribe messages from the Discovery Proxy to the Discovery Relay indicate to the Discovery Relay that mDNS messages from one or more specified multicast links are to be relayed to the Discovery Proxy.

mDNS messages from a Discovery Proxy to a Discovery Relay cause the Discovery Relay to transmit the mDNS message on one or more multicast links to which the Discovery Relay host is directly attached.

mDNS messages from a Discovery Relay to a Discovery Proxy are sent whenever an mDNS message is received on a multicast link to which the Discovery Relay has subscribed.

During periods with no traffic flowing, Discovery Proxies are responsible for generating any necessary keepalive traffic, as stated in the DSO specification [I-D.ietf-dnsop-session-signal].
3.2. mDNS Messages On Multicast Links

Discovery Relays listen for mDNS traffic on all configured multicast links that have at least one active subscription from a Discovery Proxy. When an mDNS message is received on a multicast link, it is forwarded on every open Discovery Proxy connection that is subscribed to mDNS traffic on that multicast link. In the event of congestion, where a particular Discovery Proxy connection has no buffer space for an mDNS message that would otherwise be forwarded to it, the mDNS message is not forwarded to it. Normal mDNS retry behavior is used to recover from this sort of packet loss. Discovery Relays are not expected to buffer more than a few mDNS packets. Excess mDNS packets are silently discarded. In reality this is expected to be a nonissue. Particularly on networks like Wi-Fi, multicast packets are transmitted at rates ten or even a hundred times slower than unicast packets. This means that even at peak multicast packets rates, it is likely that a unicast TCP connection will be able to carry those packets with ease.

Discovery Proxies send mDNS messages they wish to have sent on their behalf on remote multicast link(s) on which the Discovery Proxy has an active subscription. A Discovery Relay will not transmit mDNS packets on any multicast link on which the remote Discovery Proxy does not have an active subscription, since it makes no sense for a Discovery Proxy to ask to have a query sent on its behalf if it’s not able to receive the responses to that query.

4. Connections between Proxies and Relays (details)

When a Discovery Relay starts, it opens a passive TCP listener to receive incoming connection requests from Discovery Proxies. This listener may be bound to one or more source IP addresses, or to the wildcard address, depending on the implementation. When a connection is received, the relay must first validate that it is a connection to an IP address to which connections are allowed. For example, it may be that only connections to ULAs are allowed, or to the IP addresses configured on certain interfaces. If the listener is bound to a specific IP address, this check is unnecessary.

If the relay is using an IP address whitelist, the next step is for the relay to verify that that the source IP address of the connection is on its whitelist. If the connection is not permitted either because of the source address or the destination address, the Discovery Relay responds to the TLS Client Hello message from the Discovery Proxy with a TLS user_canceled alert ([I-D.ietf-tls-tls13] Section 6.1).
Otherwise, the Discovery Relay will attempt to complete a TLS handshake with the Discovery Proxy. Discovery Proxies are required to send the post_handshake_auth extension ([I-D.ietf-tls-tls13] Section 4.2.5). If a Discovery Relay receives a ClientHello message with no post_handshake_auth extension, the Discovery Relay rejects the connection with a certificate_required alert ([I-D.ietf-tls-tls13] Section 6.2).

Once the TLS handshake is complete, the Discovery Relay MUST request post-handshake authentication as described in ([I-D.ietf-tls-tls13] Section 4.6.2). If the Discovery Proxy refuses to send a certificate, or the key presented does not match the key associated with the IP address from which the connection originated, or the CertificateVerify does not validate, the connection is dropped with the TLS access_denied alert ([I-D.ietf-tls-tls13] Section 6.2).

Once the connection is established and authenticated, it is treated as a DNS TCP connection [RFC1035].

Aliveness of connections between Discovery Proxies and Relays is maintained as described in Section 4 of [I-D.ietf-dnsop-session-signal]. Discovery Proxies must also honor the ‘Retry Delay’ TLV (section 5 of [I-D.ietf-dnsop-session-signal]) if sent by the Discovery Relay.

Discovery Proxies may establish more than one connection to a specific Discovery Relay. This would happen in the case that a TCP connection stalls, and the Discovery Proxy is able to reconnect before the previous connection has timed out. It could also happen as a result of a server restart. It is not likely that two active connections from the same Discovery Proxy would be present at the same time, but it must be possible for additional connections to be established. The Discovery Relay may drop the old connection when the new one has been fully established, including a successful TLS handshake. What it means for two connections to be from the same Discovery Proxy is that the connections both have source addresses that belong to the same Discovery Proxy, and that they were authenticated using the same client certificate.

5. Traffic from Relays to Proxies

The mere act of connecting to a Discovery Relay does not result in any mDNS traffic being forwarded. In order to request that mDNS traffic from a particular multicast link be forwarded on a particular connection, the Discovery Proxy must send one or more DSO messages, each containing a single mDNS Link Request TLV (Section 8.1) indicating the multicast link from which traffic is requested.
When such a message is received, the Discovery Relay validates that the specified multicast link is available for forwarding, and that forwarding is enabled for that multicast link. For each such message the Discovery Relay validates the multicast link specified and includes, in a single response, RCODE 0 if the multicast link specified is valid, or RCODE 3 (NXDOMAIN / Name Error -- Named entity does not exist) otherwise. For each valid multicast link, it begins forwarding all mDNS traffic from that link to the Discovery Proxy. Delivery is not guaranteed: if there is no buffer space, packets will be dropped. It is expected that regular mDNS retry processing will take care of retransmission of lost packets. The amount of buffer space is implementation dependent, but generally should not be more than the bandwidth delay product of the TCP connection [RFC1323].

The Discovery Relay should use the TCP_NOTSENT_LOWAT mechanism [NOTSENT][PRIO] or equivalent, to avoid building up a backlog of data in excess of the amount necessary to have in flight to fill the bandwidth delay product of the TCP connection.

mDNS messages from Relays to Proxies are framed within DSO messages. Each DSO message can contain multiple TLVs, but only a single mDNS message is conveyed per DSO message. Each forwarded mDNS message is contained in an mDNS Message TLV (Section 8.4). The layer two source address of the message, if known, MAY be encoded in a Layer Two Source TLV (Section 8.5). The source IP address and port of the message MUST be encoded in an IP Source TLV (Section 8.6). The multicast link on which the message was received MUST be encoded in a Link Identifier TLV (Section 8.3). The Discovery Proxy MUST silently ignore unrecognized TLVs in mDNS messages, and MUST NOT discard mDNS messages that include unrecognized TLVs.

A Discovery Proxy may discontinue listening for mDNS messages on a particular multicast link by sending a DSO message containing an mDNS Link Discontinue TLV (Section 8.2). Subsequent messages from that link that had previously been queued may arrive after listening has been discontinued. The Discovery Proxy should silently discard such messages. The Discovery Relay MUST discontinue generating such messages as soon as the request is received. The Discovery Relay does not respond to this message other than to discontinue forwarding mDNS messages from the specified links.

6. Traffic from Proxies to Relays

Like mDNS traffic from relays, each mDNS message sent by a Discovery Proxy to a Discovery Relay is encapsulated in an mDNS Message TLV (Section 8.4) within a DSO message. Each message MUST contain one or more Link Identifier TLVs (Section 8.3). The Discovery Relay will transmit the message to the mDNS port and multicast address on each link specified in the message using the specified IP address family.
7. Discovery Proxy Behavior

Discovery Proxies treat multicast links for which Discovery Relay service is being used as if they were virtual interfaces; in other words, a Discovery Proxy serving multiple multicast links using multiple Discovery Relays behaves the same as a Discovery Proxy serving multiple multicast links using multiple physical network interfaces. In this section we refer to multicast links served directly by the Discovery Proxy as locally-connected links, and multicast links served through the Discovery Relay as relay-connected links.

What this means is that when a Discovery Proxy receives a DNSSD query from a client via unicast, it will generate mDNS query messages on the relevant multicast link(s) for which it is acting as a proxy. For locally-connected link(s), those query messages will be sent directly. For relay-connected link(s), the query messages will be sent through the Discovery Relay that is being used to serve that multicast link.

Responses from devices on locally-connected links are processed normally. Responses from devices on relay-connected links are received by the Discovery Relay, encapsulated, and forwarded to the Discovery Proxy; the Discovery Proxy then processes these messages using the link-identifying information included in the encapsulation.

Discovery Proxies do not generally respond to mDNS queries on relay-connected links. The one exception is responding to the Domain Enumeration queries used to bootstrap unicast service discovery ("lb._dns-sd._udp.local", etc.) [RFC6763]. Apart from these Domain Enumeration queries, if any other mDNS query is received from a Discovery Relay, the Discovery Proxy silently discards it.

In principle it could be the case that some device is capable of performing service discovery using Multicast DNS, but not using traditional unicast DNS. Responding to mDNS queries received from the Discovery Relay could address this use case. However, continued reliance on multicast is counter to the goals of the current work in service discovery, and to benefit from wide-area service discovery such client devices should be updated to support service discovery using unicast queries.
8. DSO TLVs

This document defines a modest number of new DSO TLVs.

8.1. mDNS Link Request

The mDNS Link Request TLV conveys a link identifier from which a Discovery Proxy is requesting that a Discovery Relay forward mDNS traffic. The link identifier comes from the provisioning configuration (see Section 9). The DSO-TYPE for this TLV is TBD-R. DSO-LENGTH is always 5. DSO-DATA is the 8-bit address family followed by the 32-bit link identifier, in network byte order, as described in Section 9. An address family value of 1 indicates IPv4 and 2 indicates IPv6, as recorded in the IANA Registry of Address Family Numbers [AdFam].

The mDNS Link Request TLV can only be used as a primary TLV, and requires an acknowledgement.

At most one mDNS Link Request TLV may appear in a DSO message. To request multiple link subscriptions, multiple separate DSO messages are sent, each containing a single mDNS Link Request TLV.

8.2. mDNS Link Discontinue

The mDNS Link Discontinue TLV is used by Discovery Proxies to unsubscribe to mDNS messages on the specified multicast link. DSO-TYPE is TBD-D. DSO-LENGTH is always 5. DSO-DATA is the 8-bit address family followed by the 32-bit link identifier, in network byte order, as described in Section 9.

The mDNS Link Discontinue TLV can only be used as a primary TLV, and is not acknowledged.

At most one mDNS Link Discontinue TLV may appear in a DSO message. To unsubscribe from multiple links, multiple separate DSO messages are sent, each containing a single mDNS Link Discontinue TLV.

8.3. Link Identifier

This option is used both in DSO messages from Discovery Relays to Discovery Proxies that contain received mDNS messages, and from Discovery Proxies to Discovery Relays that contain mDNS messages to be transmitted on the multicast link. In the former case, it indicates the multicast link on which the message was received; in the latter case, it indicates the multicast link on which the message should be transmitted. DSO-TYPE is TBD-L. DSO-LENGTH is always 5.
8.4. mDNS Message

The mDNS Message TLV is used to encapsulate an mDNS message that is being forwarded from a multicast link to a Discovery Proxy, or is being sent from a Discovery Proxy for transmission on a multicast link. Only the application layer payload of the mDNS message is carried in the DSO mDNS Message TLV, i.e., just the DNS message itself, beginning with the DNS Message ID, not the IP or UDP headers. The DSO-TYPE for this TLV is TBD-M. DSO-LENGTH is the length of the encapsulated mDNS message. DSO-DATA is the content of the encapsulated mDNS message.

The mDNS Message TLV can only be used as a primary TLV, and is not acknowledged.

8.5. Layer Two Source Address

The Layer Two Source Address TLV is used to report the link-layer address from which an mDNS message was received. This TLV is optionally present in DSO messages from Discovery Relays to Discovery Proxies that contain mDNS messages when the source link-layer address is known. The DSO-TYPE is TBD-2. DSO-LENGTH is variable, depending on the length of link-layer addresses on the link from which the message was received. DSO-DATA is the link-layer address as it was received on the link.

The Layer Two Source Address TLV can only be used as an additional TLV.

8.6. IP Source

The IP Source TLV is used to report the IP source address and port from which an mDNS message was received. This TLV is present in DSO messages from Discovery Relays to Discovery Proxies that contain mDNS messages. DSO-TYPE is TBD-A. DSO-LENGTH is either 6, for an IPv4 address, or 18, for an IPv6 address. DSO-DATA is the source port, followed by the IP Address, in network byte order.

The IP Source TLV can only be used as an additional TLV.
9. Provisioning

In order for a Discovery Proxy to use Discovery Relays, it must be configured with sufficient information to identify multicast links on which service discovery is to be supported and connect to discovery relays supporting those multicast links, if it is not running on a host that is directly connected to those multicast links.

A Discovery Relay must be configured both with a set of multicast links to which the host on which it is running is connected, on which mDNS relay service is to be provided, and also with a list of one or more Discovery Proxies authorized to use it.

On a network supporting DNS Service Discovery using Discovery Relays, more than one different Discovery Relay implementation is likely be present. While it may be that only a single Discovery Proxy is present, that implementation will need to be able to be configured to interoperate with all of the Discovery Relays that are present. Consequently, it is necessary that a standard set of configuration parameters be defined for both Discovery Proxies and Discovery Relays.

DNS Service Discovery generally operates within a constrained set of links, not across the entire internet. This section assumes that what will be configured will be a limited set of links operated by a single entity or small set of cooperating entities, among which services present on each link should be available to users on that link and every other link. This could be, for example, a home network, a small office network, or even a network covering an entire building or small set of buildings. The set of Discovery Proxies and Discovery Relays within such a network will be referred to in this section as a ‘Discovery Domain’.

Depending on the context, several different candidates for configuration of Discovery Proxies and Discovery relays may be applicable. The simplest such mechanism is a manual configuration file, but regardless of provisioning mechanism, certain configuration information needs to be communicated to the devices, as outlined below.

9.1. Provisioned Objects

Three types of objects must be described in order for Discovery Proxies and Discovery Relays to be provisioned: Discovery Proxies, Multicast Links, and Discovery Relays. "Human-readable" below means actual words or proper names that will make sense to an untrained human being. "Machine-readable" means a name that will be used by machines to identify the entity to which the name refers. Each
entity must have a machine-readable name and may have a human-readable name. No two entities can have the same human-readable name. Similarly, no two entities can have the same machine-readable name.

9.1.1. Multicast Link

The description of a multicast link consists of:

- **link-identifier**: A 32-bit identifier that uniquely identifies that link within the Discovery Domain. Each link MUST have exactly one such identifier. Link Identifiers do not have any special semantics, and are not intended to be human-readable.

- **ldh-name**: A fully-qualified domain name for the multicast link that is used to form an LDH domain name as described in section 5.3 of the Discovery Proxy specification [I-D.ietf-dnssd-hybrid]. This name is used to identify the link during provisioning, and must be present.

- **hr-name**: A human-readable user-friendly fully-qualified domain name for the multicast link. This name MUST be unique within the Discovery Domain. Each multicast link MUST have exactly one such name. The hr-name MAY be the same as the ldh-name. (The hr-name is allowed to contain spaces, punctuation and rich text, but it is not required to do so.)

The ldh-name and hr-name can be used to form the LDH and human-readable domain names as described in [I-D.ietf-dnssd-hybrid], section 5.3.

Note that the ldh-name and hr-name can be used in two different ways.

On a small home network with little or no human administrative configuration, link names may be directly visible to the user. For example, a search in ‘home.arpa’ on a small home network may discover services on both ethernet.home.arpa and wi-fi.home.arpa. In the case of a home user who has one Ethernet-connected printer and one Wi-Fi-connected printer, discovering that they have one printer on ethernet.home.arpa and another on wi-fi.home.arpa is understandable and meaningful.

On a large corporate network with hundreds of Wi-Fi Access Points, the individual link names of the hundreds of multicast links are less likely to be useful to end users. In these cases, Discovery Broker functionality [I-D.sctl-discovery-broker] is used to translate the many link names to something more meaningful to users. For example, in a building with 50 Wi-Fi Access Points, each with their own link
names, services on all the different physical links may be presented
to the user as appearing in ‘headquarters.example.com’. In this
case, the individual link names can be thought of similar to MAC
addresses or IPv6 addresses. They are used internally by the
software as unique identifiers, but generally are not exposed to end
users.

9.1.2. Discovery Proxy

The description of a Discovery Proxy consists of:

- **name**  a machine-readable name used to reference this Discovery Proxy
  in provisioning.

- **hr-name**  an optional human-readable name which can appear in
  provisioning, monitoring and debugging systems. Must be unique
  within a Discovery Domain.

- **public-key**  a public key that identifies the Discovery Proxy. This
  key can be shared across services on the Discovery Proxy Host.
  The public key is used both to uniquely identify the Discovery
  Proxy and to authenticate connections from it.

- **private-key**  the private key corresponding to the public key.

- **source-ip-addresses**  a list of IP addresses that may be used by the
  Discovery Proxy when connecting to Discovery Relays. These
  addresses should be addresses that are configured on the Discovery
  Proxy Host. They should not be temporary addresses. All such
  addresses must be reachable within the Discovery Domain.

- **public-ip-addresses**  a list of IP addresses that may be used to
  submit DNS queries to the Discovery Proxy. This is not used for
  interoperation with Discovery Relays, but is mentioned here for
  completeness; this list of addresses may differ from the ‘source-
ip-addresses’ list. If any of these addresses are reachable from
  outside of the Discovery Domain, services in that domain will be
  discoverable outside of the domain.

- **multicast links**  a list of multicast links on which this Discovery
  Proxy is expected to provide service

The private key should never be distributed to other hosts; all of
the other information describing a Discovery Proxy can be safely
shared with Discovery Relays.
9.1.3. Discovery Relay

The description of a Discovery Relay consists of:

name  a required machine-readable identifier used to reference the relay

hr-name  an optional human-readable name which can appear in provisioning, monitoring and debugging systems. Must be unique within a Discovery Domain.

public-key  a public key that identifies the Discovery Relay. This key can be shared across services on the Discovery Relay Host. Indeed, if a Discovery Proxy and Discovery Relay are running on the same host, the same key may be used for both. The public key uniquely identifies the Discovery Relay and is used by the Discovery Proxy to verify that it is talking to the intended Discovery Relay after a TLS connection has been established.

private-key  the private key corresponding to the public key.

connect-tuples  a list of IP address/port tuples that may be used to connect to the Discovery Relay. The relay may be configured to listen on all addresses on a single port, but this is not required, so the port as well as the address must be specified.

multicast links  a list of multicast links to which this relay is physically connected.

The private key should never be distributed to other hosts; all of the other information describing a Discovery Relay can be safely shared with Discovery Proxies.

9.2. Configuration Files

For this discussion, we assume the simplest possible means of configuring Discovery Proxies and Discovery Relays: the configuration file. Any environment where changes will happen on a regular basis will either require some automatic means of generating these configuration files as the network topology changes, or will need to use a more automatic method for configuration, such as HNCP [RFC7788].

There are many different ways to organize configuration files. This discussion assumes that multicast links, relays and proxies will be specified as objects, as described above, perhaps in a master file, and then the specific configuration of each proxy or relay will reference the set of objects in the master file, referencing objects...
by name. This approach is not required, but is simply shown as an example. In addition, the private keys for each proxy or relay must appear only in that proxy or relay’s configuration file.

The master file contains a list of Discovery Relays, Discovery Proxies and Multicast Links. Each object has a name and all the other data associated with it. We do not formally specify the format of the file, but it might look something like this:

```plaintext
Relay upstairs
  public-key xxx
  connect-tuple 192.0.2.1 1917
  connect-tuple fd00::1 1917
  link upstairs-wifi
  link upstairs-wired
Relay downstairs
  public-key yyy
  connect-tuple 192.51.100.1 2088
  connect-tuple fd00::2 2088
  link downstairs-wifi
  link downstairs-wired
Proxy main
  public-key zzz
  address 203.1.113.1
Link upstairs-wifi
  id 1
    name Upstairs Wifi
Link upstairs-wired
  id 2
    hr-name Upstairs Wired
Link downstairs-wifi
  id 3
    name Downstairs Wifi
Link downstairs-wired
  id 4
    hr-name Downstairs Wired
```
9.3. Discovery Proxy Configuration

The Discovery Proxy configuration contains enough information to identify which Discovery Proxy is being configured, enumerate the list of multicast links it is intended to serve, and provide keying information it can use to authenticate to Discovery Relays. It may also contain custom information about the port and/or IP address(es) on which it will respond to DNS queries.

An example configuration, following the convention used in this section, might look something like this:

```
Proxy main
  private-key zzz
  subscribe upstairs-wifi
  subscribe downstairs-wifi
  subscribe upstairs-wired
  subscribe downstairs-wired
```

When combined with the master file, this configuration is sufficient for the Discovery Proxy to identify and connect to the relay proxies that serve the links it is configured to support.

9.4. Discovery Relay Configuration

The discovery relay configuration just needs to tell the discovery relay what name to use to find its configuration in the master file, and what the private key is corresponding to its public key in the master file. For example:

```
Relay Downstairs
  private-key yyy
```
10. Security Considerations

11. IANA Considerations

The IANA is kindly requested to update the DSO Type Codes Registry [I-D.ietf-dnsop-session-signal] by allocating codes for each of the TBD type codes listed in the following table, and by updating this document, here and in Section 8. Each type code should list this document as its reference document.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Status</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD-R</td>
<td>Standard</td>
<td>mDNS Link Request</td>
</tr>
<tr>
<td>TBD-D</td>
<td>Standard</td>
<td>mDNS Discontinue</td>
</tr>
<tr>
<td>TBD-L</td>
<td>Standard</td>
<td>Link Identifier</td>
</tr>
<tr>
<td>TBD-M</td>
<td>Standard</td>
<td>mDNS Message</td>
</tr>
<tr>
<td>TBD-2</td>
<td>Standard</td>
<td>Layer Two Source Address</td>
</tr>
<tr>
<td>TBD-A</td>
<td>Standard</td>
<td>IP Source</td>
</tr>
</tbody>
</table>

DSO Type Codes to be allocated

12. Acknowledgments
13. References

13.1. Normative References

[I-D.ietf-dnsop-session-signal]

[I-D.ietf-dnssd-hybrid]
Cheshire, S., "Discovery Proxy for Multicast DNS-Based Service Discovery", draft-ietf-dnssd-hybrid-07 (work in progress), September 2017.

[I-D.ietf-tls-tls13]

[I-D.sctl-discovery-broker]


13.2. Informative References


Authors' Addresses

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
USA
Phone: +1 408 974 3207
Email: cheshire@apple.com

Ted Lemon
Nibbhaya Consulting
P.O. Box 958
Brattleboro, Vermont 05301
United States of America
Email: mellon@fugue.com
Abstract

The DNS-SD Service Registration Protocol uses the standard DNS Update mechanism to enable DNS-Based Service Discovery using only unicast packets. This eliminates the dependency on Multicast DNS as the foundation layer, which greatly improves scalability and improves performance on networks where multicast service is not an optimal choice, particularly 802.11 (Wi-Fi) and 802.15.4 (IoT) networks. DNS-SD Service registration uses public keys and SIG(0) to allow services to defend their registrations against attack.
1. Introduction

DNS-Based Service Discovery [RFC6763] is a component of Zero Configuration Networking [RFC6760] [ZC] [I-D.cheshire-dnssd-roadmap].

This document describes an enhancement to DNS-Based Service Discovery [RFC6763] that allows services to automatically register their services using the DNS protocol rather than using mDNS. There is already a large installed base of DNS-SD clients that can do service discovery using the DNS protocol. This extension makes it much easier to take advantage of this existing functionality.

This document is intended for three audiences: implementors of software that provides services that should be advertised using DNS-SD, implementors of DNS servers that will be used in contexts where DNS-SD registration is needed, and administrators of networks where DNS-SD service is required. The document is intended to provide sufficient information to allow interoperable implementation of the registration protocol.

DNS-Based Service Discovery (DNS-SD) allows services to advertise the fact that they provide service, and to provide the information required to access that service. Clients can then discover the set of services of a particular type that are available. They can then select a service from among those that are available and obtain the information required to use it.

The DNS-SD Service Registration protocol, described in this document, provides a reasonably secure mechanism for publishing this information. Once published, these services can be readily discovered by clients using standard DNS lookups.

In the DNS-Based Service Discovery specification [RFC6763] Section 10 "Populating the DNS with Information" briefly discusses ways that services can publish their information in the DNS namespace. In the case of Multicast DNS [RFC6762], it allows services to publish their information on the local link, using names in the ".local" namespace, which makes their services directly discoverable by peers attached to that same local link.

RFC6763 also allows clients to discover services using the DNS protocol [RFC1035]. This can be done by having a system administrator manually configure service information in the DNS, but
manually populating DNS authoritative server databases is costly and potentially error-prone, and requires a knowledgable network administrator. Consequently, although all DNS-SD client implementations of which we are aware support DNS-SD using DNS queries, in practice it is used much less frequently than mDNS. The Discovery Proxy [I-D.ietf-dnssd-hybrid] provides one way to automatically populate the DNS namespace, but is only appropriate on networks where services are already advertised using mDNS. This document describes a solution more suitable for networks where multicast is inefficient, or undesirable for other reasons, by supporting both offering of services, and discovery of services, using unicast.

2. Service Registration Protocol

Services that implement the DNS-SD Service Registration Protocol use DNS Update [RFC2136] [RFC3007] to publish service information in the DNS. Two variants exist, one for full-featured devices, and one for devices designed for "Constrained-Node Networks" [RFC7228].

Full-featured devices are either configured manually, or use the "dr._dns-sd._udp" query [RFC6763] to learn the default registration domain from the network. Using the chosen service registration domain, full-featured devices construct the names of the SRV, TXT, and PTR records describing their service(s). For these names they then discover the zone apex of the closest enclosing DNS zone using SOA queries [I-D.ietf-dnssd-push]. Having discovered the enclosing DNS zone, they query for the "_dns-update._udp<zone>" SRV record to discover the server to which they should send DNS updates.

For devices designed for "Constrained-Node Networks" [RFC7228] some simplifications are used. Instead of being configured with (or discovering) the service registration domain, the (proposed) special use domain name [RFC6761] "services.arpa" is used. Instead of learning the server to which they should send DNS updates, a fixed IPv6 anycast address is used (value TBD). It is the responsibility of a "Constrained-Node Network" supporting DNS-SD Service Registration Protocol to provide appropriate anycast routing to deliver the DNS updates to the appropriate server. It is the responsibility of the DNS-SD Service Registration server on a "Constrained-Node Network" to handle the updates appropriately. In some network environments, updates may be accepted directly into a local "services.arpa" zone, which has only local visibility. In other network environments, updates for names ending in "services.arpa" may be rewritten internally to names with broader visibility.
The reason for these different assumptions is that "Constrained-Node Networks" generally require special egress support, and Anycast packets captured at the "Constrained-Node Network" egress can be assumed to have originated locally. Low-power devices that typically use "Constrained-Node Networks" may have very limited battery power. The additional DNS lookups required to discover a registration server and then communicate with it will increase the power required to advertise a service; for low-power devices, the additional flexibility this provides does not justify the additional use of power.

General networks have the potential to have more complicated topologies at the Internet layer, which makes anycast routing more difficult. Such networks may or may not have the infrastructure required to route anycast to a server that can process it. However, they can be assumed to be able to provide registration domain discovery and routing. By requiring the use of TCP, the possibility of off-network spoofing is eliminated.

We will discuss several parts to this process: how to know what to publish, how to know where to publish it (under what name), how to publish it, how to secure its publication, and how to maintain the information once published.

2.1. What to publish

We refer to the message that services using the DNSSD Registration Protocol send as a Registration. Three types of updates appear in a Registration: Service Discovery records, Service Description records, and Host Description records.

- Service Discovery records are one or more PTR RRs, mapping from the generic service type (or subtype) to the specific Service Instance Name.

- Service Description records are exactly one SRV RR, and one or more TXT RRs, both with the same name, the Service Instance Name ([RFC6763] section 4.1). In principle Service Description records can include other record types, with the same Service Instance Name, though in practice they rarely do. The Service Instance Name MUST be referenced by one or more Service Discovery PTR records, unless it is a placeholder service registration for an intentionally non-discoverable service name.

- The Host Description records for a service are a KEY RR, used to claim exclusive ownership of the service registration, and one or more RRs of type A or AAAA, giving the IPv4 or IPv6 address(es) of the host where the service resides.
RFC 6763 describes the details of what each of these types of updates contains and is the definitive source for information about what to publish; the reason for mentioning it here is to provide the reader with enough information about what will be published that the service registration process can be understood at a high level without first learning the full details of DNS-SD. Also, the "Service Instance Name" is an important aspect of first-come, first-serve naming, which we describe later on in this document.

2.2. Where to publish it

Multicast DNS uses a single namespace, ".local", which is valid on the local link. This convenience is not available for DNS-SD using the DNS protocol: services must exist in some specific unicast namespace.

As described above, full-featured devices are responsible for knowing in what domain they should register their services. Devices made for "Constrained-Node Networks" register in the (proposed) special use domain name [RFC6761] "services.arpa", and let the DNS-SD Service Registration server handle rewriting that to a different domain if necessary.

2.3. How to publish it

It is possible to issue a DNS Update that does several things at once; this means that it’s possible to do all the work of adding a PTR resource record to the PTR RRset on the Service Name if it already exists, or creating one if it doesn’t, and creating or updating the Service Instance Name and Host Description in a single transaction.

A Registration is therefore implemented as a single DNS Update message that contains a service’s Service Discovery records, Service Description records, and Host Description records.

Updates done according to this specification are somewhat different than regular DNS Updates as defined in RFC2136. RFC2136 assumes that updating is a fairly heavyweight process, so you might first attempt to add a name if it doesn’t exist, and then in a second message update the name if it does exist but matches certain preconditions. Because the registration protocol uses a single transaction, some of this adaptability is lost.

In order to allow updates to happen in a single transaction, Registrations do not include update constraints. The constraints specified in Section 2.4.2 are implicit in the processing of
Registrations, and so there is no need for the service sending the Registration to put in any explicit constraints.

2.3.1. How DNS-SD Service Registration differs from standard RFC2136 DNS Update

DNS-SD Service Registration is based on standard RFC2136 DNS Update, with some differences:

- It implements first-come first-served name allocation, protected using SIG(0).
- It enforces policy about what updates are allowed.
- It optionally performs rewriting of "services.arpa" to some other domain.
- It optionally performs automatic population of the address-to-name reverse mapping domains.
- A DNS-SD Service Registration server is not required to implement general DNS Update prerequisite processing.
- Simplified clients are allowed to send updates to an anycast address, for names ending in "services.arpa"

2.3.2. Testing using standard RFC2136-compliant servers

It may be useful to set up a DNS server for testing that does not implement the Registration protocol. This can be done by configuring the server to listen on the anycast address, or advertising it in the _dns-update._udp SRV record. It must be configured to be authoritative for "services.arpa", and to accept updates from hosts on local networks for names under "services.arpa" without authentication.

A server configured in this way will be able to successfully accept and process Registrations from services that send Registrations. However, no constraints will be applied, and this means that the test server will accept internally inconsistent Registrations, and will not stop two Registrations, sent by different services, that claim the same name(s), from overwriting each other.
2.3.3. How to allow services to update standard RFC2136-compliant servers

Ordinarily Registrations will fail when sent to any non-Registration Protocol server because the zone being updated is "services.arpa", and no DNS server that is not a Registration Protocol server should normally be configured to be authoritative for "services.arpa". Therefore, a service that sends a Registration can tell that the receiving server does not support the Registration Protocol, but does support RFC2136, because the RCODE will either be NOTZONE, NOTAUTH or REFUSED, or because there is no response to the update request (when using the anycast address).

In this case a service MAY attempt to register itself using regular RFC2136 DNS updates. To do so, it must discover default registration zone and the DNS server designated to receive updates for that zone, as described earlier using the _dns-update._udp SRV record. It can then make the update using the port and host pointed to by the SRV record, and should use appropriate constraints to avoid overwriting competing records. Such updates are out of scope for the DNSSD Registration Protocol, and a service that implements the DNSSD Registration Protocol MUST first attempt to use the Registration Protocol to register itself, and should only attempt to use RFC2136 backwards compatibility if that fails.

2.4. How to secure it

Traditional DNS update is secured using the TSIG protocol, which uses a secret key shared between the client (which issues the update) and the server (which authenticates it). This model does not work for automatic service registration.

The goal of securing the DNS-SD Registration Protocol is to provide the best possible security given the constraint that service registration has to be automatic. It is possible to layer more operational security on top of what we describe here, but what we describe here improves upon the security of mDNS. The goal is not to provide the level of security of a network managed by a skilled operator.

2.4.1. First-Come First-Served Naming

First-Come First-Serve naming provides a limited degree of security: a service that registers its service using DNS-SD Registration protocol is given ownership of a name for an extended period of time based on the key used to authenticate the DNS Update. As long as the registration service remembers the Service Instance Name and the key...
used to register that Service Instance Name, no other service can add or update the information associated with that Service Instance Name.

2.4.1.1. Service Behavior

The service generates a public/private key pair. This key pair MUST be stored in stable storage; if there is no writable stable storage on the client, the client MUST be pre-configured with a public/private key pair that can be used.

When sending DNS updates, the service includes a KEY record containing the public portion of the key in each Host Description update. The update is signed using SIG(0), using the private key that corresponds to the public key in the KEY record. The lifetimes of the records in the update is set using the EDNS(0) Update Lease option.

The lifetime of the DNS-SD PTR, SRV, A, AAAA and TXT records [RFC6763] is typically set to two hours. This means that if a device is disconnected from the network, it does not appear in the user interfaces of devices looking for services of that type for too long.

However, the lifetime of its KEY record should be set to a much longer time, typically 14 days. The result of this is that even though a device may be temporarily unplugged, disappearing from the network for a few days, it makes a claim on its name that lasts much longer.

This way, even if a device is unplugged from the network for a few days, and its services are not available for that time, no other rogue device can come along and immediately claim its name the moment it disappears from the network. In the event that a device is unplugged from the network and permanently discarded, then its name is eventually cleaned up and made available for re-use.

2.4.2. Registration Server Behavior

The Registration server checks each update in the Registration to see that it contains a Service Discovery update, a Service Description update, and a Host Description update.

An update is a Service Discovery update if it contains

- exactly one RRset update,
- which is for a PTR RR,
- which points to a Service Instance Name
- for which an update is present in the Registration.
An update is a Service Description update if, for the appropriate Service Instance Name, it contains

- exactly one "Delete all RRsets from a name" update,
- exactly one SRV RRset update,
- one or more TXT RRset updates,
- and the target of the SRV record update references a hostname for which there is a Host Description update in the Registration.

An update is a Host Description update if, for the appropriate hostname, it contains

- exactly one "Delete all RRsets from a name" update,
- A or AAAA RR update(s)
- a KEY RR update that adds a KEY RR that contains the public key corresponding to the private key that was used to sign the message,
- there is a Service Instance Name update in the Registration that updates an SRV RR so that it points to the hostname being updated by this update.

A Registration MUST include at least one Service Name update, at least one Service Description update, and exactly one Host Description update. An update message that does not contain a Registration. An update message that contains any other updates, or any update constraints, is not a Registration. Such messages should either be processed as regular RFC2136 updates, including access control checks and constraint checks, if supported, or else rejected with RCODE=REFUSED.

Note that if the definitions of each of these update types are followed carefully, this means that many things that look very much like Registrations nevertheless are not. For example, a Registration that contains an update to a Service Name and an update to a Service Instance Name, where the Service Name does not reference the Service Instance Name, is not a valid Registration message, but may be a valid RFC2136 update.

Assuming that an update message has been validated with these conditions and is a valid Registration, the server checks that the name in the Host Description update exists. If so, then the server checks to see if the KEY record on the name is the same as the KEY record in the update. If it is not, then the server MUST reject the Registration with the YXDOMAIN RCODE.

Otherwise, the server validates the update using SIG(0) on the public key in the KEY record of the Host Description update. If the validation fails, the server MUST reject the rejection rejected
with the REFUSED RCODE. Otherwise, the update is considered valid and authentic, and is processed according to the method described in RFC2136. The status that is returned depends on the result of processing the update.

The server MAY add a Reverse Mapping that corresponds to the Host Description. This is not required because the Reverse Mapping serves no protocol function, but it may be useful for debugging, e.g. in annotating network packet traces or logs.

The server MAY apply additional criteria when accepting updates. In some networks, it may be possible to do out-of-band registration of keys, and only accept updates from pre-registered keys. In this case, an update for a key that has not been registered should be rejected with the REFUSED RCODE.

There are at least two benefits to doing this rather than simply using normal SIG(0) DNS updates. First, the same registration protocol can be used in both cases, so both use cases can be addressed by the same service implementation. Second, the registration protocol includes maintenance functionality not present with normal DNS updates.

Note that the semantics of using the Registration Protocol in this way are different than for typical RFC2136 implementations: the KEY used to sign the update in the Registration Protocol only allows the client to update records that refer to its Host Description. RFC2136 implementations do not normally provide a way to enforce a constraint of this type.

The server may also have a dictionary of names or name patterns that are not permitted. If such a list is used, updates for Service Instance Names that match entries in the dictionary are rejected with YXDOMAIN.

2.5. TTL Consistency

All RRs within an RRset are required to have the same TTL (Clarifications to the DNS Specification [RFC2181], Section 5.2). In order to avoid inconsistencies, the Registration Protocol places restrictions on TTls sent by services and requires that Registration Protocol Servers enforce consistency.

Services sending Registrations MUST use consistent TTls in all RRs within the Registration.
Registration Protocol servers MUST check that the TTLs for all RRs within the Registration are the same. If they are not, the Registration MUST be rejected with a REFUSED RCODE.

Additionally, when adding RRs to an RRset, for example when processing Service Discovery records, the server MUST use the same TTL on all RRs in the RRset. How this consistency is enforced is up to the implementation.

2.6. Maintenance

2.6.1. Cleaning up stale data

Because the DNS-SD registration protocol is automatic, and not managed by humans, some additional bookkeeping is required. When an update is constructed by the client, it MUST include include an EDNS(0) Update Lease Option [I-D.sekar-dns-ul]. The Update Lease Option contains two lease times: the Update Lease Time and the Instance Lease Time.

These leases are promises, similar to DHCP leases [RFC2131], from the client that it will send a new update for the service registration before the lease time expires. The Update Lease time is chosen to represent the time after the update during which the registered records other than the KEY record should be assumed to be valid. The Instance Lease time represents the time after the update during which the KEY record should be assumed to be valid.

The reasoning behind the different lease times is discussed in the section on first-come, first-served naming Section 2.4.1. DNS-SD Registration Protocol servers may be configured with limits for these values. A default limit of two hours for the Update Lease and 14 days for the SIG(0) KEY are currently thought to be good choices. Clients that are going to continue to use names on which they hold leases should update well before the lease ends, in case the registration service is unavailable or under heavy load.

The Registration Protocol server MUST include an EDNS(0) Update Lease option in the response if the lease time proposed by the service has been shortened. The service MUST check for the EDNS(0) Update Lease option in the response and MUST use the lease times from that option in place of the options that it sent to the server when deciding when to update its registration.

Clients should assume that each lease ends N seconds after the update was first transmitted, where N is the lease duration. Servers should assume that each lease ends N seconds after the update that was successfully processed was received. Because the server will always...
receive the update after the client sent it, this avoids the possibility of misunderstandings.

DNS-SD Registration Protocol servers MUST reject updates that do not include an EDNS(0) Update Lease option. Dual-use servers MAY accept updates that don’t include leases, but SHOULD differentiate between DNS-SD registration protocol updates and other updates, and MUST reject updates that are known to be DNS-SD Registration Protocol updates if they do not include leases.

2.6.2. Sleep Proxy

Another use of Service Registration Protocol is for devices that sleep to reduce power consumption.

In this case, in addition to the DNS Update Lease option [I-D.sekar-dns-ul] described above, the device includes an EDNS(0) OWNER Option [I-D.cheshire-edns0-owner-option].

The EDNS(0) Update Lease option constitutes a promise by the device that it will wake up before this time elapses, to renew its registration and thereby demonstrate that it is still attached to the network. If it fails to renew the registration by this time, that indicates that it is no longer attached to the network, and its registration (except for the KEY in the Host Description) should be deleted.

The EDNS(0) OWNER Option indicates that the device will be asleep, and will not be receptive to normal network traffic. When a DNS server receives a DNS Update with an EDNS(0) OWNER Option, that signifies that the Registration Protocol server should set up a proxy for any IPv4 or IPv6 address records in the DNS Update message. This proxy should send ARP or ND messages claiming ownership of the IPv4 and/or IPv6 addresses in the records in question. In addition, proxy should answer future ARP or ND requests for those IPv4 and/or IPv6 addresses, claiming ownership of them. When the DNS server receives a TCP SYN or UDP packet addressed to one of the IPv4 or IPv6 addresses for which it proxying, it should then wake up the sleeping device using the information in the EDNS(0) OWNER Option. At present version 0 of the OWNER Option specifies the "Wake-on-LAN Magic Packet" that needs to be sent; future versions could be extended to specify other wakeup mechanisms.

Note that although the authoritative DNS server that implements the DNSSSD Service Registration Protocol function need not be on the same link as the sleeping host, the Sleep Proxy must be on the same link.
3. Security Considerations

DNS-SD Service Registration Protocol updates have no authorization semantics other than first-come, first-served. This means that if an attacker from outside of the administrative domain of the server knows the server’s IP address, it can in principle send updates to the server that will be processed successfully. Servers should therefore be configured to reject updates from source addresses outside of the administrative domain of the server.

For Anycast updates, this validation must be enforced by every router that connects the CDN to the unconstrained portion of the network. For TCP updates, the initial SYN-SYN+ACK handshake prevents updates being forged from off-network. In order to ensure that this handshake happens, Service Discovery Protocol servers MUST NOT accept 0-RTT TCP payloads.

Note that these rules only apply to the validation of DNS-SD registration protocol updates. A server that accepts updates from DNS-SD registration protocol clients may also accept other DNS updates, and those DNS updates may be validated using different rules. However, in the case of a DNS service that accepts automatic updates, the intersection of the DNS-SD service registration update rules and whatever other update rules are present must be considered very carefully.

For example, a normal, authenticated RFC2136 update to any RR that was added using the Registration protocol, but that is authenticated using a different key, could be used to override a promise made by the registration protocol, by replacing all or part of the service registration information with information provided by a different client. An implementation that allows both kinds of updates should not allow updates to records added by Registrations using different authentication and authorization credentials.

4. Privacy Considerations

5. Acknowledgments

Thanks to Toke Hoeiland-Joergensen for a thorough technical review, to Tamara Kemper for doing a nice developmental edit, Tim Wattenberg for doing a service implementation at the Montreal Hackathon at IETF 102, and [...] more reviewers to come, hopefully.
6. References

6.1. Normative References


6.2. Informative References


Internet-Draft        Service Registration Protocol            July 2018

to Replace the AppleTalk Name Binding Protocol (NBP)",
RFC 6760, DOI 10.17487/RFC6760, February 2013,

RFC 6761, DOI 10.17487/RFC6761, February 2013,

[ RFC 6762 ]  Cheshire, S. and M. Krochmal, "Multicast DNS", RFC 6762,
DOI 10.17487/RFC6762, February 2013,

[ RFC 7228 ]  Bormann, C., Ersue, M., and A. Keranen, "Terminology for
Constrained-Node Networks", RFC 7228,
DOI 10.17487/RFC7228, May 2014,

[I-D.ietf-dnssd-hybrid]
    Cheshire, S., "Discovery Proxy for Multicast DNS-Based
    Service Discovery", draft-ietf-dnssd-hybrid-08 (work in
    progress), March 2018.

[I-D.ietf-dnssd-push]
    Pusateri, T. and S. Cheshire, "DNS Push Notifications",
draft-ietf-dnssd-push-14 (work in progress), March 2018.

[I-D.cheshire-dnssd-roadmap]
    Cheshire, S., "Service Discovery Road Map",
draft-cheshire-dnssd-roadmap-01 (work in progress), March 2018.

[I-D.cheshire-edns0-owner-option]
    Cheshire, S. and M. Krochmal, "EDNS0 OWNER Option",
draft-cheshire-edns0-owner-option-01 (work in progress), July
    2017.

[ZC]  Cheshire, S. and D. Steinberg, "Zero Configuration
    Networking: The Definitive Guide", O'Reilly Media, Inc.,

Authors’ Addresses
Stuart Cheshire
Apple Inc.
One Apple Park Way
Cupertino, California 95014
USA

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

Ted Lemon
Nibbhaya Consulting
P.O. Box 958
Brattleboro, Vermont 05302
United States of America

Email: mellon@fugue.com